LACTF-2024 - Reversing shattered-memories

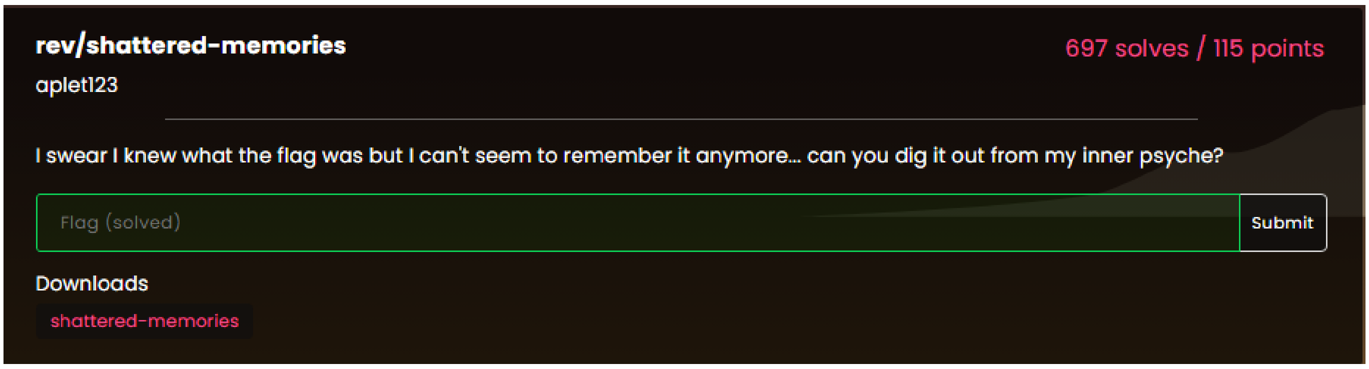
github.com/anre18

23.02.2024

Contents

1 Challenge-Introduction

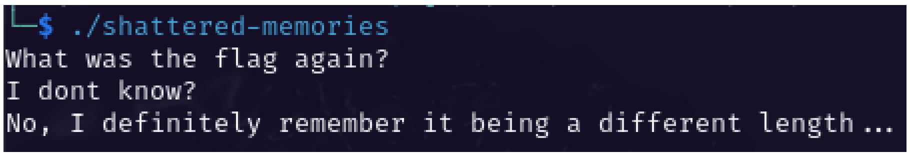
The following Write up is intended as an short overview for this years LACTF-Reversing-Challenge **shattered-memories**.
The Event contained a bunch of CTF-Challenges from all typical Categories from Rev to misc. For more information about the event, look here on the Official Website.
I myself attended the Event online as a member of our University's IT-Security team, and primarily focused on the Reversing and Pwn Challenges.
All the Challenges can be found on the Official LACTF Github Repostitory.


rev/shattered-memories
aplet123
697 solves / 115 points
I swear I knew what the flag was but I can't seem to remember it anymore... can you dig it out from my inner psyche?

The Challenge itself starts with a short introductory Text: *I swear I knew what the flag was but I can't seem to remember it anymore... can you dig it out from my inner psyche?*
This introduction will prove to be quite fitting, as the flag (as we will see later), is truly "**shattered**" inside the **Programs** memory.
After this, the only remaining thing we get, is the Challenge Executable, called **shattered-memories**.

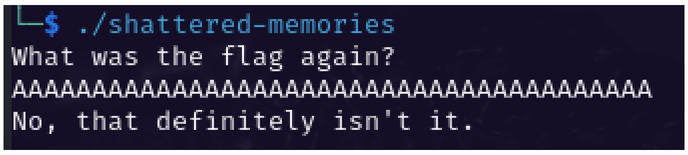
1.1 Running the Binary

To get an early idea of the challenge we're facing, we will simply run the binary, and check out what it does. We're faced with a text prompt asking us for the flag. After entering some text, we're informed that the, by us, provided flag, is of **incorrect length.**



```
└─$ ./shattered-memories
What was the flag again?
I dont know?
No, I definitely remember it being a different length...
```

When trying out different input lengths (manually OR automatically), we'll find out that the expected length is **40 Characters** (or at least that length leads to a different response.)



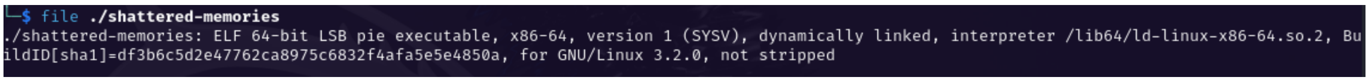
```
└─$ ./shattered-memories
What was the flag again?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
No, that definitely isn't it.
```

Now at least we can suppose, that the Program wants some input with **a length 40**. (*I also tried really long inputs for buffer overflows ,and others to check for format String exploits, but with no luck*)

2 Reversing

2.1 Initial-Checks

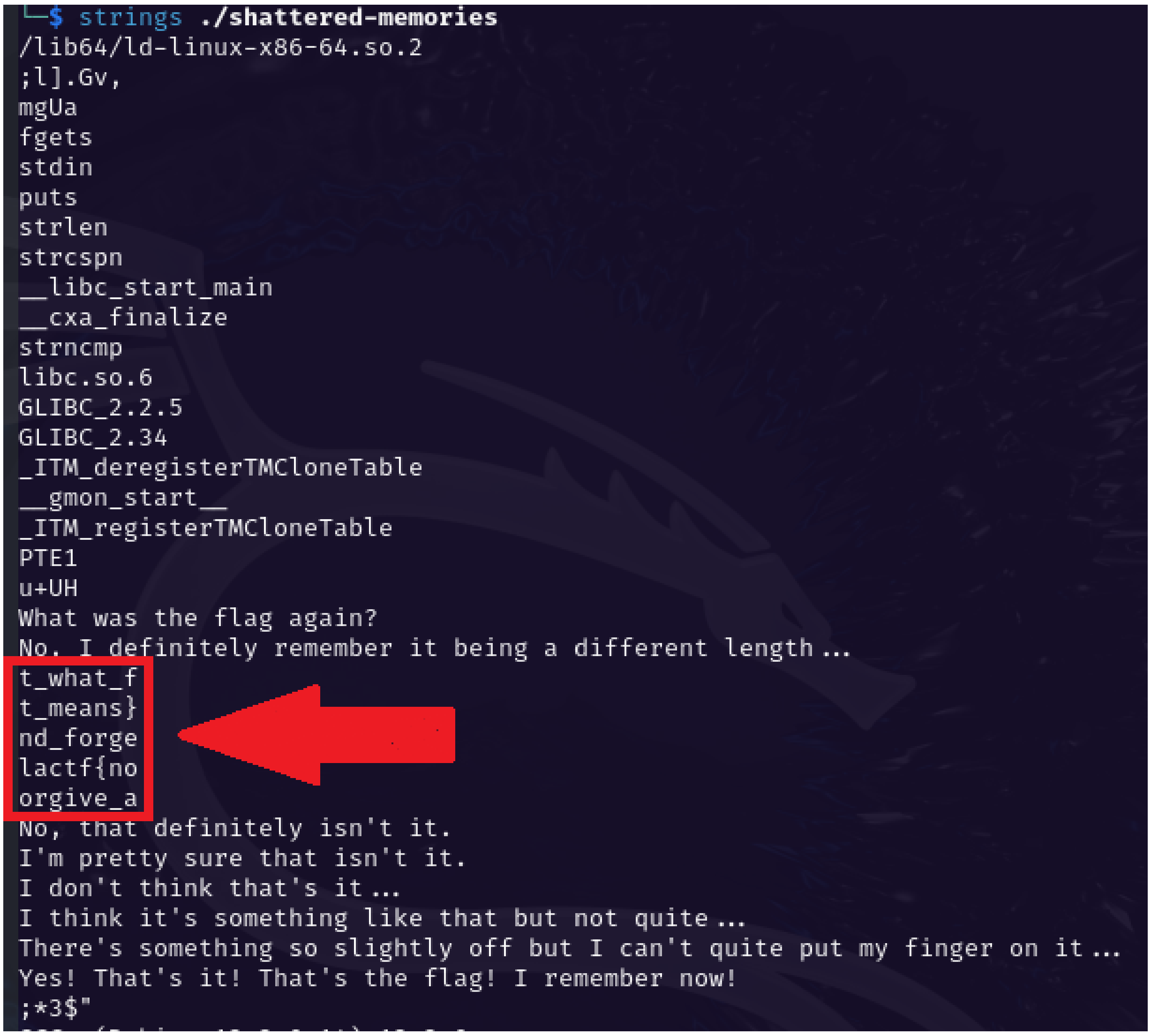
After playing around with different inputs (as mentioned before), I went on to the Typical checks:



```
└─$ file ./shattered-memories
./shattered-memories: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, Bu
ildID[sha1]=df3b6c5d2e47762ca8975c6832f4afa5e5e4850a, for GNU/Linux 3.2.0, not stripped
```

A **file** check revealed nothing special or interesting. We're dealing with a typical 64-bit, dynamically linked, unstripped ELF executable. The only truly relevant information we gathered from this is that the binary is **unstripped**, making **'decompilation'** relatively straightforward.

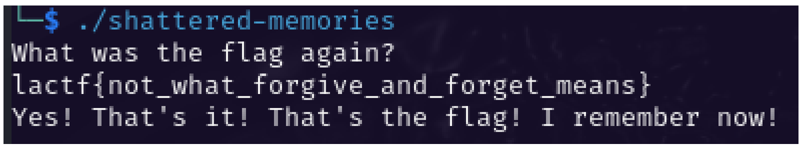
A check with the **strings** utility, to extract all the printable character sequences found inside the binary (*that are atleast 4 Characters long*), proved quite valuable.



```
└─$ strings ./shattered-memories
/lib64/ld-linux-x86-64.so.2
;l].Gv,
mgUa
fgets
stdin
puts
strlen
strcspn
__libc_start_main
__cxa_finalize
strncmp
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
What was the flag again?
No. I definitely remember it being a different length...
t_what_f
t_means}
nd_forge
lactf{no
orgive_a
No, that definitely isn't it.
I'm pretty sure that isn't it.
I don't think that's it...
I think it's something like that but not quite...
There's something so slightly off but I can't quite put my finger on it...
Yes! That's it! That's the flag! I remember now!
;*3$"
```

As we can see we already extracted **five** consecutive Strings, which already look like **part** of a **possible Flag**. When summed up they even **Total to 40 Characters**, just like we need for the **Flag**.
If we now look more closely at the 5 parts and **rearrange** them, we can relatively easily arrive at a reasonably looking and formatted flag: ***lactf{not_what_forgive_and_forget_means}***

If we **try** this flag candidate as input, we're greeted with a **positive** response, informing us that our found flag, is indeed the **correct** one. This is also **confirmed** by providing the flag as a challenge response on the official CTF platform.



```
└─$ ./shattered-memories
What was the flag again?
lactf{not_what_forgive_and_forget_means}
Yes! That's it! That's the flag! I remember now!
```

That's all great and would technically be *good enough* for the challenge. But I somehow felt a little *underwhelmed* by this conclusion, and wanted to **go a bit further** and **analyze** the code, to really **fully understand** what's going on here.
Therefore we're going to continue now with the **static Code analysis**.

2.2 Static Code Analysis

To analyze the program, I put it in Ghidra, which I then used for the analysis.
Ghidra found **two** especially **relevant functions**, that we're going to analyze more in-depth now. The following code snippets are from the decompiled ***main*** and ***strip_newline*** functions. The **variables** and **types** have already been **renamed** by me to help with understanding what's going on. I'm going to split the snippets in shorter more meaningful blocks, to be more easily understandable.

```
1  int main(void)
2  {
3    int flagCompare;
4    size_t userinputLength;
5    char userinput [8];
6    char flag1 [8];
7    char flag4 [8];
8    char flag3 [8];
9    char flag2 [108];
10   int compareIncrement;
11
12   puts("What was the flag again?");
13   fgets(userinput,0x80,stdin);
14   strip_newline(userinput);
15   userinputLength = strlen(userinput);
16   if (userinputLength == 0x28) {
```

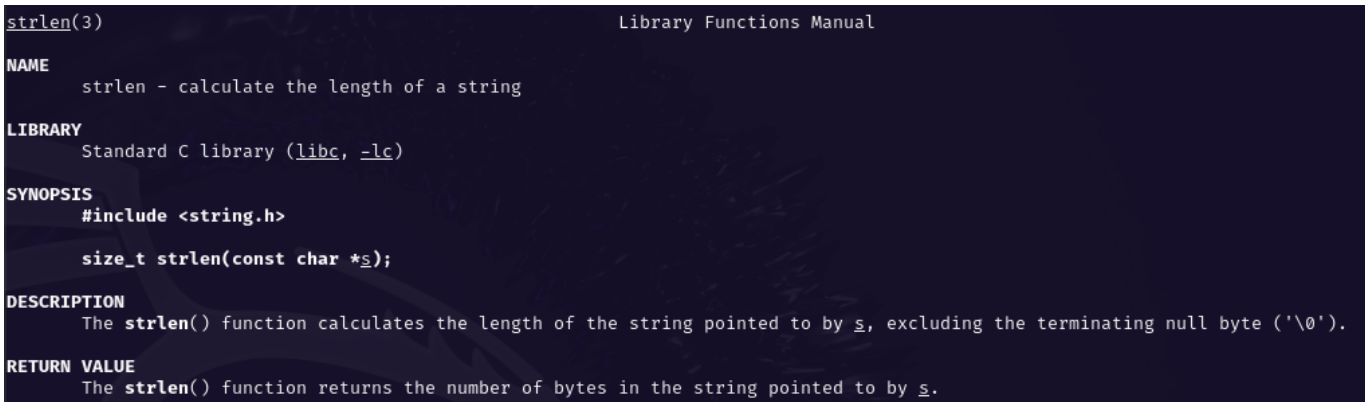
We're going to start with the **main** function.
This one starts with a bunch of **variable definitions** (*lines 3-10*), followed by the **Text prompt**, asking us, for the Flag *(line 12)*.
As it turns out, our **assumption** was indeed correct, and the program really **checks** our input for a **length of 0x28(dec40) characters** *(lines 15-16)* , after it has read the same, for **up to 0x80 (dec128)** characters, from stdin, and stored it in one of the previously defined variables *(lines 13-14)*.
Especially interesting, is that our userinput is given to the ***strip_newline*** function, which then returns it **modified**.

To really understand what's done with our input, we're now going to **take a look** at this function:

```
1  void strip_newline(char *data_string)
2  {
3    size_t newLine_Location;
4
5    newLine_Location = strcspn(data_string,"\n");
6    data_string[newLine_Location] = '\0';
7    return;
8  }
```

As we can see, this is really **short and compact**, and doesn't do a lot. It simply gets a **reference** to our userinput String (*line 1*), checks for the first **new line** character (*line 5*) (should normally be at the end of our input), and then **replaces** this one with an **null** Terminator/Character (*line 6*).

This was quite interesting for me at the beginning, as I did not knew that the c-lib **strlen** function,(used to check the length of our user input), is using the **nullbyte** as **end marker** for the **length** calculation.
But this i easily verifiable, my simply checking the manual page for strlen:



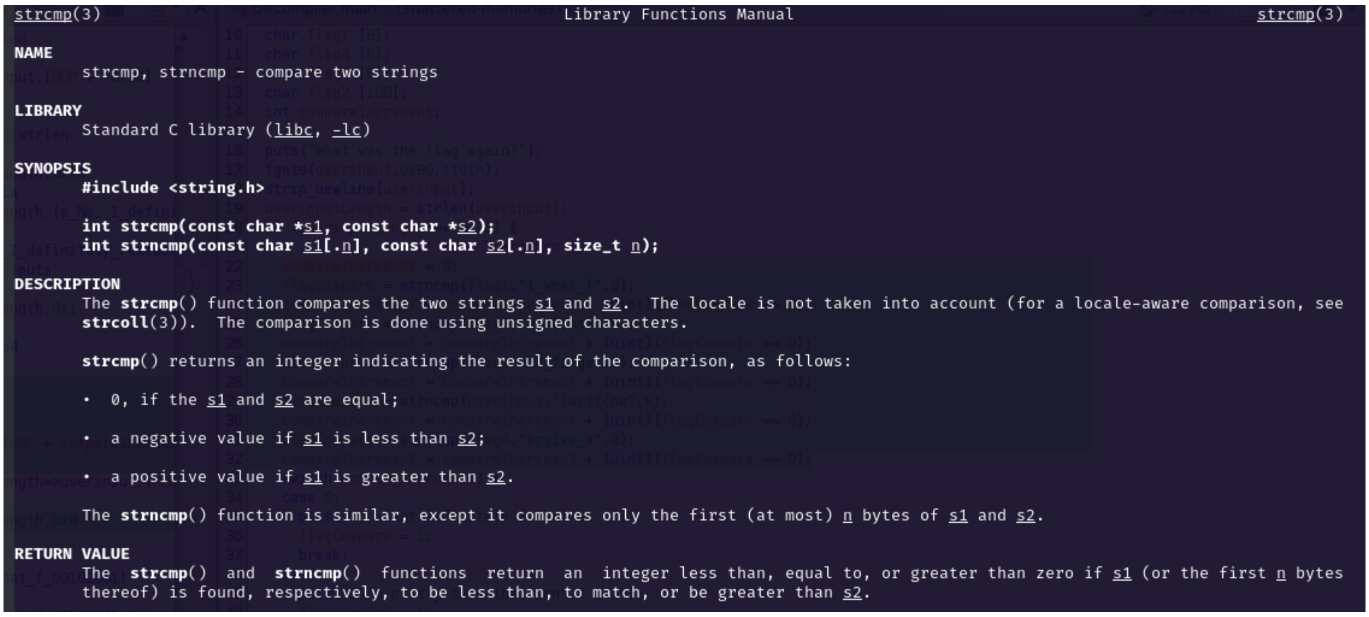
```
strlen(3)                                     Library Functions Manual

NAME
       strlen - calculate the length of a string

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <string.h>

       size_t strlen(const char *s);

DESCRIPTION
       The strlen() function calculates the length of the string pointed to by s, excluding the terminating null byte ('\0').

RETURN VALUE
       The strlen() function returns the number of bytes in the string pointed to by s.
```

Knowing this, we can continue with our analysis of the **main function**.

```
  if (userinputLength == 0x28) {
    compareIncrement = 0;
    flagCompare = strncmp(flag1,"t_what_f",8);
    compareIncrement = compareIncrement + (uint)(flagCompare == 0);
    flagCompare = strncmp(flag2,"t_means}",8);
    compareIncrement = compareIncrement + (uint)(flagCompare == 0);
    flagCompare = strncmp(flag3,"nd_forge",8);
    compareIncrement = compareIncrement + (uint)(flagCompare == 0);
    flagCompare = strncmp(userinput,"lactf{no",8);
    compareIncrement = compareIncrement + (uint)(flagCompare == 0);
    flagCompare = strncmp(flag4,"orgive_a",8);
    compareIncrement = compareIncrement + (uint)(flagCompare == 0);
    switch(compareIncrement) {
    case 0:
      puts("No, that definitely isn\'t it.");
      flagCompare = 1;
      break;
    case 1:
      puts("I\'m pretty sure that isn\'t it.");
      flagCompare = 1;
      break;
    case 2:
      puts("I don\'t think that\'s it...");
      flagCompare = 1;
      break;
    case 3:
      puts("I think it\'s something like that but not quite...");
      flagCompare = 1;
      break;
    case 4:
      puts("There\'s something so slightly off but I can\'t quite put my finger on it
        ...");
      flagCompare = 1;
      break;
    case 5:
      puts("Yes! That\'s it! That\'s the flag! I remember now!");
      flagCompare = 0;
```

```
52        break;
53      default:
54        flagCompare = 0;
55      }
56    }
57    else {
58      puts("No, I definitely remember it being a different length...");
59      flagCompare = 1;
60    }
61    return flagCompare;
62  }
```

Currently we know, that the program wants us to **enter 40 Characters**, and that its **replacing** the fist **new Line** with a **null byte**.
Continuing within the length check (*aka. when our input had the correct length*), we see a bunch of **sub-string Comparisons** *(lines 18-27)*). Every one of those checks, checks if another part of the Flag is within our input. If a part of the Flag is found within our user input, the program increments a counter*(lines 19,21,23,25,27*), which will be important later.
To understand how this is done, the following explanation:
The counter starts at zero (*line 17*). Now follows the string comparison via the c-lib strcmp function. If we look at the manual page for strcmp we can see, that this function takes up to three inputs (*that's the case used here*), in the form of two character sequences (our input and the flag part its compared against), and one integer for size. The size integer is always 8 for our checks, as the flag parts are also 8 characters long. The Return of this function is an integer, which is zero if both strings are equal, and positive or negative, if one or the other is greater/smaller.
The **result** of this (aka the return value) is saved inside another **variable**, which is then used to **increment** our **counter** variable, if the **result** of the previous comparison was **zero** aka the flag string was **contained** in our input.
This check is the than repeated for all flag parts.
At the end we have a **counter** containing the **number of flag parts**, contained **within our user input**.



```
strcmp(3)                         Library Functions Manual                         strcmp(3)

NAME
       strcmp, strncmp - compare two strings

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <string.h>

       int strcmp(const char *s1, const char *s2);
       int strncmp(const char s1[.n], const char s2[.n], size_t n);

DESCRIPTION
       The strcmp() function compares the two strings s1 and s2.  The locale is not taken into account (for a locale-aware comparison, see
       strcoll(3)).  The comparison is done using unsigned characters.

       strcmp() returns an integer indicating the result of the comparison, as follows:

       •  0, if the s1 and s2 are equal;

       •  a negative value if s1 is less than s2;

       •  a positive value if s1 is greater than s2.

       The strncmp() function is similar, except it compares only the first (at most) n bytes of s1 and s2.

RETURN VALUE
       The  strcmp()  and  strncmp()  functions  return  an  integer less than, equal to, or greater than zero if s1 (or the first n bytes
       thereof) is found, respectively, to be less than, to match, or be greater than s2.
```

Thats great, as we now **for certain** know, where the **strings** utility got the **flag snippets** from.
It simply found the **strings** used here, to compare against our input.

This counter is than used within a switch-case (*lines 28-55*). The Program prints **Error Messages** for **all** Counter Values, **except** 5(*lines 29-38*) (aka when we have all five flag parts, within our input), in all of those wrong cases, our previously set **integer Variable**, previously used to **save** the **results of the string comparisons**, is set to 1 (its later used as the return value for main), except for when we get a different counter than 0-5. (which practically should be impossible without modifying the programs memory at run time, and therefore is irrelevant)
But if we have all the five Flag-Parts, we get the **congratulations** message, and the return variable is set to **zero**.

3 Conclusion/Final Statement

What we got here, was an really easy Reversing challenge, which should be trivial to solve even for Novices. To be honest, I was a little sad at the beginning, seeing how easy the Flag was obtainable, but even Challenges like this one have their place in an Competition like this one.
They are great for complete Beginners and therefore help them getting into Reverse Engineering, and can also be a small reward, when stuck on harder Challenges. Such small rewards can really boost your motivation to continue, especially when you're just stuck trying to solve a challenge, with no progress in hours. That's why I would still recommend and hope to see at least a few of such challenges *(maybe even in other categories)* in the next year's LACTF Competition. And I'm hoping that any true beginner who has solved this, as their first reversing challenge, feels motivated to explore and learn more about this great field. As Reverse engineering, can be a quite fun experience.