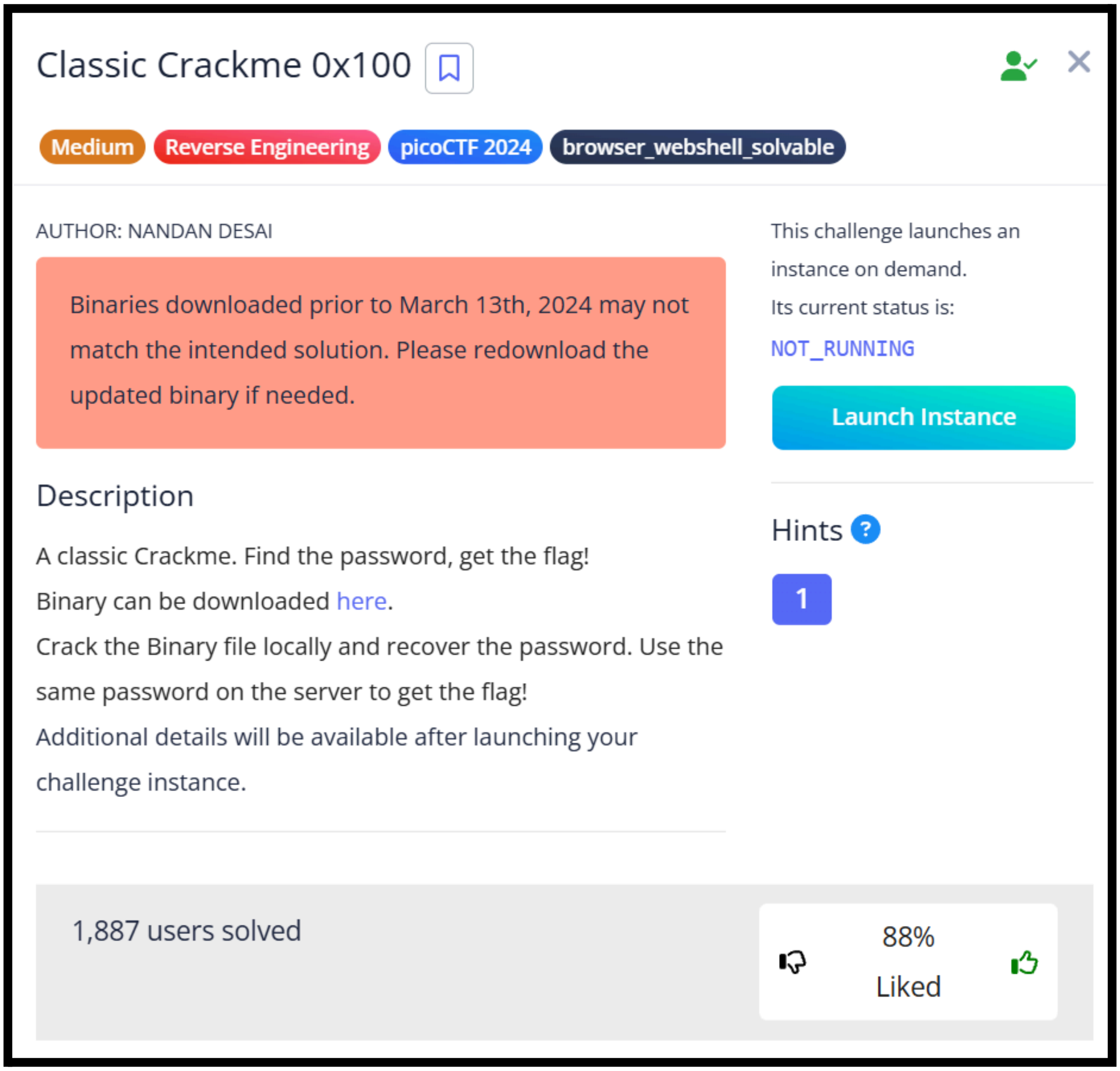
Written by Billie
https://play.picoctf.org/practice/challenge/409



Classic Crackme 0x100

Medium | Reverse Engineering | picoCTF 2024 | browser_webshell_solvable

AUTHOR: NANDAN DESAI

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

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.

This challenge launches an instance on demand.
Its current status is:
NOT_RUNNING

1,887 users solved

88% Liked

I'm not gonna explain how to open the binary file into a disassembler. Just know that essentially all I did was open the file using IDA and let it do its thing. After that, I located the main function which consists of the code below:

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  char input[64]; // [rsp+0h] [rbp-A0h] BYREF
  char output[60]; // [rsp+40h] [rbp-60h] BYREF
  int random2; // [rsp+7Ch] [rbp-24h]
  int random1; // [rsp+80h] [rbp-20h]
  char fix; // [rsp+87h] [rbp-19h]
  int secret3; // [rsp+88h] [rbp-18h]
  int secret2; // [rsp+8Ch] [rbp-14h]
  int secret1; // [rsp+90h] [rbp-10h]
  int len; // [rsp+94h] [rbp-Ch]
  int i_0; // [rsp+98h] [rbp-8h]
  int i; // [rsp+9Ch] [rbp-4h]

  strcpy(output, "apijaczhzgtfnyjgrdvqrjbmcurcmjczsvbwgdelvxxxjkyigy");
  setvbuf(_bss_start, 0, 2, 0);
  printf("Enter the secret password: ");
  __isoc99_scanf("%50s", input);
  i = 0;
  len = strlen(output);
  secret1 = 85;
  secret2 = 51;
  secret3 = 15;
  fix = 97;
  while ( i <= 2 )
  {
    for ( i_0 = 0; i_0 < len; ++i_0 )
    {
      random1 = (secret1 & (i_0 % 255)) + (secret1 & ((i_0 % 255) >> 1));
      random2 = (random1 & secret2) + (secret2 & (random1 >> 2));
      input[i_0] = ((random2 & secret3) + input[i_0] - fix + (secret3 & (random2 >>
4))) % 26 + fix;
    }
    ++i;
  }
  if ( !memcmp(input, output, len) )
    printf("SUCCESS! Here is your flag: %s\n", "picoCTF{sample_flag}");
  else
    puts("FAILED!");
  return 0;
}
```

As you can see above, IDA did most of the work already. If we were using Ghidra, we would have to assemble the long string in the “output” variable by analyzing some memory. But luckily, we don’t have to do that in IDA.

After shortly reading the code above, we see that we have to input our password, and then that password would be encrypted using the algorithm below.

```
  while ( i <= 2 )
  {
    for ( i_0 = 0; i_0 < len; ++i_0 )
    {
      random1 = (secret1 & (i_0 % 255)) + (secret1 & ((i_0 % 255) >> 1));
      random2 = (random1 & secret2) + (secret2 & (random1 >> 2));
      input[i_0] = ((random2 & secret3) + input[i_0] - fix + (secret3 & (random2 >>
4))) % 26 + fix;
    }
    ++i;
  }
```

After the encryption, then the code would compare our encrypted password with the string from the variable “output” as shown in the code snippet below

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

Our task is to find the input so that when the input is encrypted, it results in the same string as the “output” variable. The approach that I’m gonna use is to decrypt the “output” variable itself. To do this we must reverse the encryption.

```
  while ( i <= 2 )
  {
    for ( i_0 = 0; i_0 < len; ++i_0 )
    {
      random1 = (secret1 & (i_0 % 255)) + (secret1 & ((i_0 % 255) >> 1));
      random2 = (random1 & secret2) + (secret2 & (random1 >> 2));
      input[i_0] = ((random2 & secret3) + input[i_0] - fix + (secret3 & (random2 >>
4))) % 26 + fix;
    }
    ++i;
  }
```

After reading the encryption algorithm and the main function carefully we realize that:

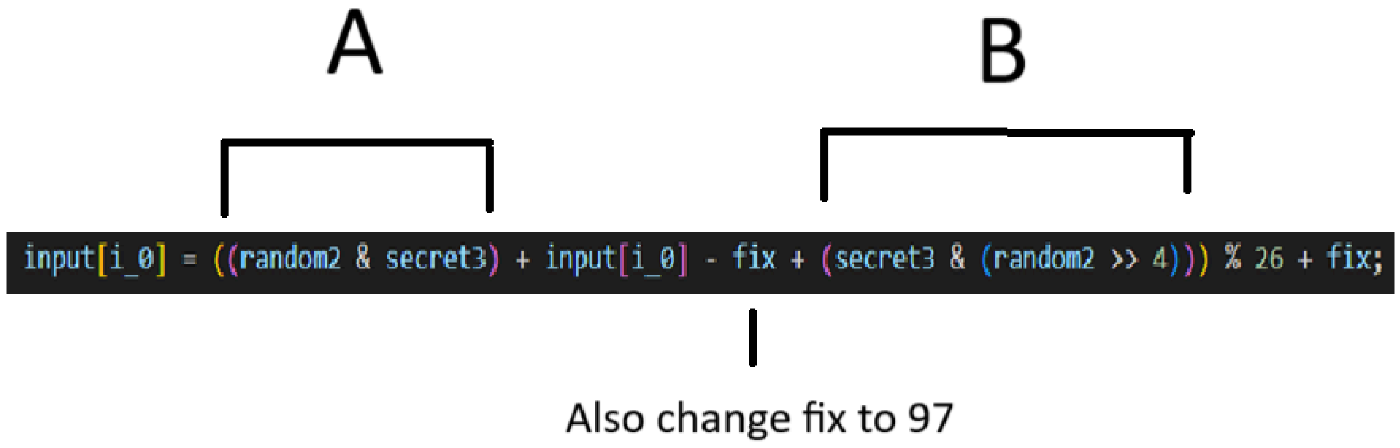
1. The variables “random1” and “random2” change each for-loop iteration.
2. The variables “secret1”, “secret2”, “secret3” and “fix” are constant.
3. Our input is encrypted using the algorithm 3 times (the for loop is looped 3 times by the while loop).
4. Even when “random1” and “random2” are changed in each iteration of the for-loop, they don’t change in each for-loop instance in the while loop iteration. To illustrate this point:

```
For-loop 1:
    For-loop1 iteration 1:
        Random1 = a
        Random2 = b
    For-loop1 iteration 2:
        Random1 = b
        Random2 = c
    .
    .
    .
```

```
For-loop 2:
    For-loop2 iteration 1:
        Random1 = a
        Random2 = b
    For-loop2 iteration 2:
        Random1 = b
        Random2 = c
    .
    .
    .
```

```
For-loop 3:
    For-loop3 iteration 1:
        Random1 = a
        Random2 = b
    For-loop3 iteration 2:
        Random1 = b
        Random2 = c
    .
    .
    .
```

As you can see above, there are 3 instances of for-loops. But in each iteration, the “random1” and “random2” variables are the same in each for-loop iteration. This has several implications that I’m going to explain below



```
input[i_0] = ((random2 & secret3) + input[i_0] - fix + (secret3 & (random2 >> 4))) % 26 + fix;
```

To simplify the algorithm, we change it into

```
input[i]= (A[i] + input[i] -97 + B[i])%26 + 97;
```

The algorithm above is essentially what the encryption algorithm really is. This is it in its simplest form.

Why can we do this? As stated above, in each for-loop initialization, “random1” and “random2” are the same. That means “A” and “B” must also be the same in each for-loop initialization because they are only affected by “random1”, “random2”, and other constant variables. This way, we can make an array for the value A and B just by copying the code and saving the value.

```
      A[i] = (random2 & secret3);
      B[i] = (secret3 & (random2 >> 4));
```

A = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 0, 1}

B = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2}
These are all the values for A and B.

The string "`apijaczhzgtfnyjgrdvqrjbmcurcmjczsvbwgdelvxxxjkyigy`" is essentially the password that we are looking for that have been inside the "output" variable will turn into:
Output = {97,112,105,106,97,99,122,104,122,103,116,102,110,121,106,103,114,100,118,113,114,106,98,109,99,117,114,99,109,106,99,122,115,118,98,119,103,100,101,108,118,120,120,120,106,107,121,105,103,121}
The numbers above are essentially the string that is inside the "output" variable. Since we have converted them into integers, I will now refer "output" variable as the output array.

We subtract all of those numbers by 97 so that we have the "equation":

```
 output[i]= (A[i] + input[i] + B[i])%26 //notice -97 and +97 is gone
```

This "equation" is essentially the entire encryption algorithm simplified. The reason -97 and +97 is gone is simply because in each iteration of the encryption program it adds 97 (note the + 97 after the module operation), and in the next encryption it will subtract 97 from the values of the encrypted program. Since this is redundant, we can just erase them.

After we subtract 97 in each of the numbers, we have the output array as shown below:
Output = {0, 15, 8, 9, 0, 2, 25, 7, 25, 6, 19, 5, 13, 24, 9, 6, 17, 3, 21, 16, 17, 9, 1, 12, 2, 20, 17, 2, 12, 9, 2, 25, 18, 21, 1, 22, 6, 3, 4, 11, 21, 23, 23, 23, 9, 10, 24, 8, 6, 24}
These numbers are essentially the product of the encryption algorithm above.

We can ignore the %26 because essentially the numbers above are some value above 26 that has been reduced into numbers below 26. For all we know, when we see 15, the original value might be 15, 41 (15+26), 67 (15+26+26). That means, if we input the character ')' (with an ascii value of 41) or the character 'c' (with an ascii value of 67), it will all be transformed into the value 15. So we can just rewrite the encryption algorithm into:

```
output[i] = (A[i] + input[i] + B[i])
```

Using basic algebra we find that

```
output[i]- A[i] -B[i] = input[i]
```

But we must also remember that the encryption algorithm is done three times, so we must also do the decryption algorithm 3 times. So the decryption algorithm would be

```
 input[i] = output[i] - 3*A[i] - 3*B[i] + 97;
```

Note that we added 97 in the end, that is because the values are too small for regular characters. We know that the password is originally from ranges 33 to 126 since those are the ascii range for commonly used characters, and adding 97 will return us right to that range.

The full decryption program is this:

```
#include <stdio.h>

int main(){
    int output[50] = {0, 15, 8, 9, 0, 2, 25, 7, 25, 6, 19, 5, 13, 24, 9, 6, 17, 3,
21, 16, 17, 9, 1, 12, 2, 20, 17, 2, 12, 9, 2, 25, 18, 21, 1, 22, 6, 3, 4, 11, 21, 23,
23, 23, 9, 10, 24, 8, 6, 24};
    int A[50] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 0, 1, 1, 2, 1, 2, 2,
3, 1, 2, 2, 3, 2, 3, 3, 4, 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 0, 1};
    int B[50] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2};
    int input[50];
    for(int i = 0; i<50; i++){
        input[i] = 0;
        input[i] = output[i] - 3*A[i] - 3*B[i] + 97;
        printf("%d ", input[i]);
    }
    return 0;
}
```

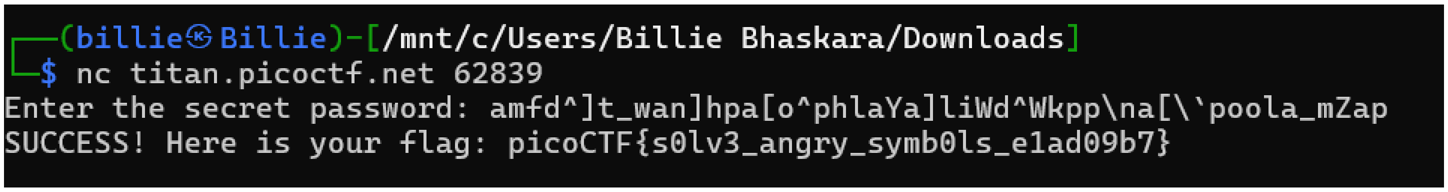
Running this program we get the output

```
97 109 102 100 94 93 116 95 119 97 110 93 104 112 97 91 111 94 112 104 108 97 89 97 93
108 105 87 100 94 87 107 112 112 92 110 97 91 92 96 112 111 111 108 97 95 109 90 97 112
```

Which if we convert them into ascii character will get us

amfd^]t_wan]hpa[o^phlaYa]liWd^Wkpp\na[\`poola_mZap

This is the password that will get us the flag



```
┌──(billie㉿Billie)-[/mnt/c/Users/Billie Bhaskara/Downloads]
└─$ nc titan.picoctf.net 62839
Enter the secret password: amfd^]t_wan]hpa[o^phlaYa]liWd^Wkpp\na[\`poola_mZap
SUCCESS! Here is your flag: picoCTF{s0lv3_angry_symb0ls_e1ad09b7}
```