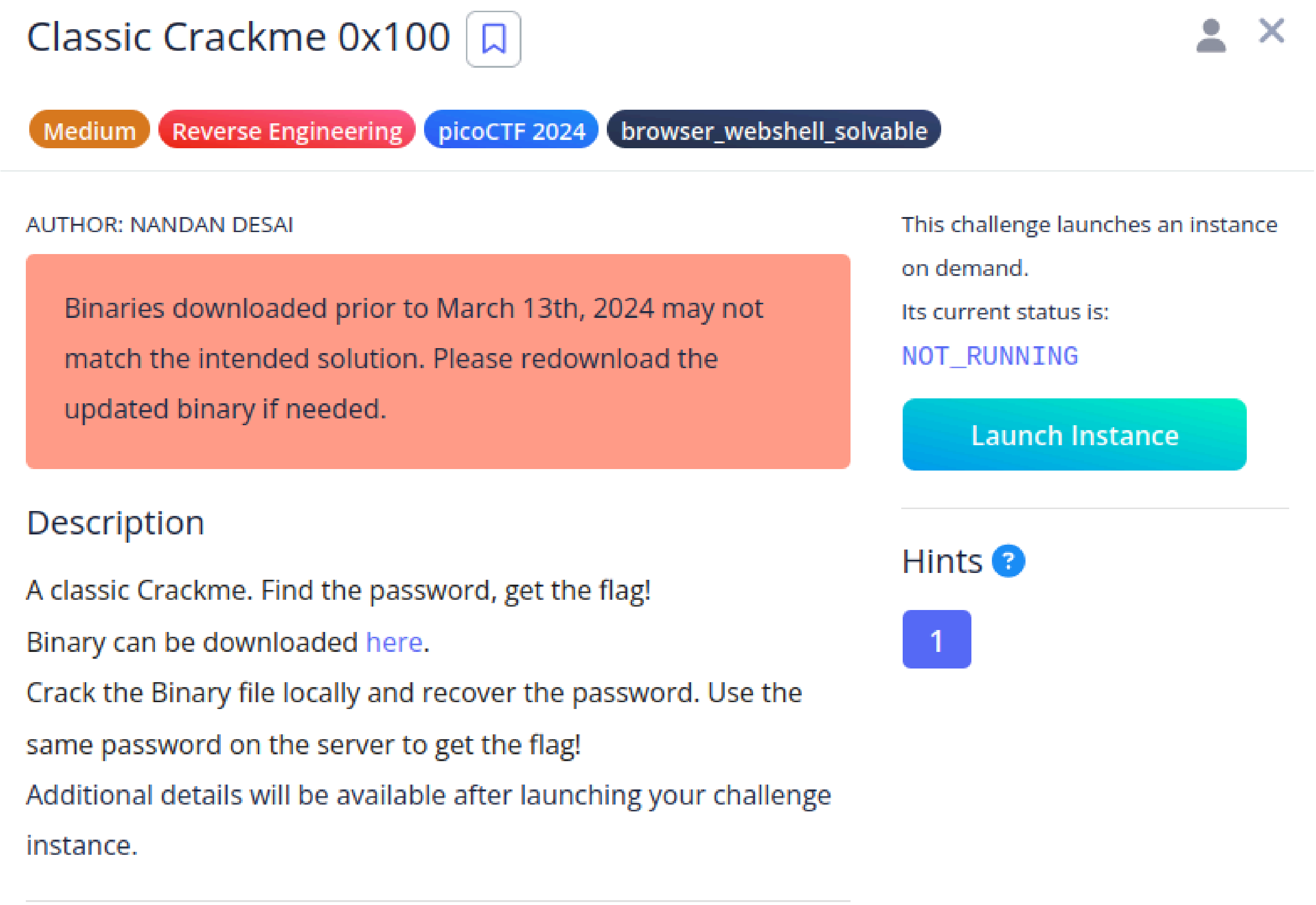
CTF-WRITEUPS

PicoCTF :

1. Classic Crackme 0x100



Classic Crackme 0x100

AUTHOR: NANDAN DESAI

Binaries downloaded prior to March 13th, 2024 may not match the intended solution. Please redownload the updated binary if needed.

This challenge launches an instance on demand.

Its current status is:

NOT_RUNNING

Description

A classic Crackme. Find the password, get the flag!

Binary can be downloaded here.

Crack the Binary file locally and recover the password. Use the same password on the server to get the flag!

Additional details will be available after launching your challenge instance.

I successfully downloaded the challenge binaries. Let's start analyzing them to extract the flag! 😊.

Now, let's run the challenge to observe its behavior and interact with the command line (if available). 😄

First, ensure the file is executable:

```
chmod +x ./crackme100
```

Since we trust picoCTF, we proceed to execute it:

```
./crackme100
```

Result:
When running the binary, the program prompts for input:

```
root@ossama:/home/ossama/Downloads# ./crackme100 Enter the secret password:`
```

It always comes to mind to brute force the input password, but this isn’t certain. Since the exercise seems simple, let’s try reversing it like master reverse engineers! 😎

with a random password i got :

```
root@ossama:/home/ossama/Downloads# ./crackme100 Enter the secret password:
gggggggggg
FAILED
```

then,
I used the `file` command to check the type of the binary and interpret the result. The output reveals the following information about `crackme100`:

```
root@ossama:/home/ossama/Downloads# file crackme100 crackme100: ELF 64-bit LSB
executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-
64.so.2, BuildID[sha1]=ece9fb88da586271de7c10da6c167388d3699e88, for GNU/Linux 3.2.0,
with debug_info, not stripped
```

This tells us that the file is an ELF (Executable and Linkable Format) for 64-bit Linux systems, dynamically linked, and includes debugging information, which can be useful for reverse engineering.

then,
I used the `ltrace` tool to analyze the program’s function calls during execution. `ltrace` is a command-line utility that traces library function calls made by a program, showing their arguments and return values. This helps us understand the program's behavior, especially when trying to identify potential vulnerabilities [not that much important in our case :( )].

In the output, we observe several important function calls:

```
root@ossama:/home/ossama/Downloads# ltrace ./crackme100 setvbuf(0x7d93b9c045c0,
0, 2, 0) = 0
printf("Enter the secret password: "Enter the secret password: ) = 27
__isoc99_scanf(0x402024, 0x7ffc5f29a4f0, 0, 0gggggggg
) = 1 strlen("ztqittwtxtieyfrslgtzuxovlfdnbrsn"...) = 50
memcmp(0x7ffc5f29a4f0, 0x7ffc5f29a530, 50, 0x7ffc5f29a530) = 0xffffffed
puts("FAILED!"FAILED!
) = 8 +++ exited (status 0) +++
```

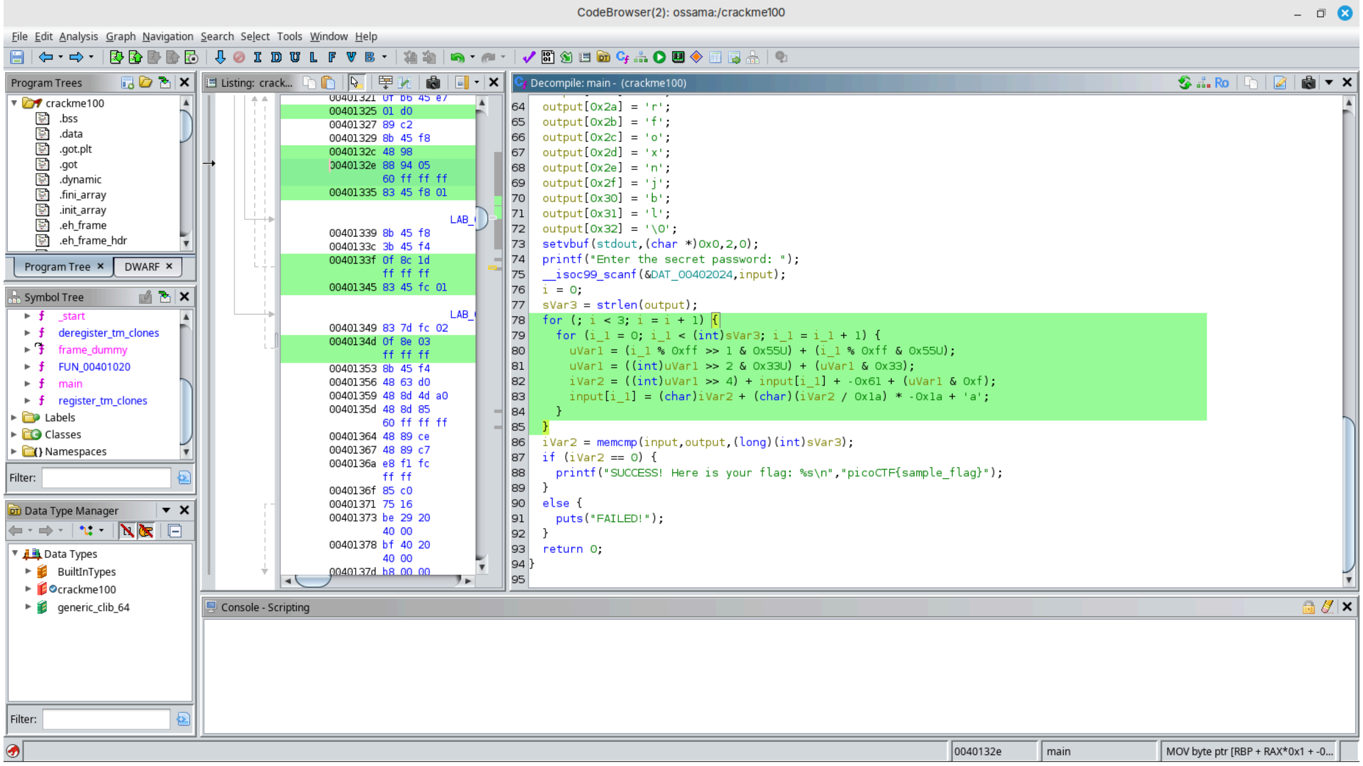
The key functions here are `__isoc99_scanf` (which handles user input) and `memcmp` (which compares the entered password with the expected value). The `scanf` function could be vulnerable to a buffer overflow if the input isn't properly validated, but based on the `strlen` and `memcmp` calls, it appears the input is being processed correctly for this context. There doesn't seem to be an immediate buffer overflow vulnerability in this trace.

The program compares the entered password with a predefined one, and if they don't match, it prints "FAILED!" and exits. To complete the challenge, we need to determine the correct password by further analyzing the binary, likely by reversing the program to locate the correct value or password check mechanism.

Now,

Ghidra is a powerful reverse engineering tool used for analyzing and decompiling binaries. Other tools with similar functionality include IDA Pro and Radare2. I prefer Ghidra because it's free, open-source, and has a user-friendly interface with strong support for various architectures and file formats.

Opening our file in Ghidra gives :



```
00401321 0f b6 45 e7
00401325 01 d0
00401327 89 c2
00401329 8b 45 f8
0040132c 48 98
0040132e 88 94 05
         60 ff ff ff
00401335 83 45 f8 01

                        LAB_
00401339 8b 45 f8
0040133c 3b 45 f4
0040133f 0f 8c 1d
         ff ff ff
00401345 83 45 fc 01

                        LAB_
00401349 83 7d fc 02
0040134d 0f 8e 03
         ff ff ff
00401353 8b 45 f4
00401356 48 63 d0
00401359 48 8d 4d a0
0040135d 48 8d 85
         60 ff ff ff
00401364 48 89 ce
00401367 48 89 c7
0040136a e8 f1 fc
         ff ff
0040136f 85 c0
00401371 75 16
00401373 be 29 20
         40 00
00401378 bf 40 20
         40 00
0040137d b8 00 00
```

```
  output[0x2a] = 'r';
  output[0x2b] = 'f';
  output[0x2c] = 'o';
  output[0x2d] = 'x';
  output[0x2e] = 'n';
  output[0x2f] = 'j';
  output[0x30] = 'b';
  output[0x31] = 'l';
  output[0x32] = '\0';
  setvbuf(stdout,(char *)0x0,2,0);
  printf("Enter the secret password: ");
  __isoc99_scanf(&DAT_00402024,input);
  i = 0;
  sVar3 = strlen(output);
  for (; i < 3; i = i + 1) {
    for (i_1 = 0; i_1 < (int)sVar3; i_1 = i_1 + 1) {
      uVar1 = (i_1 % 0xff >> 1 & 0x55U) + (i_1 % 0xff & 0x55U);
      uVar1 = ((int)uVar1 >> 2 & 0x33U) + (uVar1 & 0x33);
      iVar2 = ((int)uVar1 >> 4) + input[i_1] + -0x61 + (uVar1 & 0xf);
      input[i_1] = (char)iVar2 + (char)(iVar2 / 0x1a) * -0x1a + 'a';
    }
  }
  iVar2 = memcmp(input,output,(long)(int)sVar3);
  if (iVar2 == 0) {
    printf("SUCCESS! Here is your flag: %s\n","picoCTF{sample_flag}");
  }
  else {
    puts("FAILED!");
  }
  return 0;
}
```

Opening our file in Ghidra, we go straight to the main function. We see this code:

```
int main(void)
{
  uint uVar1;
  int iVar2;
  size_t sVar3;
  char input[51];
  char output[51];
  int random2;
  int random1;
  char fix;
  int secret3;
  int secret2;
  int secret1;
  int len;
  int i_1;
  int i;

  output[0] = 'z';
  output[1] = 't';
  output[2] = 'q';
  output[3] = 'i';
  output[4] = 't';
  output[5] = 't';
```

```
    output[6] = 'w';
    output[7] = 't';
    output[8] = 'x';
    output[9] = 't';
    output[10] = 'i';
    output[0xb] = 'e';
    output[0xc] = 'y';
    output[0xd] = 'f';
    output[0xe] = 'r';
    output[0xf] = 's';
    output[0x10] = 'l';
    output[0x11] = 'g';
    output[0x12] = 't';
    output[0x13] = 'z';
    output[0x14] = 'u';
    output[0x15] = 'x';
    output[0x16] = 'o';
    output[0x17] = 'v';
    output[0x18] = 'l';
    output[0x19] = 'f';
    output[0x1a] = 'd';
    output[0x1b] = 'n';
    output[0x1c] = 'b';
    output[0x1d] = 'r';
    output[0x1e] = 's';
    output[0x1f] = 'n';
    output[0x20] = 'l';
    output[0x21] = 'r';
    output[0x22] = 'v';
    output[0x23] = 'y';
    output[0x24] = 'h';
    output[0x25] = 'h';
    output[0x26] = 's';
    output[0x27] = 'd';
    output[0x28] = 'x';
    output[0x29] = 'x';
    output[0x2a] = 'r';
    output[0x2b] = 'f';
    output[0x2c] = 'o';
    output[0x2d] = 'x';
    output[0x2e] = 'n';
    output[0x2f] = 'j';
    output[0x30] = 'b';
    output[0x31] = 'l';
    output[0x32] = '\0';
    setvbuf(stdout,(char *)0x0,2,0);
    printf("Enter the secret password: ");
    __isoc99_scanf(&DAT_00402024,input);

    i = 0;
    sVar3 = strlen(output);
    for (; i < 3; i = i + 1) {
      for (i_1 = 0; i_1 < (int)sVar3; i_1 = i_1 + 1) {
        uVar1 = (i_1 % 0xff >> 1 & 0x55U) + (i_1 % 0xff & 0x55U);
        uVar1 = ((int)uVar1 >> 2 & 0x33U) + (uVar1 & 0x33);
        iVar2 = ((int)uVar1 >> 4) + input[i_1] + -0x61 + (uVar1 & 0xf);
        input[i_1] = (char)iVar2 + (char)(iVar2 / 0x1a) * -0x1a + 'a';
      }
    }

    iVar2 = memcmp(input,output,(long)(int)sVar3);
    if (iVar2 == 0) {
      printf("SUCCESS! Here is your flag: %s\n","picoCTF{sample_flag}");
    }
    else {
      puts("FAILED!");
    }
    return 0;
  }
```

In this program, the input is transformed by a `for` loop before being compared to the expected flag. If we enter 'a' as the first character of our input, it gets modified during the loop, so what is actually compared is not the original flag but a transformed version of it. This transformation is controlled by the `for` loop. We can think of this `for` loop as a function, denoted as `f(x)`, where it takes a character `x` and returns a transformed character `f(x)`.

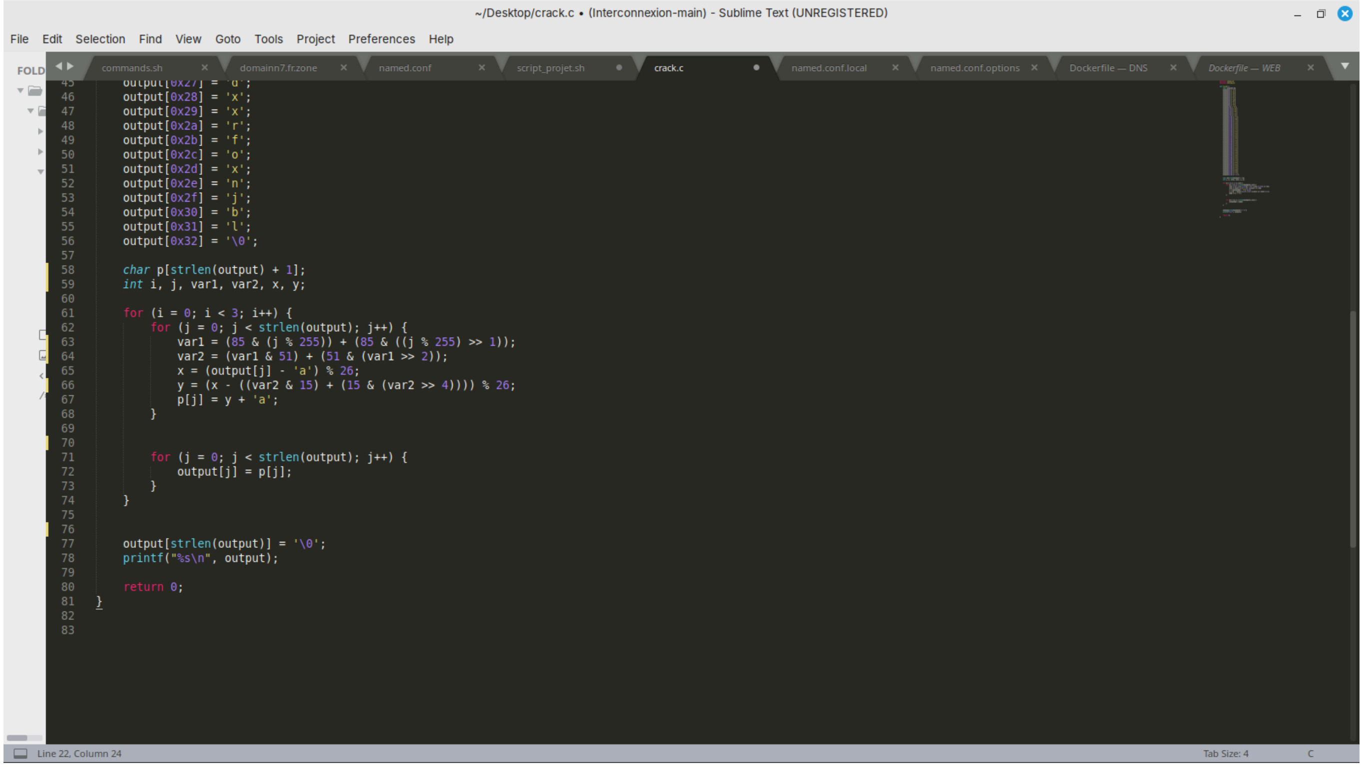
Let's define our input as `i` and the correct solution (or password) as `s`. The comparison is effectively checking if:
`f(i) = s

To control the input and reverse the transformation, we need to find a way to undo the transformation function `f`. This means we need to solve for `i` in terms of `s`, so we have:
`i = f^{-1}(s)

Where `f⁻¹` represents the inverse of the transformation function. Now, we need to reverse the function `f` and determine how to obtain the original input `i` that would result in the correct solution `s`.

I made this solution using C :

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

int main() {
    char output[51];
    output[0] = 'z';
    output[1] = 't';
    output[2] = 'q';
    output[3] = 'i';
    output[4] = 't';
    output[5] = 't';
    output[6] = 'w';
    output[7] = 't';
    output[8] = 'x';
    output[9] = 't';
    output[10] = 'i';
    output[0xb] = 'e';
    output[0xc] = 'y';
    output[0xd] = 'f';
    output[0xe] = 'r';
    output[0xf] = 's';
    output[0x10] = 'l';
    output[0x11] = 'g';
    output[0x12] = 't';
    output[0x13] = 'z';
    output[0x14] = 'u';
    output[0x15] = 'x';
    output[0x16] = 'o';
    output[0x17] = 'v';
    output[0x18] = 'l';
    output[0x19] = 'f';
    output[0x1a] = 'd';
    output[0x1b] = 'n';
    output[0x1c] = 'b';
    output[0x1d] = 'r';
    output[0x1e] = 's';
    output[0x1f] = 'n';
    output[0x20] = 'l';
    output[0x21] = 'r';
    output[0x22] = 'v';
    output[0x23] = 'y';
    output[0x24] = 'h';
    output[0x25] = 'h';
    output[0x26] = 's';
    output[0x27] = 'd';
    output[0x28] = 'x';
    output[0x29] = 'x';
    output[0x2a] = 'r';
    output[0x2b] = 'f';
    output[0x2c] = 'o';
    output[0x2d] = 'x';
    output[0x2e] = 'n';
    output[0x2f] = 'j';
    output[0x30] = 'b';
    output[0x31] = 'l';
    output[0x32] = '\0';

    char p[strlen(output) + 1];
    int i, j, var1, var2, x, y;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < strlen(output); j++) {
            var1 = (85 & (j % 255)) + (85 & ((j % 255) >> 1));
            var2 = (var1 & 51) + (51 & (var1 >> 2));
            x = (output[j] - 'a') % 26;
            y = (x - ((var2 & 15) + (15 & (var2 >> 4)))) % 26;
            p[j] = y + 'a';
        }


        for (j = 0; j < strlen(output); j++) {
            output[j] = p[j];
        }
    }


    output[strlen(output)] = '\0';
    printf("%s\n", output);

    return 0;
}
```



```c
    output[0x27] = 'd';
    output[0x28] = 'x';
    output[0x29] = 'x';
    output[0x2a] = 'r';
    output[0x2b] = 'f';
    output[0x2c] = 'o';
    output[0x2d] = 'x';
    output[0x2e] = 'n';
    output[0x2f] = 'j';
    output[0x30] = 'b';
    output[0x31] = 'l';
    output[0x32] = '\0';

    char p[strlen(output) + 1];
    int i, j, var1, var2, x, y;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < strlen(output); j++) {
            var1 = (85 & (j % 255)) + (85 & ((j % 255) >> 1));
            var2 = (var1 & 51) + (51 & (var1 >> 2));
            x = (output[j] - 'a') % 26;
            y = (x - ((var2 & 15) + (15 & (var2 >> 4)))) % 26;
            p[j] = y + 'a';
        }


        for (j = 0; j < strlen(output); j++) {
            output[j] = p[j];
        }
    }


    output[strlen(output)] = '\0';
    printf("%s\n", output);

    return 0;
}


```

after running our c code :

`ossama@ossama:~/Desktop$ ./a.out` zqncqnqkunc\s]igianqoofjf][bYfg_ilppb_jXroiZflb[\c

So solution is : `zqncqnqkunc\s]igianqoofjf][bYfg_ilppb_jXroiZflb[\c let's try to connect to the validation server to verify :D : ``nc titan.picoctf.net 64798

`Enter the secret password: zqncqnqkunc\s]igianqoofjf][bYfg_ilppb_jXroiZflb[\c

SUCCESS! Here is your flag: picoCTF{s0lv3_angry_symb0ls_XXXXXXXXXXXXX},

I can't give you the flag directly :(Sorry, it's forbidden.