Politenico di Milano

Advanced cybersecurity topics

2019-2020

Write-up Chall1

Author:
Bova Salvatore

Person code:
10499292

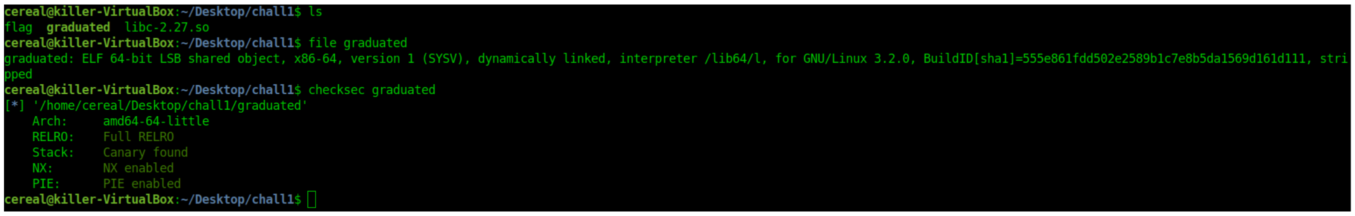
January 30, 2020

1 Introduction

The main goal of the challenge is to read the content of the *flag* file in the same folder of the given executable.
N.B. The challenge run on ubuntu 18.04 and the *libc-2.27.so* file is provided.

2 Static analysis

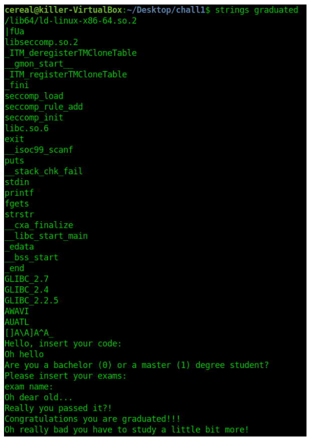
First of all I checked the binary (*graduated*) with the *file* command to determine the file type, being an ELF file, I decided to check its properties and security options running the bash script *checksec*.



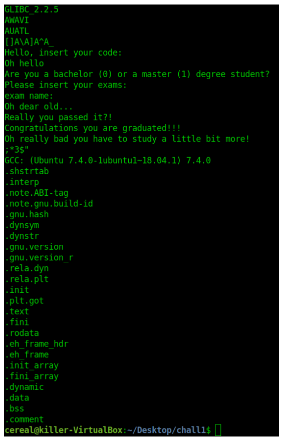
```
cereal@killer-VirtualBox:~/Desktop/chall1$ ls
flag  graduated  libc-2.27.so
cereal@killer-VirtualBox:~/Desktop/chall1$ file graduated
graduated: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0, BuildID[sha1]=555e861fdd502e2589b1c7e8b5da1569d161d111, stri
pped
cereal@killer-VirtualBox:~/Desktop/chall1$ checksec graduated
[*] '/home/cereal/Desktop/chall1/graduated'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
cereal@killer-VirtualBox:~/Desktop/chall1$ 
```

Figure 1: Output of the commands

I also decided to check, through the command *strings*, if the flag is used somewhere in the program, but without success.



```
cereal@killer-VirtualBox:~/Desktop/chall1$ strings graduated
/lib64/ld-linux-x86-64.so.2
|fUa
libseccomp.so.2
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
_fini
seccomp_load
seccomp_rule_add
seccomp_init
libc.so.6
exit
__isoc99_scanf
puts
__stack_chk_fail
stdin
printf
fgets
strstr
__cxa_finalize
__libc_start_main
_edata
__bss_start
_end
GLIBC_2.7
GLIBC_2.4
GLIBC_2.2.5
AWAVI
AUATL
[]A\A]A^A_
Hello, insert your code:
Oh hello
Are you a bachelor (0) or a master (1) degree student?
Please insert your exams:
exam name:
Oh dear old...
Really you passed it?!
Congratulations you are graduated!!!
Oh really bad you have to study a little bit more!
```

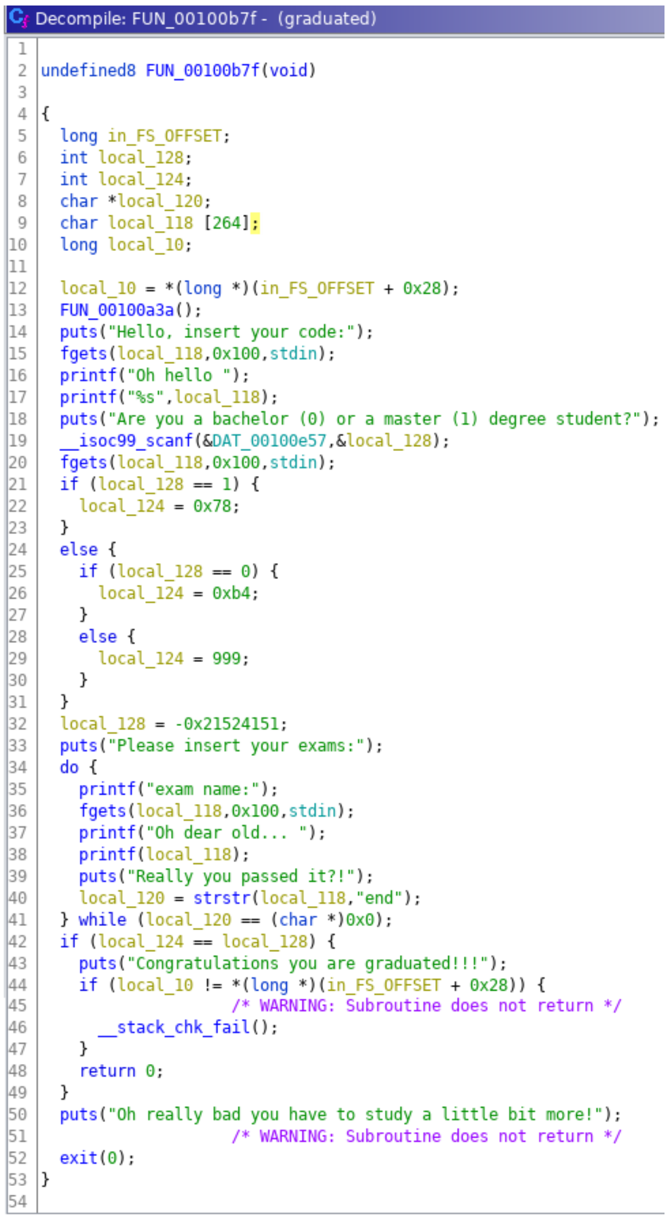


```
GLIBC_2.2.5
AWAVI
AUATL
[]A\A]A^A_
Hello, insert your code:
Oh hello
Are you a bachelor (0) or a master (1) degree student?
Please insert your exams:
exam name:
Oh dear old...
Really you passed it?!
Congratulations you are graduated!!!
Oh really bad you have to study a little bit more!
;*3$"
GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rela.dyn
.rela.plt
.init
.plt.got
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.init_array
.fini_array
.dynamic
.data
.bss
.comment
cereal@killer-VirtualBox:~/Desktop/chall1$ 
```

Figure 2: Output of strings command

2.1 Analysis with Ghidra

At this point I decided to use a disassembler tool, to disassemble end decompile the executable.



```

undefined8 FUN_00100b7f(void)

{
  long in_FS_OFFSET;
  int local_128;
  int local_124;
  char *local_120;
  char local_118 [264];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  FUN_00100a3a();
  puts("Hello, insert your code:");
  fgets(local_118,0x100,stdin);
  printf("Oh hello ");
  printf("%s",local_118);
  puts("Are you a bachelor (0) or a master (1) degree student?");
  __isoc99_scanf(&DAT_00100e57,&local_128);
  fgets(local_118,0x100,stdin);
  if (local_128 == 1) {
    local_124 = 0x78;
  }
  else {
    if (local_128 == 0) {
      local_124 = 0xb4;
    }
    else {
      local_124 = 999;
    }
  }
  local_128 = -0x21524151;
  puts("Please insert your exams:");
  do {
    printf("exam name:");
    fgets(local_118,0x100,stdin);
    printf("Oh dear old... ");
    printf(local_118);
    puts("Really you passed it?!");
    local_120 = strstr(local_118,"end");
  } while (local_120 == (char *)0x0);
  if (local_124 == local_128) {
    puts("Congratulations you are graduated!!!");
    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
      __stack_chk_fail();
    }
    return 0;
  }
  puts("Oh really bad you have to study a little bit more!");
                    /* WARNING: Subroutine does not return */
  exit(0);
}

```

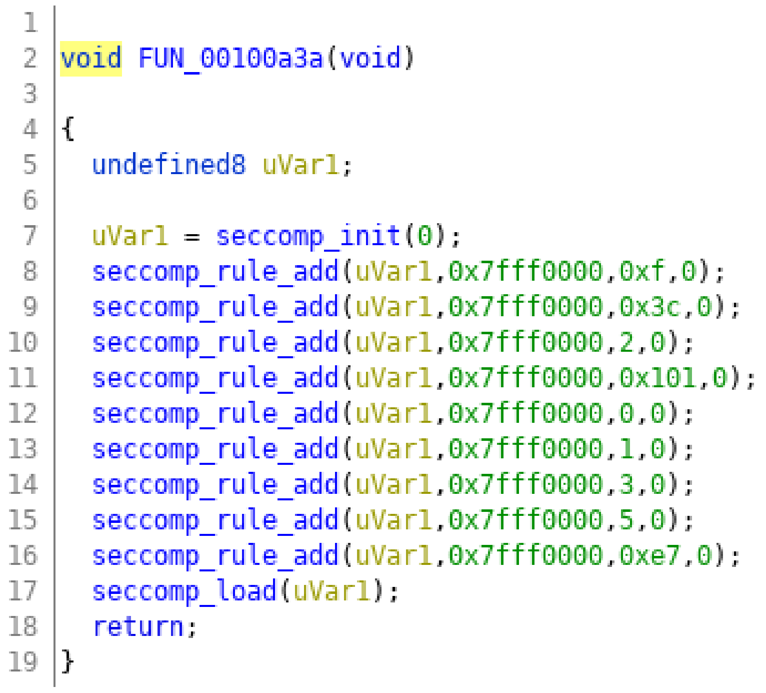
Figure 3: Decompiled code.

Figure 3 shows the decompiled code. Analyzing it, we can see that the executable asks to the user some inputs of size at most 0x100, or rather 256, through the *fgets* and saves them in an array of char of 264 elements, so there is no chance for a buffer overflow vulnerability.
Then the executable prints to standard output, with the *printf* function, our provided input but forgetting, at line *38*, to use a placeholder!

So we are in presence of a string format bug vulnerability, moreover is placed in a while loop that ends when we decide, so we have the possibility to leak the whole content of the stack and write wherever we want, a good candidate it would be the GOT, but we have FULL RELRO active, so we can't write there.

So in this case, a good way to proceed it would be to overwrite the return instruction pointer, with the founded vulnerability, to execute a Ret2libc or a ROP attack.

But first, let's analyze the *FUN_00100a3a()* function.



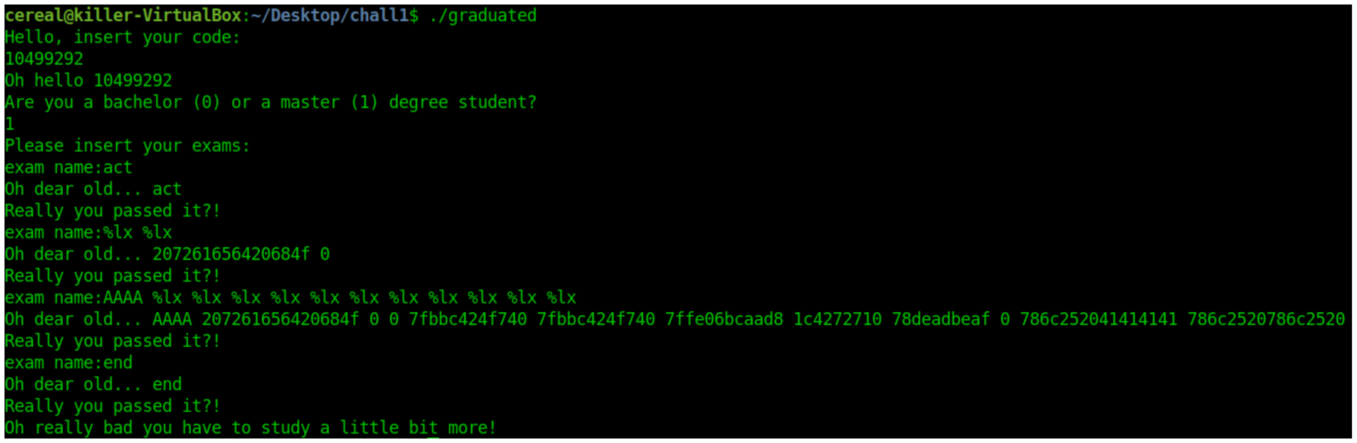
```

void FUN_00100a3a(void)

{
  undefined8 uVar1;

  uVar1 = seccomp_init(0);
  seccomp_rule_add(uVar1,0x7fff0000,0xf,0);
  seccomp_rule_add(uVar1,0x7fff0000,0x3c,0);
  seccomp_rule_add(uVar1,0x7fff0000,2,0);
  seccomp_rule_add(uVar1,0x7fff0000,0x101,0);
  seccomp_rule_add(uVar1,0x7fff0000,0,0);
  seccomp_rule_add(uVar1,0x7fff0000,1,0);
  seccomp_rule_add(uVar1,0x7fff0000,3,0);
  seccomp_rule_add(uVar1,0x7fff0000,5,0);
  seccomp_rule_add(uVar1,0x7fff0000,0xe7,0);
  seccomp_load(uVar1);
  return;
}
```

Figure 4: FUN_00100a3a()

From figure 4, we can deduce that the libseccomp library is used. This library provide a syscall filtering mechanism, so checking the third parameters of the functions we can know which are the only system call that we can use. No one of the *exec* function family is available, so we can't spawn a shell, but we could try to build a ROP chain through multiple uses of the string format bug vulnerability, to execute in row a open, a read and a write on the file flag (they are all system calls admitted).
We need also to overwrite the value of the *local_128* variable, otherwise we can't reach the ret instruction, but we'll ended up to an exit, that will make our ROP chain useless, but this is not a big problem, because we can overwrite the variable easily always with the string format bug vulnerability.

3 Dynamic analysis

Just to confirm our deductions, we can launch the binary, few times and analyze it with GDB.

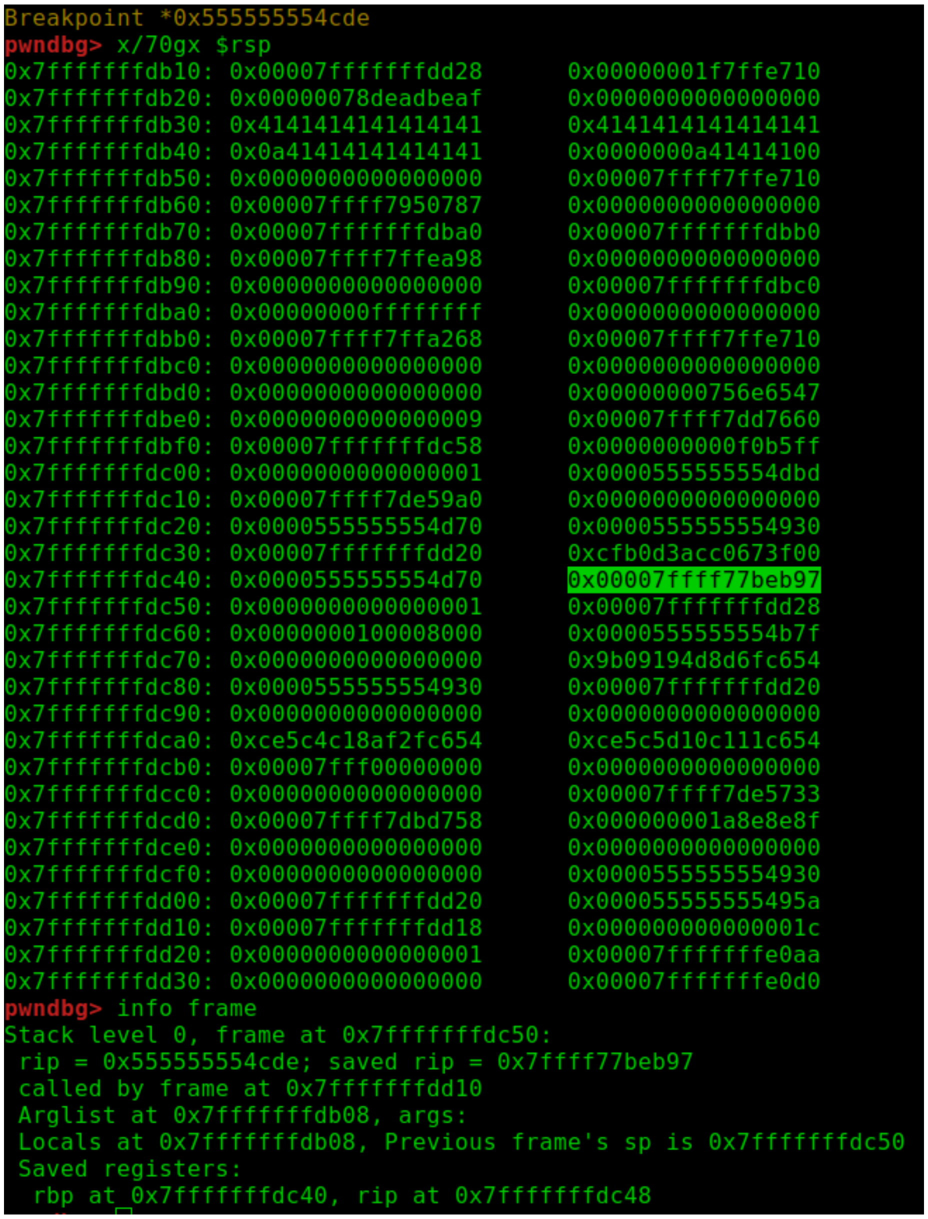


```
cereal@killer-VirtualBox:~/Desktop/chall1$ ./graduated
Hello, insert your code:
10499292
Oh hello 10499292
Are you a bachelor (0) or a master (1) degree student?
1
Please insert your exams:
exam name:act
Oh dear old... act
Really you passed it?!
exam name:%lx %lx
Oh dear old... 207261656420684f 0
Really you passed it?!
exam name:AAAA %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx
Oh dear old... AAAA 207261656420684f 0 0 7fbbc424f740 7fbbc424f740 7ffe06bcaad8 1c4272710 78deadbeaf 0 786c252041414141 786c2520786c2520
Really you passed it?!
exam name:end
Oh dear old... end
Really you passed it?!
Oh really bad you have to study a little bit more!
```

Figure 5: A run of the program.

As predicted, providing as input the placeholder *%lx*, because we are on 64 bit, we can leak the content of the stack.

3.1 Analysis with GDB



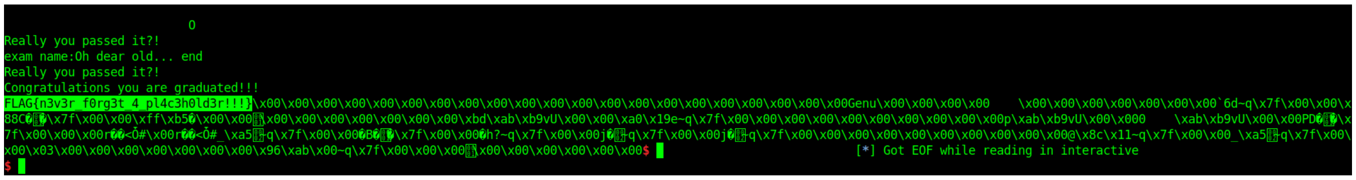
```
Breakpoint *0x555555554cde
pwndbg> x/70gx $rsp
0x7fffffffdb10: 0x00007fffffffdd28      0x00000001f7ffe710
0x7fffffffdb20: 0x00000078deadbeaf      0x0000000000000000
0x7fffffffdb30: 0x4141414141414141      0x4141414141414141
0x7fffffffdb40: 0x0a41414141414141      0x0000000a41414100
0x7fffffffdb50: 0x0000000000000000      0x00007ffff7ffe710
0x7fffffffdb60: 0x00007ffff7950787      0x0000000000000000
0x7fffffffdb70: 0x00007fffffffdba0      0x00007fffffffdbb0
0x7fffffffdb80: 0x00007ffff7ffea98      0x0000000000000000
0x7fffffffdb90: 0x0000000000000000      0x00007fffffffdbc0
0x7fffffffdba0: 0x00000000ffffffff      0x0000000000000000
0x7fffffffdbb0: 0x00007ffff7ffa268      0x00007ffff7ffe710
0x7fffffffdbc0: 0x0000000000000000      0x0000000000000000
0x7fffffffdbd0: 0x0000000000000000      0x00000000756e6547
0x7fffffffdbe0: 0x0000000000000009      0x00007ffff7dd7660
0x7fffffffdbf0: 0x00007fffffffdc58      0x0000000000f0b5ff
0x7fffffffdc00: 0x0000000000000001      0x0000555555554dbd
0x7fffffffdc10: 0x00007ffff7de59a0      0x0000000000000000
0x7fffffffdc20: 0x0000555555554d70      0x0000555555554930
0x7fffffffdc30: 0x00007fffffffdd20      0xcfb0d3acc0673f00
0x7fffffffdc40: 0x0000555555554d70      0x00007ffff77beb97
0x7fffffffdc50: 0x0000000000000001      0x00007fffffffdd28
0x7fffffffdc60: 0x0000000100008000      0x0000555555554b7f
0x7fffffffdc70: 0x0000000000000000      0x9b09194d8d6fc654
0x7fffffffdc80: 0x0000555555554930      0x00007fffffffdd20
0x7fffffffdc90: 0x0000000000000000      0x0000000000000000
0x7fffffffdca0: 0xce5c4c18af2fc654      0xce5c5d10c111c654
0x7fffffffdcb0: 0x00007fff00000000      0x0000000000000000
0x7fffffffdcc0: 0x0000000000000000      0x00007ffff7de5733
0x7fffffffdcd0: 0x00007ffff7dbd758      0x000000001a8e8e8f
0x7fffffffdce0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcf0: 0x0000000000000000      0x0000555555554930
0x7fffffffdd00: 0x00007fffffffdd20      0x000055555555495a
0x7fffffffdd10: 0x00007fffffffdd18      0x000000000000001c
0x7fffffffdd20: 0x0000000000000001      0x00007fffffffe0aa
0x7fffffffdd30: 0x0000000000000000      0x00007fffffffe0d0
pwndbg> info frame
Stack level 0, frame at 0x7fffffffdc50:
 rip = 0x555555554cde; saved rip = 0x7ffff77beb97
 called by frame at 0x7fffffffdd10
 Arglist at 0x7fffffffdb08, args:
 Locals at 0x7fffffffdb08, Previous frame's sp is 0x7fffffffdc50
 Saved registers:
  rbp at 0x7fffffffdc40, rip at 0x7fffffffdc48
```

Figure 6: Stack's content.

In figure 6 we can see that we can leak addresses of the stack and of the libc, useful to defeat ASLR, to compute the address of useful gadget in the libc provided and to write on the saved rip position, in this case *0x7fffffffdc48*.

4 Conclusion

I built and attached to this write-up a script (*solution.py*) to exploit the vulnerability and read the flag, using pwntools and building the ROP chain through the string format bug vulnerabilities. It is well commented so for any further explanation there is the possibility to consult it.
Running it few times, we can see that we get the flag!



```
                          O
Really you passed it?!
exam name:Oh dear old... end
Really you passed it?!
Congratulations you are graduated!!!
FLAG{n3v3r_f0rg3t_4_pl4c3h0ld3r!!!}\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00Genu\x00\x00\x00\x00    \x00\x00\x00\x00\x00\x00\x00`6d~q\x7f\x00\x00\x
88C�␁�\x7f\x00\x00\xff\xb5�\x00\x00␁\x00\x00\x00\x00\x00\x00\x00\xbd\xab\xb9vU\x00\x00\xa0\x19e~q\x7f\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00p\xab\xb9vU\x00\x000    \xab\xb9vU\x00\x00PD�␁�\x
7f\x00\x00\x00r��<Ǒ#\x00r��<Ǒ#_\xa5␂q\x7f\x00\x00�B�␁�\x7f\x00\x00�h?~q\x7f\x00\x00j�␂q\x7f\x00\x00j�␂q\x7f\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x8c\x11~q\x7f\x00\x00_\xa5␂q\x7f\x00\
x00\x03\x00\x00\x00\x00\x00\x00\x00\x96\xab\x00~q\x7f\x00\x00\x00␁\x00\x00\x00\x00\x00\x00$                          [*] Got EOF while reading in interactive
$
```

Figure 7: Output of the script