Challenge 1: check_passwd

The vulnerabilities in this program are the possible buffer overflow in ask_magic_word which uses strcpy without bounds checking and the format string vulnerability in main which uses printf (str_buf, flag) where str_buf contains a format specifier. buf[0×16] also takes in 22 bytes.
It's possible to overflow the magic word because fgets allows buf to have 22 characters and strcpy doesn't have any bound checking.
The correct magic word is 10 characters or 9 counting from 0. The 15th index is the null terminator, so adding whitespaces after Q in the magic word up to the 15th index should allow strcmp to pass. Adding %s in the 16th index will cause the overflow and read the string at that memory address which was confirmed to be the correct memory address using %p. This returned the flag *flag{arctic-leaf}*.

Challenge 2: segmentation

This challenge was very simple. Just needed to identify the process id of the program through

```
ps -e -o ruser,pid,comm
```

This showed the process id of the program to be 2473. Then to cause a segmentation fault, the SIGSEGV signal can be used as such:

```
kill -11 2473
```

Doing so returned the flag *flag{warm-bargain}*

Challenge 3: out_of_canons

The solution to this challenge was to make use of symlinks in order to trick the program into reading the admin's text file while inputting user's text file.
However, because the validate_path function checks for symlinks between files, we need to create a symbolic link between directories, in this case soft linking the admin directory to a directory I created 'admin_link' inside the user directory as such:

```
ln -s ../admin ./admin_link
```

The program constructs the path as `./user/admin_link/Elvi.txt` which resolves to `./admin/Elvi.txt`. By creating a symlink at the directory level, we effectively create a new path the program can follow without detecting any malicious patterns. This resulted in the flag *flag{human-model}*

Challenge 4: whats_an_inode

This challenge is similar to the previous except this time we can directly create a soft link between the text files since the program doesn't check for symlinks. I create a symlink between Carlos.txt and a new file in user directory and obtained the flag *flag{chilly-hardball}*

Challenge 5: crypt_flag

Running `cat admin.bin` outputs `Salted__` followed by a bunch of ASCII characters implying the program has been encrypted with OpenSSL's AES encryption and has been salted.
In order to decrypt the file, I would have to make use of hashcat to brute force the password.
The plan was to generate a bunch of passwords and write them to a file using hashcat's rule based attack as well as the brute force method. Since I didn't have permission to write in the directory I decided to create the passwords file in `/home/admin2700`. However doing this in the VM was quite a slow process so I did this step in my local machine, utilizing my GPU to speed up the process.
For the rule-based passwords I used the following commands:

```
hashcat -a 0 --stdout /usr/share/dict/cracklib-small -r
/usr/share/hashcat/rules/*.rule > generated_passwords.txt
```

```
hashcat -a 0 --stdout /usr/share/dict/cracklib-small -r
/usr/share/hashcat/rules/hybrid/*.rule >> generated_passwords.txt
```

For the brute force method I gathered all the mask combinations written in .hcmask files from the mask directory in hashcat to a single fine called mask_rules.txt. Then I wrote a shell script to generate passwords based on the masks:

```
#!/bin/bash

# Path to your masks file
MASKS_FILE="mask_rules.txt"
```

```
# Path to output file
OUTPUT_FILE="masks_generated_passwords.txt"

# Ensure the output file is empty before starting
> "$OUTPUT_FILE"

# Read each line from the masks file and use it to generate passwords
while IFS= read -r mask; do
    echo "Generating passwords for mask: $mask"
    hashcat -D 1,2 --stdout -a 3 "$mask" >> "$OUTPUT_FILE"
done < "$MASKS_FILE"

echo "Password generation complete. Results saved in $OUTPUT_FILE"
```

However this was starting to take up a lot of disk space so I had to end the script early and wasn't able to generate all the possible passwords. Looking back on it, this might've not been the right approach anyway.
With all the passwords generated, I copied the files to the lab VM and started writing a script to use all the passwords as input to openssl decryption command hoping one of them will be able to decrypt the program. For the brute force attack, the script was:

```
#!/bin/bash

# Path to encrypted file and decrypted output
ENCRYPTED_FILE="/ctf/crypt_flag/JJ8AQA/admin.bin"
DECRYPTED_FILE="/home/admin2700/decrypted_file.txt"

# Path to the generated passwords file
PASSWORD_FILE="/home/admin2700/masks_generated_passwords.txt"

# Generate passwords using Hashcat's mask attack mode
while IFS= read -r password; do
        echo "Trying password: $password"
        # Attempt to decrypt using OpenSSL
        if openssl enc -d -aes-128-cbc -pbkdf2 -pass pass:"$password" -
iter 10000 -salt -in "$ENCRYPTED_FILE" -out "$DECRYPTED_FILE"; then
                # Check if decryption was successful
                echo "Password $password worked, checking contents..."
                # Check if decrypted file contains flag
                if grep -q "flag" "$DECRYPTED_FILE"; then
                        echo "Success. Password is: $password"
                        cat "$DECRYPTED_FILE"
                        exit 0
                else
                        echo "Password correct, but no flag found.
Continuing..."
                fi
        fi
```

```
done < "$PASSWORD_FILE"

echo "Failed to crack the password"
```

and for the rule based attack, the script was:

```
#!/bin/bash

# Path to encrypted file and decrypted output
ENCRYPTED_FILE="/ctf/crypt_flag/JJ8AQA/admin.bin"
DECRYPTED_FILE="/home/admin2700/decrypted_file.txt"

# Path to the generated passwords file
PASSWORDS_FILE="/home/admin2700/generated_passwords.txt"

# Iterate over each password in the generated list
shuf "$PASSWORDS_FILE" | while IFS= read -r password; do
        echo "Trying password: $password"
        # Attempt to decrypt using OpenSSL
        openssl enc -d -aes-128-cbc -pbkdf2 -pass pass:"$password" -iter
10000 -salt -in "$ENCRYPTED_FILE" -out "$DECRYPTED_FILE"

        # Check if decryption was successful
        if [ $? -eq 0 ]; then
                echo "Password $password worked, checking contents..."
                # Check if decrypted file contains flag
                if grep -q "flag{" "$DECRYPTED_FILE"; then
                        echo "Success. Password is: $password"
                        cat "$DECRYPTED_FILE"
                        exit 0
                else
                        echo "Password correct, but no flag found in file.
Continuing"
                fi
        fi
done < "$PASSWORDS_FILE"

echo "Failed to crack the password"
```

running this to decrypt user.bin worked. The script successfully guessed the password to be password123 and was able to decrypt the contents of user.bin. But before it was able to, the script was guessing the wrong passwords which were being accepted as the correct password. So I added a if condition to check if the decrypted content included the word flag.
I also used randomization to potentially optimize run-time by shuffling the passwords around using shuf which didn't look like it shuffled the passwords much if at all.

However all this work was in vain, as the scripts were not able to find the correct password for admin.bin even after a long time. They just kept going through all the passwords not finding a match and after a long time, something went wrong with ssh connection, resulting in the connection being closed and I didn't want to start all over again.

Challenge 6: num_sum

The numbers array is 256 bytes and the array is stored next to 'a' in the stack. The difference in memory addresses between the array and 'a' is 260, so 4 more bytes after array should overwrite a. Since the program declares a first, 'a' is at a higher address than array, therefore to access and overwrite a, we need to use the add function to overflow the array.
Using the debug option to see the addresses of k, i and answer[24], the memory layout was as follows:

higher addr		lower addr
k	i	answer

k was 4 bytes after i (curr idx) and i was 24 bytes after answer.
inputting more than 24 bytes to answer, since fgets allows 42 bytes to be copied into answer from stdin, should overwrite i. More than that (28 bytes) should overwrite k (the number that actually gets put in the array). I being the current index, overwriting i should allow me to keep adding more numbers in the array past 64, which should allow me to overwrite a, since a lies past the 64th byte in the array.
After a few attempts, I learned that stdin will allow 25 bytes of input to add to the array, any more than that will cause a segmentation fault. I crafted this input to add to the array: `3743e678901234567890123 49`. atoi will read till e and only accept 3743. the rest should cause a overflow. However what this did was create 58 elements in the array with the 58th element being 3743. If I could do this to create 65 elements in the array, with the 65th number being 3743, that should cause a to be overwritten to be 3743. After a few attempts, I used the string `3743e6789012345678901230A` which created 66 elements with the 66th element being 3743. I though if I could remove one number from the array, the values would shift upwards or to the left but that wasn't the case. In the end I couldn't figure which number or letter at the end of the string will only create 65 elements which should've been where 'a' is.

Challenge 7: random_guess

the answer was at the address: 0xffb18ffc and since win is declared at the beginning in main, it would be at a lower address than the variables in play(). Initially I tried a buffer overflow approach, inputting large numbers. Then I tried inputting multiple

numbers separated by spaces and commas. After a few such attempts, I noticed Lilith was only picking 1 after the first few runs. Even after closing the program and rerunning it, this persisted and Lilith just kept picking 1 after picking random numbers the first 4 or 5 times. On one of the following attempts, I was able to guess the first random number she was thinking which was 12, so after that all I had to do was input 1's until she reached the point where all the numbers she was thinking were also 1 and doing that I beat her 6 times and she gave me the flag *flag{energetic-upload}*