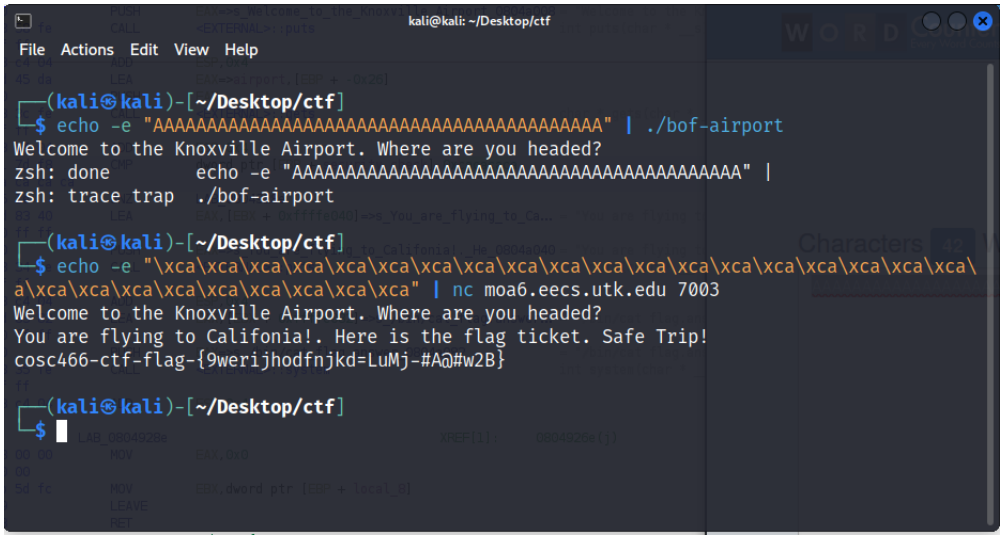
Preston Beaty and Noah Morris
COSC 466
2/21/2023
1- Airport

To begin with, I started off by loading the executable file into Ghidra. Once it was loaded into Ghidra, I decompiled the executable in order to see the contents of memory addresses and their actions. This same process could have been done through the objdump command line. Below were my results:

```
08049244 e8 47 fe ff ff    CALL    <EXTERNAL>::setbuf                          void setbuf(FILE * __stream, cha...
08049249 83 c4 08          ADD     ESP,0x8
0804924c 8d 83 08 e0 ff ff LEA     EAX,[EBX + 0xffffe008]=>s_Welcome_to_the_Knoxv... = "Welcome to the Knoxville Airp...
08049252 50                PUSH    EAX=>s_Welcome_to_the_Knoxville_Airport_0804a008 = "Welcome to the Knoxville Airp...
08049253 e8 58 fe ff ff    CALL    <EXTERNAL>::puts                            int puts(char * __s)
08049258 83 c4 04          ADD     ESP,0x4
0804925b 8d 45 da          LEA     EAX=>airport,[EBP + -0x26]
0804925e 50                PUSH    EAX
0804925f e8 3c fe ff ff    CALL    <EXTERNAL>::gets                            char * gets(char * __s)
08049264 83 c4 04          ADD     ESP,0x4
08049267 81 7d f8 ca ca ca ca CMP  dword ptr [EBP + security_check],0xcacacaca
0804926e 75 1e             JNZ     LAB_0804928e
08049270 8d 83 40 e0 ff ff LEA     EAX,[EBX + 0xffffe040]=>s_You_are_flying_to_Ca... = "You are flying to Califonia!...
08049276 50                PUSH    EAX=>s_You_are_flying_to_Califonia!._He_0804a040 = "You are flying to Califonia!...
08049277 e8 34 fe ff ff    CALL    <EXTERNAL>::puts                            int puts(char * __s)
0804927c 83 c4 04          ADD     ESP,0x4
0804927f 8d 83 82 e0 ff ff LEA     EAX,[EBX + 0xffffe082]=>s_/bin/cat_flag.answer... = "/bin/cat flag.answer"
08049285 50                PUSH    EAX=>s_/bin/cat_flag.answer_0804a082        = "/bin/cat flag.answer"
08049286 e8 35 fe ff ff    CALL    <EXTERNAL>::system                          int system(char * __command)
0804928b 83 c4 04          ADD     ESP,0x4
```

As you can see, the structure shows the prompted question “Welcome to the Knoxville Airport” followed by a compare located at address 0x08049267. This compares the value to 0xcacacaca. At this point, I knew the value of the compare statement. The next step was to determine the number of characters needed in order to create a segfault.



As you can see above, after determining the number of characters necessary to create the default, I then replaced it with ‘\xca’ and added four more in order to overwrite the comparison value. After this point, the code then continued to execute in order to show the flag store in /bin/cat_flag.answer.

Within our code, we are taking the value within EBX, you can see this on 0x08049270, which is our input given to the executable, and compare it to the value of 0xcacacaca. However, in our case, we are overwriting the buffer we have allocated at EBX and overflowing it onto EBP which is the base pointer. Since in our case there is only one local variable. You can think of the buffer being perfectly in between ESP and EBP. Once the EBP is overwritten due to the overflow it then also leaks onto the return pointer directly underneath. Here is where we cause the segmentation fault to leak into the value that is being compared. This is how the buffer overflow attack succeeds.