babyreee

Challenge Description: You've never seen a flagchecker this helpful.
Challenge Author: Fawl
Genre: Reversing
Provided: "chall" Linux executable program

In this challenge, we want to retrieve the flag from the executable program provided, most likely by decompiling it. We first try running the file.
(If you are using a Windows-based system, consider setting up WSL (Windows Subsystem for Linux). Windows cmd.exe will not run the file, more to be explained later.)

Upon executing the file, we receive the following prompt:

```
ING/rev_babyreeee/distrib$ ./chall
Hello! Welcome to SEETF. Please enter the flag.
```

No other information is provided. Proceed to decompile the application. For this challenge, IDA software has been recommended for beginners. Open the program with IDA.

```
; +-------------------------------------------------------------------------+
; |      This file was generated by The Interactive Disassembler (IDA)      |
; |           Copyright (c) 2022 Hex-Rays, <support@hex-rays.com>           |
; |                            Freeware version                             |
; +-------------------------------------------------------------------------+
;
; Input SHA256 : E246B954A10C56A974381CE4A285F744E02D5854286ADB47EA1F7C42FF9A3D7A
; Input MD5    : 7F3C4533F3E5EF53317597ECA724E02D
; Input CRC32  : 3D9575EB

; File Name   : C:\Users\lucas\OneDrive - Nanyang Technological University\NTU\Comps\SEETF\REVERSING\rev_babyreeee\distrib\chall
; Format      : ELF64 for x86-64 (Shared object)
; Interpreter '/lib64/ld-linux-x86-64.so.2'
; Needed Library 'libc.so.6'
;
```

(Clearly, the program uses a Linux-based interpreter. Hence, cmd.exe cannot run the file.)

Generate a pseudo-code of the program to understand it better. (From the menu, View > Open subviews > Generate pseudocode)
A short program is written for the application in the `main` function.

```c
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  size_t v4; // rax
  __int64 v5; // rdx
  size_t v6; // rax
  int v7; // esi
  char v8; // cl
  unsigned int v9; // r8d
  char v10[128]; // [rsp+0h] [rbp-158h] BYREF
  __int128 v11[13]; // [rsp+80h] [rbp-D8h]

  puts("Hello! Welcome to SEETF. Please enter the flag.");
  v11[0] = (__int128)_mm_load_si128((const __m128i *)&xmmword_20F0);
  v11[1] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2100);
  v11[2] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2110);
  v11[3] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2120);
  v11[4] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2130);
  v11[5] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2140);
  v11[6] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2150);
  v11[7] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2160);
  v11[8] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2170);
  v11[9] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2180);
  v11[10] = (__int128)_mm_load_si128((const __m128i *)&xmmword_2190);
  v11[11] = (__int128)_mm_load_si128((const __m128i *)&xmmword_21A0);
  v11[12] = (__int128)_mm_load_si128((const __m128i *)&xmmword_21B0);
  fgets(v10, 128, stdin);
  if ( strlen(v10) == 53 )
  {
    puts("Good work! Your flag is the correct size.");
    puts("On to the flag check itself...");
    v4 = strlen(v10);
    v5 = 0LL;
    v6 = v4 - 1;
    do
    {
      v9 = v5;
      if ( v6 == v5 )
      {
        puts("Success! Go get your points, champ.");
        return 0LL;
      }
      v7 = *((_DWORD *)v11 + v5);
      v8 = v5 ^ (v10[v5] + 69);
      ++v5;
    }
    while ( (_BYTE)v7 == v8 );
    printf("Flag check failed at index: %d", v9);
    return 1LL;
  }
  else
  {
    printf("Flag wrong. Try again.");
```

```
        return 1LL;
    }
}
```

Do not be intimidated by the code; it can be broken up into small segments.

(1) Code before `puts("Hello...`

These are all declarations of variables, there is nothing much to care about yet.
`puts()` prints string into the output stream.

(2) `v11[i]` for all i, until `fgets(v10, 128, stdin);`

- We can interpret `v11` to be some data array. Note the keyword `const` and characters `*` and `&`.
- `const` denotes that the data array is a constant variable => this array will not be changed during the program execution. **It is highly probable that `v11` contains the flag**.
- `(const *)` type casts each element of `v11` to be a constant pointer variable. A pointer variable "points" to some address in memory storing the data. **The flag could be stored in this memory location**.
- `&xmmword_k` for some *k*. `&` is the address operator. In this code, information in the variable `xmmword_k` is referenced to by its address, through the use of `&` and the pointer array `v11`.
- `fgets()` stores the user's flag input into the variable `v10`.

(3) The `if()` condition

`strlen(v10)==53` checks if the user's input has 53 characters. The code checks if the user's input is of the correct length. However, we are not certain how the compiler or interpreter runs, and thus the length may include some of the following characters:

- `'\0'`, the null character, which denotes the end of a string in some languages such as C
- `'\n'`, the newline character, which is entered by the user at the end of the input string

For now, we take the flag to be *roughy* 53 characters long.

(4) `v5`, `v6` and the `do-while` loop

This is the most crucial segment of the code.

- `++v5`: `v5` is incremented with each loop => it should be the counter variable.
- `v9 = v5`: `v9` counts along with `v5`, but is updated only in the next loop (`v9` lags `v5` by one loop). `v9` is probably used when the loop breaks.
- `v4 = strlen(v10); v6 = v4 - 1`: `v6` is 1 less than the flag length. Since indexing usually starts from 0, `v6` likely denotes the last index count of the string.
- `if (v6 == v5)`: Based on the above interpretations, this would mean that `v5` has incremented to the last index => the `while` condition has been met for every index of the string. We will look into the `while` condition shortly. Since the `if` condition has been met, the entire flag has been processed and the exit program can be run, as presented in the code.
- `v7 = *((_DWORD *)v11 + v5)`: `v11` points to the head of the data array and `v5` is a counter => `v11 + v5` is the address of the `v5` th character in the array. `(_DWORD *)` is a type cast; each character at the address is stored in the size of a DOUBLE WORD. The very first `*` is a dereference operator, retrieving the information stored at a given address. This information is stored in `v7`.
- `v8 = v5 ^ (v10[v5] + 69)`: Take the `v5` th element of the user input, `v10`, and add 69 to it. XOR this value against its index in `v10`. Store the value into `v8`. Let's call this algorithm "*Algo v8*", invoked upon some character of the user input.
- `while ( (_BYTE)v7 == v8)`: We can now confirm that both `v7` and `v8` are characters, probably represented by their ASCII values (`v8` must be a character since it is computed from user input). => Each

character of the flag is stored in `v11` after invoking *Algo v8*, in the same indexing order. Each character is represented within 1 byte in ASCII.

We can describe the `while` loop with the above information.
The loop parses through each character of the user input. If the character is equal to the stored data after invoking *Algo v8*, then the character is correct and the loop continues. Otherwise, the loop is broken, and the index of the error will be printed out (index stored in `v9`). If the loop successfully parses through to the last character, then the entire user input is correct and the flag has been found.

Having understood the code, proceed to `v11 => xmmword_k` to retrieve the stored data. This data is the flag after invoking *Algo v8*. Double-click on `xmmword_k` in the pseudocode to be directed to the variables in "IDA View".

```
xmmword_20F0    xmmword 0C3000000880000008B00000098h
                                        ; DATA XREF: main+14↑r
xmmword_2100    xmmword 0A30000007E000000B600000071h
                                        ; DATA XREF: main+36↑r
xmmword_2110    xmmword 7D00000073000000BB00000072h
                                        ; DATA XREF: main+46↑r
xmmword_2120    xmmword 7300000074000000A90000007Ah
                                        ; DATA XREF: main+56↑r
xmmword_2130    xmmword 6E000000B6000000A400000068h
                                        ; DATA XREF: main+66↑r
xmmword_2140    xmmword 6100000061000000BC00000062h
                                        ; DATA XREF: main+76↑r
xmmword_2150    xmmword 0BC00000067000000B300000062h
                                        ; DATA XREF: main+86↑r
xmmword_2160    xmmword 0B5000000B80000006B00000061h
                                        ; DATA XREF: main+96↑r
xmmword_2170    xmmword 55000000890000005400000056h
                                        ; DATA XREF: main+A6↑r
xmmword_2180    xmmword 510000005B000000500000008Ch
                                        ; DATA XREF: main+B6↑r
xmmword_2190    xmmword 5E0000005D0000005400000053h
                                        ; DATA XREF: main+C6↑r
xmmword_21A0    xmmword 89000000890000008600000050h
                                        ; DATA XREF: main+D6↑r
xmmword_21B0    xmmword 0F1000000490000004F00000048h
                                        ; DATA XREF: main+E6↑r
```

Notice there are only 13 variables (`v11` only has 13 elements). Also note that the data is stored in hexadecimal (hex). Recall, each ASCII character can be represented in 1 byte, 2 hex digits. Observe that each variable has 4 pairs of isolated, non-zero hex digits. These are the data-storing bits. We also know from earlier that each character is stored in a double-word. We can thus infer that each variable has 4 ASCII characters => there are a total of 52 characters.

For e.g., the first variable `xmmword_20F0` has 4 characters: `00000098h` , `0000008Bh` , `00000088h` , `0C3h` , where the last character has been truncated off its leading (and unnecessary) zeroes.

Let's try running the program and giving an input of length 52.

```
Hello! Welcome to SEETF. Please enter the flag.
1234567890123456789012345678901234567890123456789012
Good work! Your flag is the correct size.
On to the flag check itself...
Flag check failed at index: 0 [redacted]
```

Awesome, the correct length has been confirmed. Since we know the flag has a format of `SEE{ -+}` , we can attempt to verify the first 4 characters.

```
Hello! Welcome to SEETF. Please enter the flag.
SEE{5678901234567890123456789012345678901234567890123
Good work! Your flag is the correct size.
On to the flag check itself...
Flag check failed at index: 4 [redacted]
```

Indeed, the first 4 characters are `SEE{` . Apply *Algo v8* on each character.

- *Index 0*, `S = 0x98`
- *Index 1*, `E = 0x8B`
- *Index 2*, `E = 0x88`
- *Index 3*, `{ = 0xC3`

This matches our first variable `xmmword_20F0` , using the **Little Endian** system for storing data.

We now need to reverse *Algo v8* on the stored data. Transferring the data to Python allows for easy manipulation. To reverse *Algo v8*, we need to apply the following functions on each stored character:

1. XOR against its index (XOR is an involutory function; it is self-inverse)
2. Subtract 69
3. Print each ASCII numeric as its character.

In [1]:

```python
# data stored as given
data= [[0xc3, 0x88, 0x8b, 0x98], [0xa3, 0x7e, 0xb6, 0x71], [0x7d, 0x73, 0xbb, 0x72], [0x73,
# correct little endian and flatten the array
flag_char= []
for x in data:
    x.reverse()
    flag_char+= x

# apply reverse algo
for i in range(len(flag_char)): flag_char[i]= (flag_char[i] ^ i) - 69
flag= ''.join([chr(x) for x in flag_char])

print(flag)
```

```
SEE{0n3_5m411_573p_81d215e8b81ae10f1c08168207fba396}
```

Run the computed flag into the application. The flag has been found. :D

```
Hello! Welcome to SEETF. Please enter the flag.
SEE{0n3_5m411_573p_81d215e8b81ae10f1c08168207fba396}
Good work! Your flag is the correct size.
On to the flag check itself...
Success! Go get your points, champ.
```