Macks Treasure (RE) Writeup

Overview

General Summary

Mack's Treasure is an RE challenge that involves analyzing a Linux .ELF file. It begins with a user getting a random file and they need to perform some tasks on it to get it ready to run on a linux machine. Once the challenger gets past that they then are presented a program in which they need to enter a password. The password is determined by a serious of functions some involve encryption. The encryptions are xor based encryption but have small differences to make them different from each other. One of the sections of the password requires the other functions to be solved. The password is the flag that they solve for. The encryptions are expected to be passed by running python scripts or any other coding scripts. It is encouraged to do that instead of doing it by hand.

> It is encouraged that the program is analyzed in a VM even though the program is not malicious.
>
> > Use of a decompiler is highly recommended.

Learning Goals

- Understand how to perform basic Software RE Analysis.
- Learn how to use ida/ghidra and organize reversed code.
- Learn to understand C function calls.
- Learn how to crack simple encryptions.
- Learn how to perform basic static analysis on executable files.
- Learn how to use automate decryption using python or other programming languages.

Terminology

- The flag is split up into different parts called chunks. I explain how each chunk is found and what each chunk contains. I also give the final flag at the end.
- I did not include any scripting to decipher the code as this is to be created by the challenger and that there is multiple ways you could write the scripts to do so. You can see the encryption algorithms in my screen shots and the c source code.
- I gave the c code to the program in the code appendix as a good reference in case a challenger gets stuck and a mentor needs to quickly figure out how the program works.

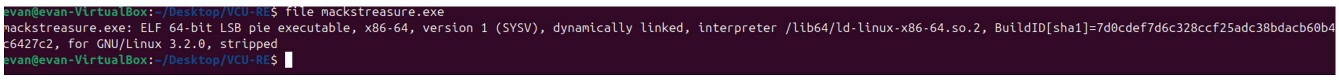
Challenge Goals

- FIND THE FLAG
- Be able to explain the functionality of the program.
- Bonus: Automate all the xor decryption by writing programs.
- Bonus; Write a C program to generate the same srand() values. Have them learn that different compilers use different algorithms for rand() and srand().

Solution

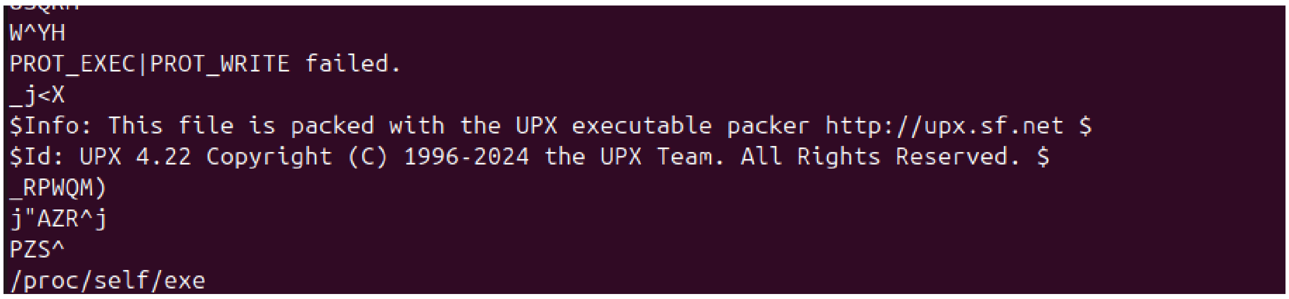
Part 1: Static Analysis and Obfuscation

1. The challenge starts off as a program called **mackstreasure.exe** that is able to be downloaded on GitHub.
    - The program is actually a .ELF file. It is labeled as an .exe to throw off the challenger. It is important that they perform basic static analysis.
    - If the challenger tries to run it on a windows machine it will give an error (which is a good hint that it is not an EXE file)
2. Upon the user running `file mackstreasure.exe` on a Linux terminal they can see that the file is an ELF executable.



```
evan@evan-VirtualBox:~/Desktop/VCU-RE$ file mackstreasure.exe
mackstreasure.exe: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=7d0cdef7d6c328ccf25adc38bdacb60b4
c6427c2, for GNU/Linux 3.2.0, stripped
evan@evan-VirtualBox:~/Desktop/VCU-RE$
```

    - Any other method that reveals the actual file extension is valid.
3. The user needs to run `chmod +x mackstreasure.exe` so the file has executable permissions.
4. Upon running `strings mackstreasure.exe` The user can see some obfuscated strings and strings saying the program has been packed with UPX packer.
    - There are other methods to see it has been packed which are all valid.
    - However, if the challenger tries to run it in a debugger/decompiler they will not have an easy time trying to analyze the packed executable.



```
W^YH
PROT_EXEC|PROT_WRITE failed.
_j<X
$Info: This file is packed with the UPX executable packer http://upx.sf.net $
$Id: UPX 4.22 Copyright (C) 1996-2024 the UPX Team. All Rights Reserved. $
_RPWQM)
j"AZR^j
PZS^
/proc/self/exe
```

5. The user needs to download the UPX packer. They need to run `upx -d mackstreasure.exe` to unpack the executable.
    - They may need this command to get upx install `sudo apt-get install upx-ucl`
    - I had to use `upx-ucl` instead of `upx` when running upx commands.

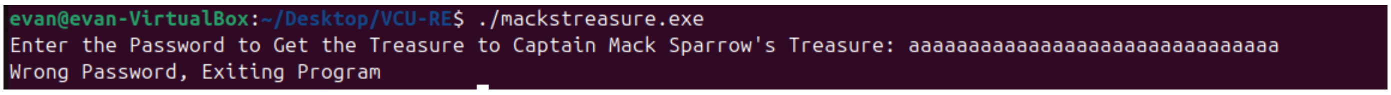
6. The program is now ready to be reversed. Debug symbols have been stripped so there may not be any obvious names for functions and variables. Use of a decompiler will be very helpful to finding the main function.

Part 2: Finding Hints and Password Generation

Once the user has unpacked the code and got it running on a Linux VM they are now able to start analyzing the program.

Running the program

1. When the user runs `./mackstreasure.exe` They will be prompted to enter a password.
2. If the password is incorrect an error message is printed and the program exits.



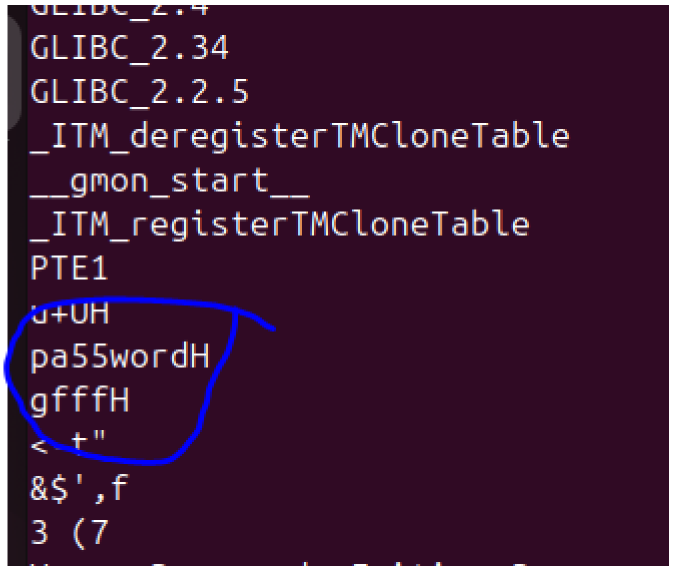
```
evan@evan-VirtualBox:~/Desktop/VCU-RE$ ./mackstreasure.exe
Enter the Password to Get the Treasure to Captain Mack Sparrow's Treasure: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Wrong Password, Exiting Program
```

Hints

There are two hints that are used to try and get the challenger to be familiar with what they are trying to find.

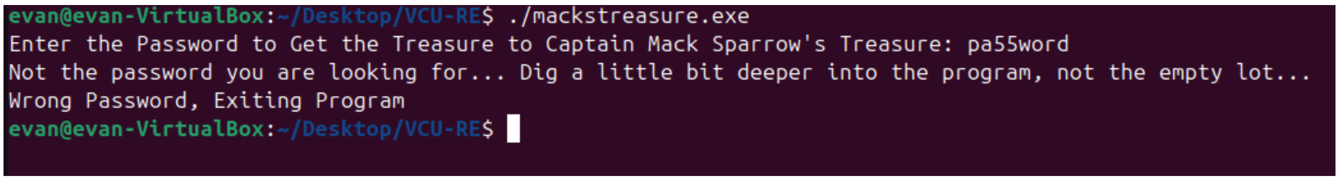
Hint 1: Digging Deeper

1. This hint is very easy and sort of acts as a red herring. The user will be able to see a statically set string. `pa55word`



```
GLIBC_2.34
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
pa55wordH
gfffH
<-t"
&$',f
3 (7
```

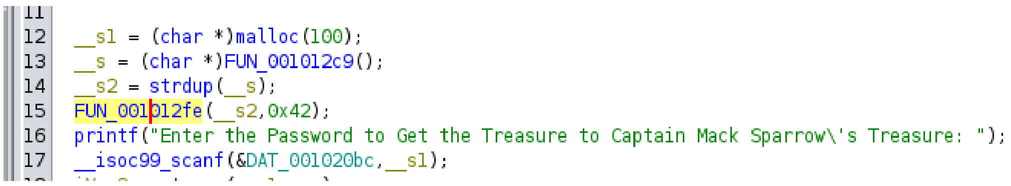
2. When they enter the string they get a print out of the hint which encourages them to keep looking into the program.



```
evan@evan-VirtualBox:~/Desktop/VCU-RE$ ./mackstreasure.exe
Enter the Password to Get the Treasure to Captain Mack Sparrow's Treasure: pa55word
Not the password you are looking for... Dig a little bit deeper into the program, not the empty lot...
Wrong Password, Exiting Program
evan@evan-VirtualBox:~/Desktop/VCU-RE$ 
```

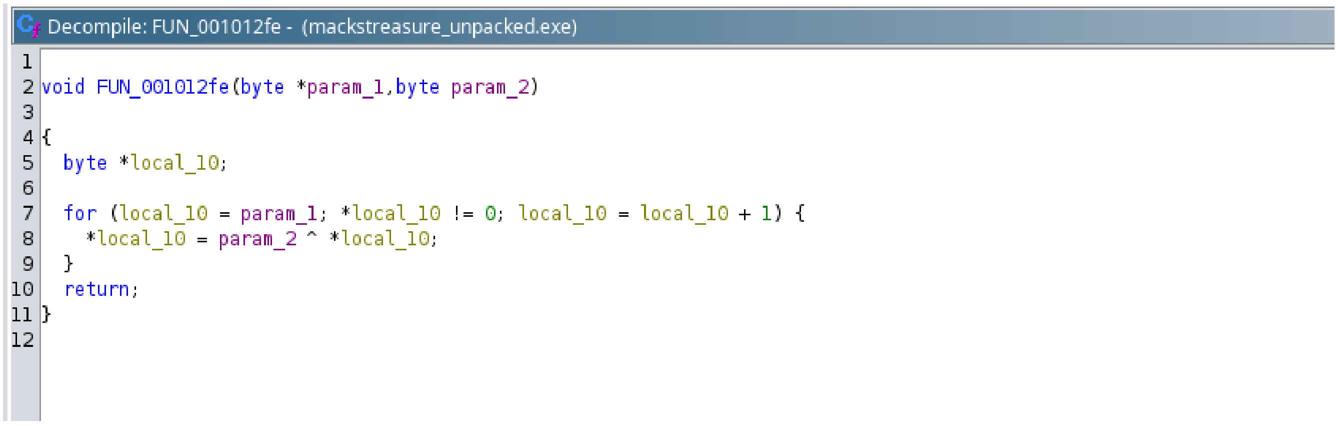
Hint 2: Xor encryption

1. This hint is to introduce the xor encryption algorithm that is going to be used through out later parts of the program.
2. The user will see a function call that passes in the number 0x42 and the string `pa55word` .



```
  __s1 = (char *)malloc(100);
  __s = (char *)FUN_001012c9();
  __s2 = strdup(__s);
  FUN_001012fe(__s2,0x42);
  printf("Enter the Password to Get the Treasure to Captain Mack Sparrow\'s Treasure: ");
  __isoc99_scanf(&DAT_001020bc,__s1);
```

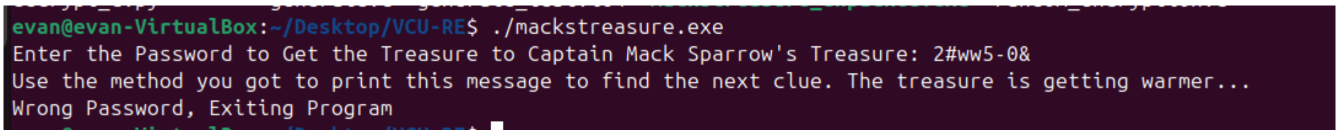
3. When they look at the function they see that every character is being xored against the 0x42. This should be an easy encryption to crack.



Decompile: FUN_001012fe - (mackstreasure_unpacked.exe)

```

void FUN_001012fe(byte *param_1,byte param_2)

{
  byte *local_10;

  for (local_10 = param_1; *local_10 != 0; local_10 = local_10 + 1) {
    *local_10 = param_2 ^ *local_10;
  }
  return;
}

```

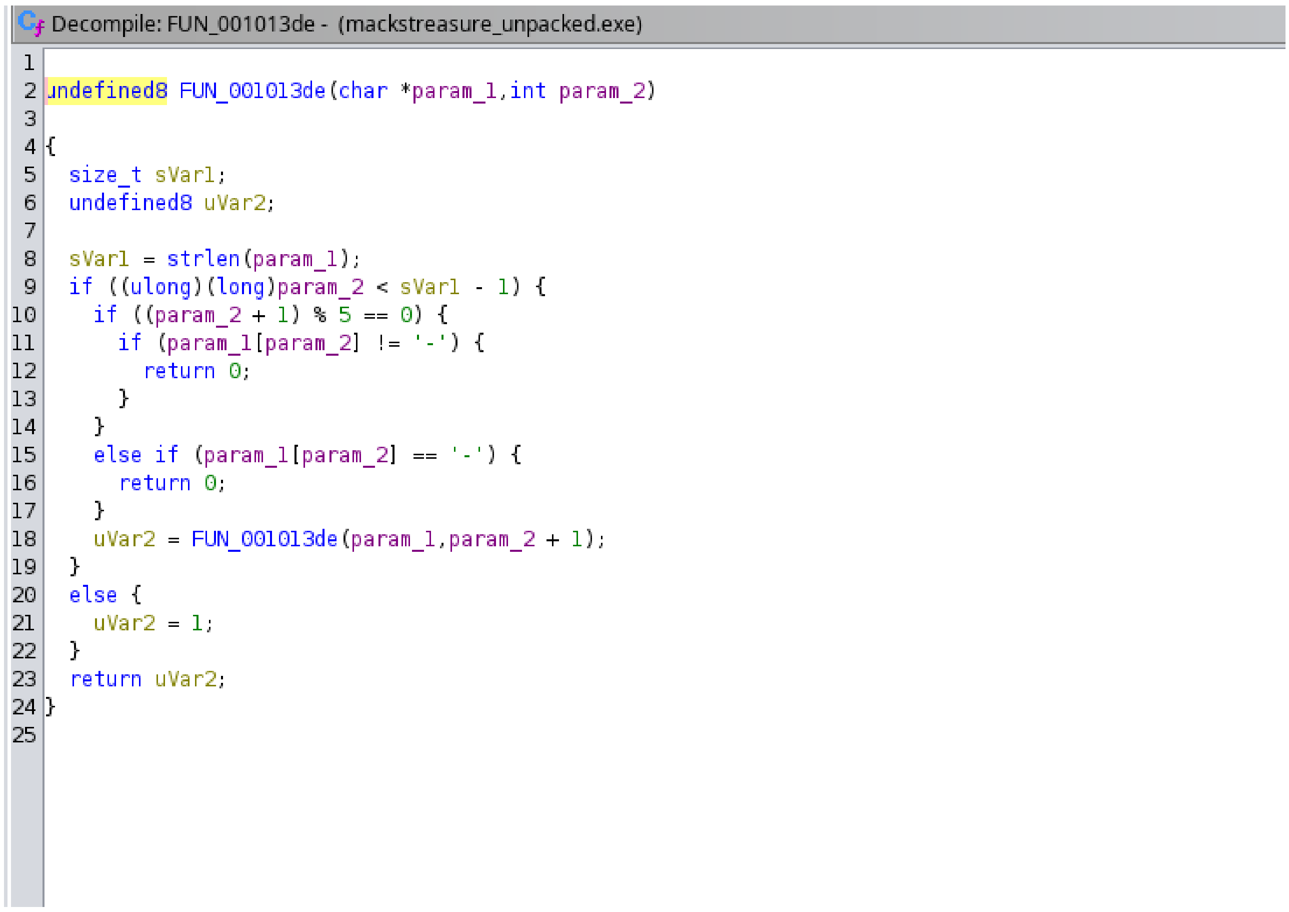
4. When they enter the string created by the xor algorithm they get a print out of the next hint that tells them that they should remember how this method is used. Xor encryption is used in later parts of the challenge.



```
evan@evan-VirtualBox:~/Desktop/VCU-RE$ ./mackstreasure.exe
Enter the Password to Get the Treasure to Captain Mack Sparrow's Treasure: 2#ww5-0&
Use the method you got to print this message to find the next clue. The treasure is getting warmer...
Wrong Password, Exiting Program
```

Password Structure

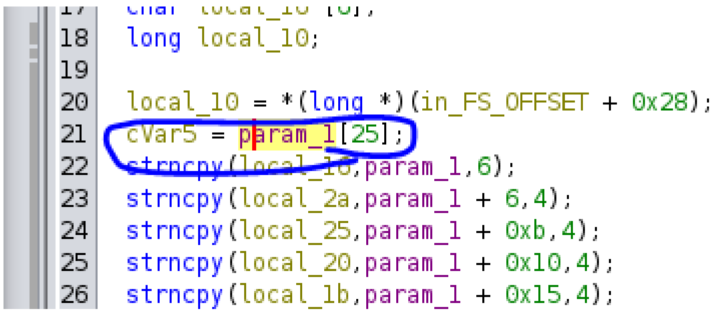
1. By looking at the one of the functions they should be able to see dashes and that the program checks for them every 4 characters.
   - The challenger may mistake them as underscores. But they could always check the ascii value to double check. The format will be like this (xxxx-xxxx-xxxx-xxxx)



Decompile: FUN_001013de - (mackstreasure_unpacked.exe)

```

undefined8 FUN_001013de(char *param_1,int param_2)

{
  size_t sVar1;
  undefined8 uVar2;

  sVar1 = strlen(param_1);
  if ((ulong)(long)param_2 < sVar1 - 1) {
    if ((param_2 + 1) % 5 == 0) {
      if (param_1[param_2] != '-') {
        return 0;
      }
    }
    else if (param_1[param_2] == '-') {
      return 0;
    }
    uVar2 = FUN_001013de(param_1,param_2 + 1);
  }
  else {
    uVar2 = 1;
  }
  return uVar2;
}

```

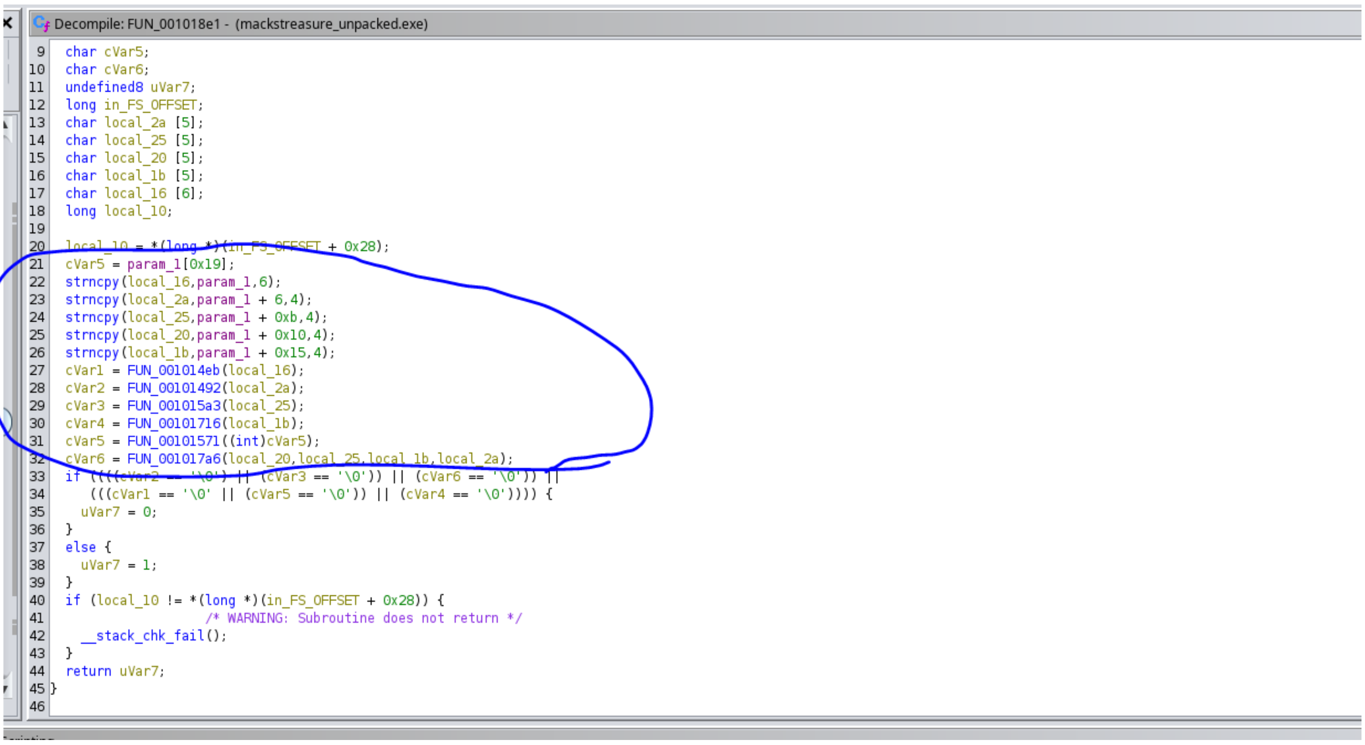
2. This should help the user see that it is similar to that of a serial code. If they pay close attention they should see that it doesn't even look for the full flag. By understanding c functions they should be able to see that the user input is being divided into different chunks.
3. The beginning *dfend{* and the end *}* are determined by xoring the characters with 0x42. This is to just obfuscate the flag.
   - The challenger may assume these letters from knowing the flag format. This will give them a slight advantage.
4. The challenger has many different ways they can find the total length. There is a if conditional that checks this but there is other functions that make use of the total string length. (**Total String Length is 26**)



```
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  cVar5 = param_1[25];
  strncpy(local_16,param_1,6);
  strncpy(local_2a,param_1 + 6,4);
  strncpy(local_25,param_1 + 0xb,4);
  strncpy(local_20,param_1 + 0x10,4);
  strncpy(local_1b,param_1 + 0x15,4);
```

5. The challenger will come across a function that shows that the user input is split into a new string with the dfend{} stripped off. They then see the new string get split up into 4 different sections. This again will be easy to figure out if the challenger looks up some of the c functions.
6. Upon further investigating they start to see that each chunk of the string is sent to a different function. These functions show how that chunk of the password has been created. By looking at the

function can understanding the use of strncpy() the challenger can determine which part of the flag the chunk is.


Decompile: FUN_001018e1 - (mackstreasure_unpacked.exe)

```
  char cVar5;
  char cVar6;
  undefined8 uVar7;
  long in_FS_OFFSET;
  char local_2a [5];
  char local_25 [5];
  char local_20 [5];
  char local_1b [5];
  char local_16 [6];
  long local_10;
  
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  cVar5 = param_1[0x19];
  strncpy(local_16,param_1,6);
  strncpy(local_2a,param_1 + 6,4);
  strncpy(local_25,param_1 + 0xb,4);
  strncpy(local_20,param_1 + 0x10,4);
  strncpy(local_1b,param_1 + 0x15,4);
  cVar1 = FUN_001014eb(local_16);
  cVar2 = FUN_00101492(local_2a);
  cVar3 = FUN_001015a3(local_25);
  cVar4 = FUN_00101716(local_1b);
  cVar5 = FUN_00101571((int)cVar5);
  cVar6 = FUN_001017a6(local_20,local_25,local_1b,local_2a);
  if ((((cVar2 == '\0') || (cVar3 == '\0')) || (cVar6 == '\0')) ||
     (((cVar1 == '\0' || (cVar5 == '\0')) || (cVar4 == '\0')))) {
    uVar7 = 0;
  }
  else {
    uVar7 = 1;
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar7;
}
```

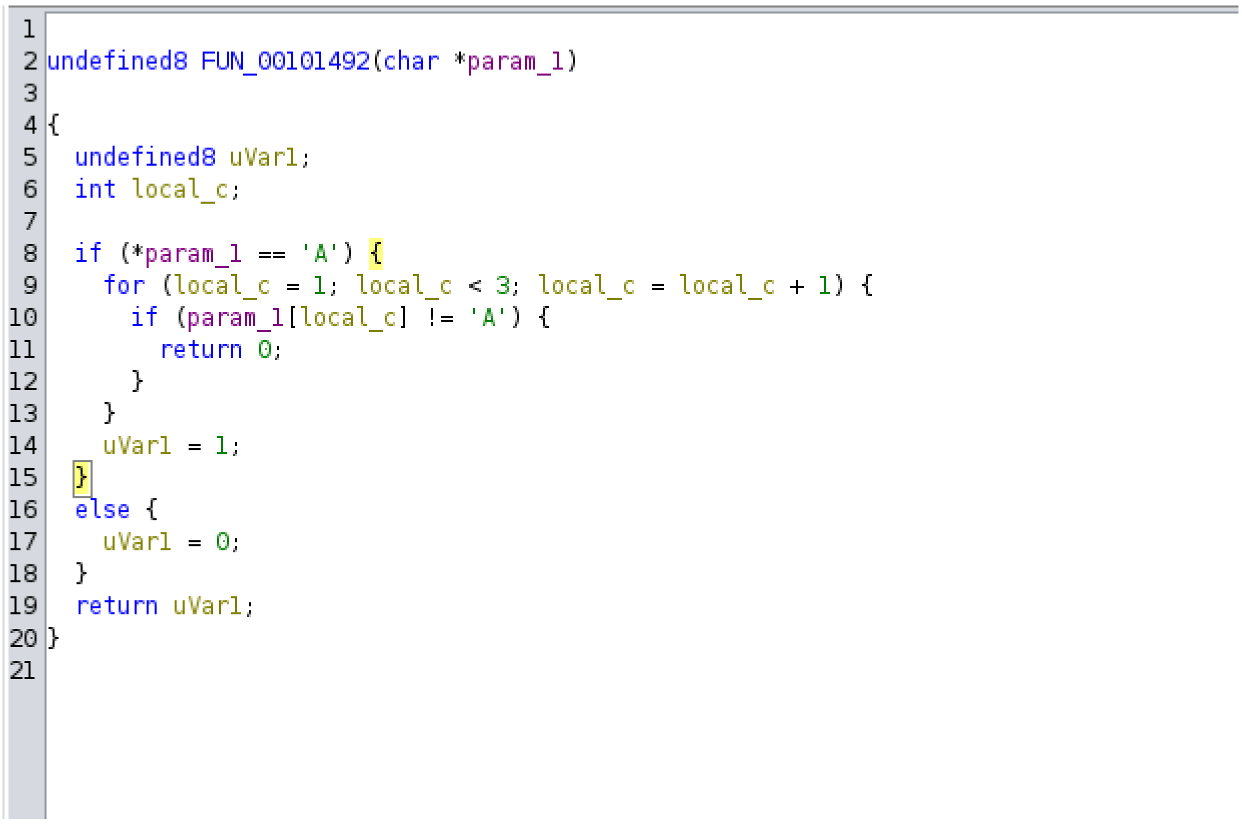
7. The challenge then shifts its focus to trying to crack the password. The challenger should figure out that they need to look at these functions and figure out what exactly is happening. Using a decompiler will make this much easier.

Part 3: Cracking the Password

Once the user has discovered how the password is structured they now have to try and figure out how to get the correct password through all the function calls in the program.

Section 1: Repeating Letters

1. This password chunk is focused on understanding what the c/asm code is doing. There is no encryption involved for this chunk.
2. The user can see a for loop and a conditional that checks if each letter of the chunk sent in is *A*.



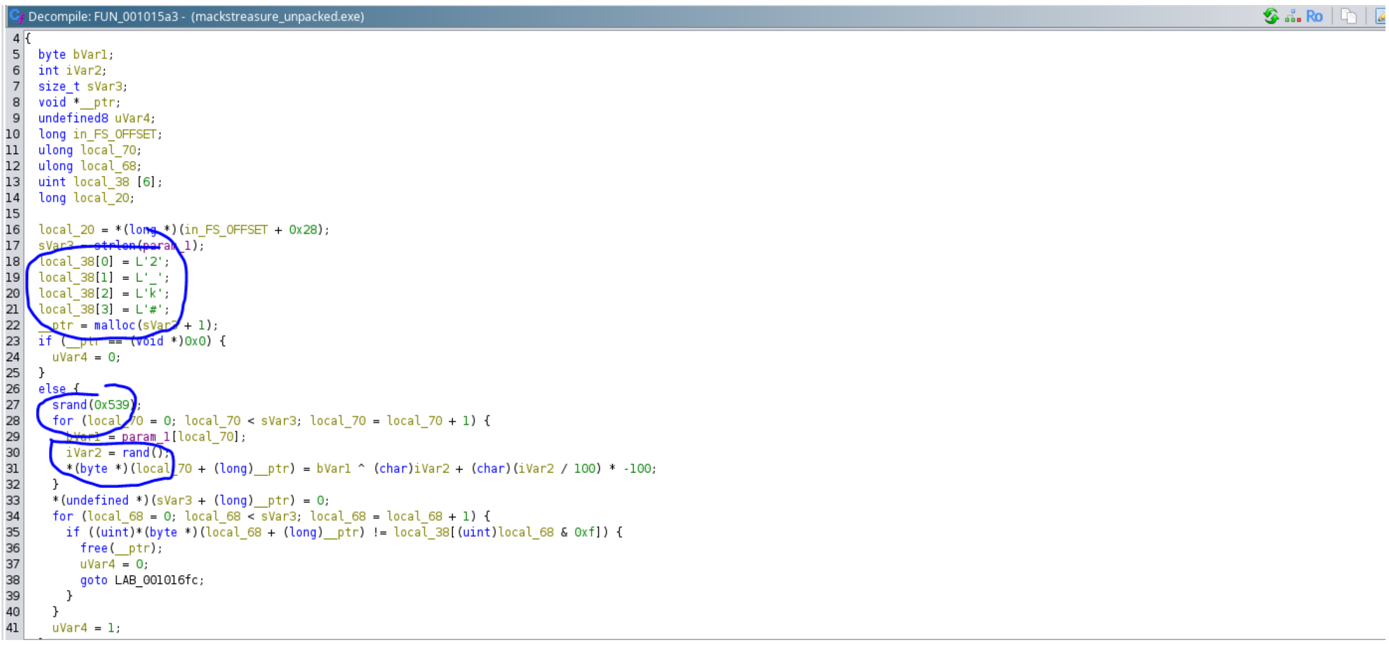
```

undefined8 FUN_00101492(char *param_1)

{
  undefined8 uVar1;
  int local_c;

  if (*param_1 == 'A') {
    for (local_c = 1; local_c < 3; local_c = local_c + 1) {
      if (param_1[local_c] != 'A') {
        return 0;
      }
    }
    uVar1 = 1;
  }
  else {
    uVar1 = 0;
  }
  return uVar1;
}

```

3. By understanding the c/asm code the challenger should put together that this chunk contains the chunk **AAAA**

Section 2: Seeded Xor Encryption

1.This password chunk uses xor encryption but features a seeded password with the *srand()* and *rand()* c functions.
2. The user can see that the program is performing a xor operation using the rand() function. They can see that srand(1337) is called meaning that there is a seed for the *random* number created.
3. The challenger can get the seeded value by stepping through ida or they could try and rewrite the c code.
- Stepping through the ida code is preferred. srand() and rand() values are very compiler dependent meaning that different compilers handle the pseudorandom algorithm differently.

4. The challenger can see that there are hex values preset. They are to assume that they are used to check for the encrypted answer.
5. The challenger can then reverse the algorithm with the information they have gathered to get the correct chunk of code.



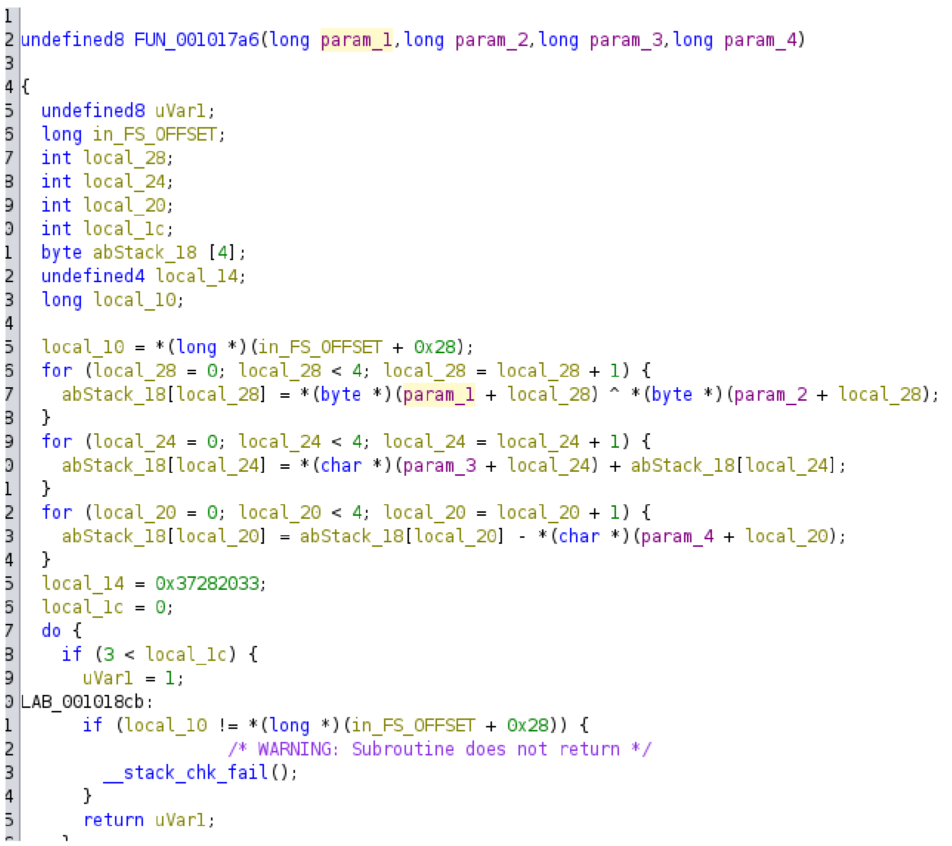
Decompile: FUN_001015a3 - (mackstreasure_unpacked.exe)

```c
{
  byte bVar1;
  int iVar2;
  size_t sVar3;
  void *__ptr;
  undefined8 uVar4;
  long in_FS_OFFSET;
  ulong local_70;
  ulong local_68;
  uint local_38 [6];
  long local_20;

  local_20 = *(long *)(in_FS_OFFSET + 0x28);
  sVar3 = strlen(param_1);
  local_38[0] = L'2';
  local_38[1] = L'_';
  local_38[2] = L'k';
  local_38[3] = L'#';
  __ptr = malloc(sVar3 + 1);
  if (__ptr == (void *)0x0) {
    uVar4 = 0;
  }
  else {
    srand(0x539);
    for (local_70 = 0; local_70 < sVar3; local_70 = local_70 + 1) {
      bVar1 = param_1[local_70];
      iVar2 = rand();
      *(byte *)(local_70 + (long)__ptr) = bVar1 ^ (char)iVar2 + (char)(iVar2 / 100) * -100;
    }
    *(undefined *)(sVar3 + (long)__ptr) = 0;
    for (local_68 = 0; local_68 < sVar3; local_68 = local_68 + 1) {
      if ((uint)*(byte *)(local_68 + (long)__ptr) != local_38[(uint)local_68 & 0xf]) {
        free(__ptr);
        uVar4 = 0;
        goto LAB_001016fc;
      }
    }
    uVar4 = 1;
```

6. When the challenger correctly reverses the encryption they should get the chunk **capt**

Section 3: Final Chunk - Encryption with All the Other Chunks.

1. This is the most complicated section. It requires all the other sections to be solved before it can be solved.
2. The function performs encryption in which all the other chunks are needed to get the correct value.
3. They can see that these chunks are passed in when looking at the function call.
4. Once the challenger understands the algorithm they can reverse it to get the correct final chunk.

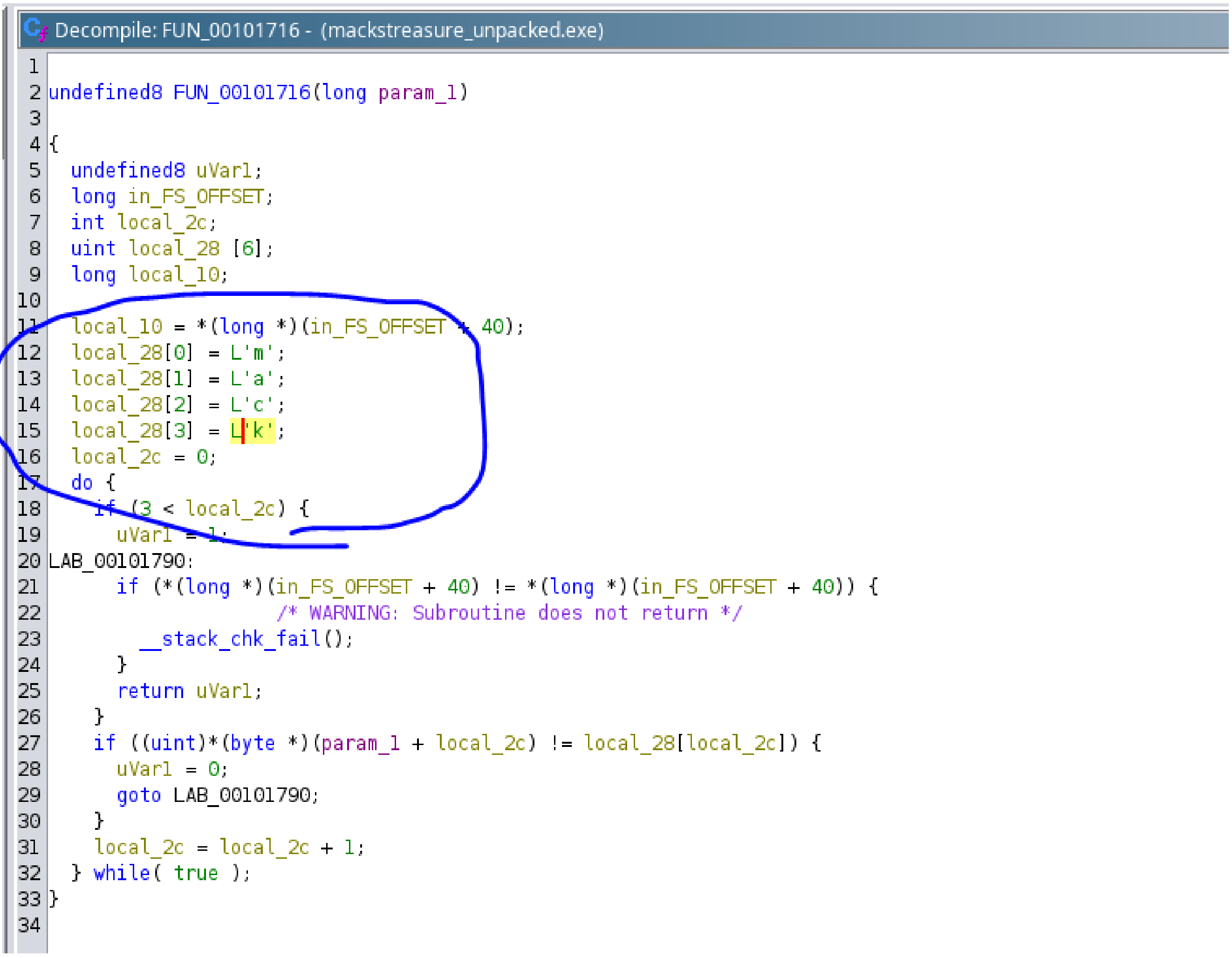


```c

undefined8 FUN_001017a6(long param_1,long param_2,long param_3,long param_4)

{
  undefined8 uVar1;
  long in_FS_OFFSET;
  int local_28;
  int local_24;
  int local_20;
  int local_1c;
  byte abStack_18 [4];
  undefined4 local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  for (local_28 = 0; local_28 < 4; local_28 = local_28 + 1) {
    abStack_18[local_28] = *(byte *)(param_1 + local_28) ^ *(byte *)(param_2 + local_28);
  }
  for (local_24 = 0; local_24 < 4; local_24 = local_24 + 1) {
    abStack_18[local_24] = *(char *)(param_3 + local_24) + abStack_18[local_24];
  }
  for (local_20 = 0; local_20 < 4; local_20 = local_20 + 1) {
    abStack_18[local_20] = abStack_18[local_20] - *(char *)(param_4 + local_20);
  }
  local_14 = 0x37282033;
  local_1c = 0;
  do {
    if (3 < local_1c) {
      uVar1 = 1;
LAB_001018cb:
      if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return uVar1;
```

5. When reversed properly they should get a chunk that contains **davy**

Section 4: Understanding Ascii/Hex

1. This section is the easiest one to uncover. When looking at the function the challenger will see that an array is being checked against another array with set values.
2. If you convert the hex values of the array it gives you the string **mack**.
   - The Ghidra or Ida decompiler might switch the hex values into ascii.
3. The challenger can that the set hex values are being checked against the string being passed in.



Decompile: FUN_00101716 - (mackstreasure_unpacked.exe)

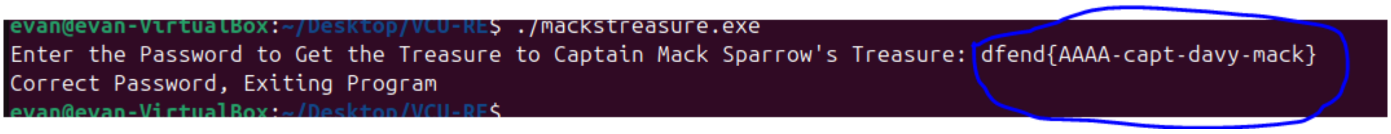
```

undefined8 FUN_00101716(long param_1)

{
  undefined8 uVar1;
  long in_FS_OFFSET;
  int local_2c;
  uint local_28 [6];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 40);
  local_28[0] = L'm';
  local_28[1] = L'a';
  local_28[2] = L'c';
  local_28[3] = L'k';
  local_2c = 0;
  do {
    if (3 < local_2c) {
      uVar1 = 1;
LAB_00101790:
      if (*(long *)(in_FS_OFFSET + 40) != *(long *)(in_FS_OFFSET + 40)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return uVar1;
    }
    if ((uint)*(byte *)(param_1 + local_2c) != local_28[local_2c]) {
      uVar1 = 0;
      goto LAB_00101790;
    }
    local_2c = local_2c + 1;
  } while( true );
}

```

4. The challenger will be able to figure out that this chunk contains **mack**

Final Flag

When the user types in the flag they will be met with a correct message. They could just enter the flag into the website and be fine as long as its correct. The correct message is just to state that the flag is the password.

dfend{AAAA-capt-davy-mack}



```
evan@evan-VirtualBox:~/Desktop/VCU-RE$ ./mackstreasure.exe
Enter the Password to Get the Treasure to Captain Mack Sparrow's Treasure: dfend{AAAA-capt-davy-mack}
Correct Password, Exiting Program
evan@evan-VirtualBox:~/Desktop/VCU-RE$
```

Red Herrings

- I statically added a few windows api calls and imports to try and confuse the challenger.
- I added a bunch of functions and function calls to make the decompiled code a bit more complicated.

Code Appendix

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

char* getString() {
    char* str = (char*)malloc(9 * sizeof(char));
    strcpy(str, "pa55word");
    return str;
}

void getEncryptedFalseString(char* str, int key) {
    while (*str) {
        *str = *str ^ key;
        str++;
    }
}

void freeNonsense(char* nonsense) {
    free(nonsense);

}

void exitProgramDenied(char* result, char* user_input) {
    free(result);
    free(user_input);
    printf("Wrong Password, Exiting Program\n");
    exit(0);
}



void exitProgramAccepted (char* result, char* user_input) {
    free(result);
    free(user_input);
    printf("Correct Password, Exiting Program\n");
    exit(0);
}
```

```
bool checkDashes(const char* serialnumber, int index) {



    if (index >= strlen(serialnumber) - 1) {
        return true;
    }

    if ((index + 1) % 5 == 0) {
        if (serialnumber[index] != '-') {
            return false;
        }
    } else {
        if (serialnumber[index] == '-') {
            return false;
        }
    }

    return checkDashes(serialnumber, index + 1);
}

bool checkChunk(const char* chunk) {
    char firstChar = chunk[0];
    if (firstChar != 'A'){
        return false;
    }
    for (int i = 1; i < 3; i++) {
        if (chunk[i] != firstChar) {
            return false;


        }
    }
    return true;
}

bool checkDfend(const char* chunk) {
    char xor_key[6] = "&$',&9";
    char character_being_checked;
    for (int i = 0; i < 6; i++) {
        character_being_checked = chunk[i] ^ 0x42;
        if (character_being_checked != xor_key[i]) {
            return false;
        }
    }
    return true;
}


bool checkEnd(char last_char) {
    char xor_key = '?';
    char character_being_checked;
    character_being_checked = last_char ^ 0x42;
    if (character_being_checked != xor_key)
    {
        return false;
    }

    return true;
}

bool seeded_encryption(char* user_input) {
    size_t input_length = strlen(user_input);
    int hex_values[] = {0x32, 0x5F, 0x6B, 0x23};
    char* encrypted_string = malloc(input_length + 1);

    char* red_herring = "Win32.dll";
    char* red_herring2 = "Kernel32.dll";
    char* red_herring3 = "ws2_32.dll";




    if (!encrypted_string) {
        return false;
    }
    srand(1337);
    for (size_t i = 0; i < input_length; ++i) {
        encrypted_string[i] = user_input[i] ^ (rand() % 100);

    }
    encrypted_string[input_length] = '\0';

    for (size_t i = 0; i < input_length; ++i) {
        if ((unsigned char)encrypted_string[i] != hex_values[i %
sizeof(hex_values)]) {
            free(encrypted_string);
            return false;
        }
    }

    return true;
}

bool checkMack(char *last_chunk)

{
int hex_values[] = {0x6D, 0x61, 0x63, 0x6B};

 for (int i = 0; i < 4; i++) {
        if ((unsigned char)last_chunk[i] != hex_values[i]) {
            return false;
        }
    }

    return true;
}

bool checkThirdChunk(char* chunk3, char* chunk2, char* chunk4, char* chunk1)
{
    char new_chars[4];

    for (int i = 0; i < 4; i++) {
        new_chars[i] = chunk3[i] ^ chunk2[i];
    }

    for (int i = 0; i < 4; i++) {
        new_chars[i] = new_chars[i] + chunk4[i];
    }

    for (int i = 0; i < 4; i++) {
        new_chars[i] = new_chars[i] - chunk1[i];
    }


    char solution_chars[4] = { 51 , 32, 40, 55};
    for (int i = 0; i < 4; i++) {
        if (new_chars[i] != solution_chars[i]) {
            return false;
        }
    }
    return true;
}




bool checkSequential(const char* str) {

    char chunk1[5], chunk2[5], chunk3[5], chunk4[5] , chunk5[6];
    char end_character = str[25];

    strncpy(chunk5, str, 6);
    strncpy(chunk1, str + 6, 4);
    strncpy(chunk2, str+ 11, 4);
    strncpy(chunk3, str + 16, 4);
    strncpy(chunk4, str + 21, 4);


    bool check_chunk1 , check_chunk2, check_chunk3 , check_chunk4,
check_chunk5, check_end;
    check_chunk5 = checkDfend(chunk5) ;
    check_chunk1 = checkChunk(chunk1);
    check_chunk2 = seeded_encryption(chunk2);
    check_chunk4 = checkMack(chunk4);
    check_end = checkEnd(end_character);
    check_chunk3 = checkThirdChunk(chunk3, chunk2, chunk4, chunk1);


    if(check_chunk1 && check_chunk2 && check_chunk3 && check_chunk5 &&
check_end && check_chunk4 )
        {
        return true;
        }
    else {



        return false;
    }
```

```
}



bool passwordGeneration(char* user_input ) {

    char* encrypted_nonsense = (char*)malloc(26 * sizeof(char));
    char serialnumber[20];
    strncpy(serialnumber, user_input + 6, 19);
    serialnumber[19] = '\0';

 if (strlen(user_input) != 26) {
        freeNonsense(encrypted_nonsense);
        return 1;
    }

    if (!checkDashes(serialnumber ,0)) {
        return 1;
    }

    if (!checkSequential(user_input)) {
        return 1;

    }
}

int main() {
    char* user_input = (char*)malloc(100 * sizeof(char));

    int key = 0x42;
    char* result = getString();
    char* fake_string = strdup(result);
    getEncryptedFalseString(fake_string, key);

    printf("Enter the Password to Get the Treasure to Captain Mack Sparrow's
Treasure: ");
    scanf("%99s", user_input);
    int obvious_compare = strcmp(user_input, result);
    int encrypted_compare = strcmp(user_input, fake_string);

    bool password_check = passwordGeneration(user_input);



    if (obvious_compare == 0) {
        printf("Not the password you are looking for... Dig a little bit
deeper into the program, not the empty lot... \n");
        exitProgramDenied(result, user_input);
    }
    else if (encrypted_compare == 0) {
        printf("Use the method you got to print this message to find the
next clue. The treasure is getting warmer...\n");
        exitProgramDenied(result, user_input);

    }
    else if (!password_check) {

        exitProgramAccepted(result, user_input);
    }
    else {
        exitProgramDenied(result, user_input);
    }

    return 0;
}
```