NOVA CTF {2023} - C2a8k

[REVERSE ENGINEERING CHALLENGE]

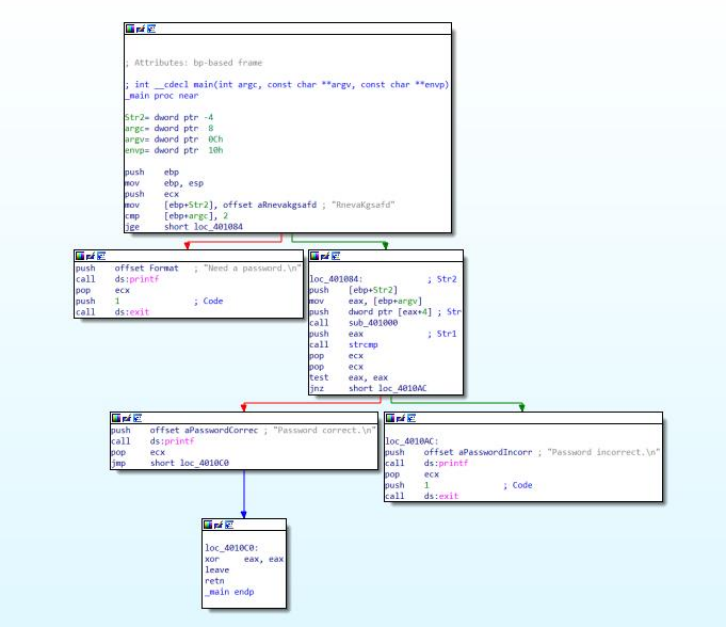
Given Description:

Eathen has obtained a laptop from an enemy agent, which contains crucial information about a planned attack. However, the laptop is password protected and Eathen needs to gain access to it quickly.

Analysing the File:

The Given File is: **virus.exe**

After opening the File in IDA, we can see that virus.exe is small Program with few blocks of code. I opened this in IDA. You can use any disassembler or debugger you are comfortable will work.



- After Analysing the main function, we can see a string “RnevaKgsafd” being initialized and placed into a variable named str2.
- The value of argc is checked to see if it’s equal to 2.
- The next line jumps to 0x401084 (via the green line) if argc is >= 2, otherwise the code execution continues to 0x401070 where it prints out that it needs a passwords.
- The program needs at least one Command Line argument and it gives error message, we can probably infer this is the password.
- In 0x401084: The program pushes our string argument onto the stack. Next, it moves the value of argv into register eax, and in the next line pushes its value + 4 onto the stack, a call to function sub_401000 is made.
- The push statements are used to push the function arguments onto the stack.
- The program pushes two arguments onto the stack before the call to sub_401000, which indicates this function takes two arguments.

- After calling sub_401000, the return value (eax) is pushed to the stack and strcmp is called and String comparison function strcmp returns 0 if the strings given to it match.
- If the strings were not equal, or eax is not 0, then the string “Password incorrect” is printed and the program exits.
- If the strings match, or eax is 0, then “Password correct” is printed.
- Therefore, sub_401000 must be encoding the password we enter in and comparing the result with the already encoded password.
- In order to figure out how the password is encoded, we have to analyse that function.

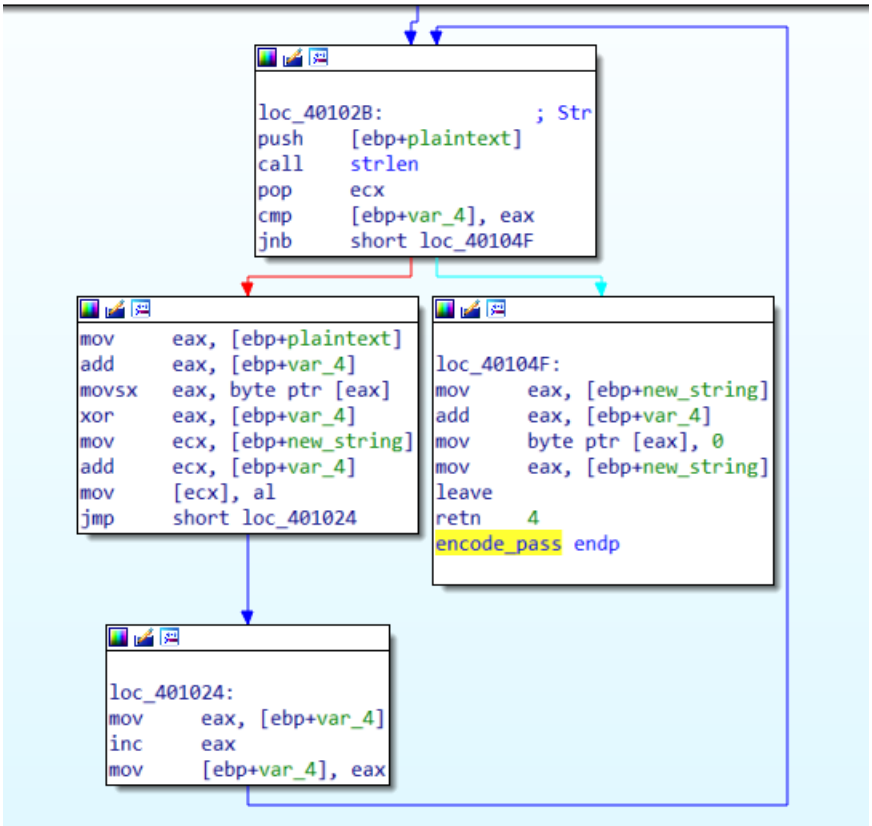
sub_401000

```
; int __stdcall encode_pass(char *plaintext)
encode_pass proc near

new_string= dword ptr -8
var_4= dword ptr -4
plaintext= dword ptr  8

push    ebp
mov     ebp, esp
push    ecx
push    ecx
and     [ebp+var_4], 0
push    [ebp+plaintext] ; Str
call    strlen          ; get strlen of the string
pop     ecx
inc     eax
push    eax             ; Size
call    ds:malloc       ; eax=mem size of plaintext + 1
pop     ecx
mov     [ebp+new_string], eax
and     [ebp+var_4], 0  ; var_4=0
jmp     short loc_40102B
```

- I renamed sub_401000 to “encode_pass”, since we know that is what its doing.
- Additionally, I renamed the parameter passed in, originally called arg_0, to “plaintext”. This is because we know the parameter being passed in is our plaintext password.
- The function strlen is called with our plaintext string as the parameter. This will return the length of our password in register eax.
- The length of plaintext is incremented by 1 and pushed to the stack. The memory allocation function malloc is called to allocate memory the size of plaintext + 1; its location is stored in eax.
- Adding 1 to the length of a string is commonly done in string operations to account for the extra null value at the end of the string.
- The newly created memory is stored in local variable var_8, which I have renamed new_string.
- Local variable var_4 is logically AND’d with 0, which is a fast way to set a value to 0. If you look at this section of code with the graphical view in IDA, you can see that the chain of code can circle back to the beginning of this section.
- This is indicative of a loop. When we are dealing with encoding/decoding routines, loops are very important to look at because they are likely where data is encoded/decoded.



The first block of code:

- The length of our plaintext password is taken again and stored into eax. The length is compared to var_4.
- If var_4 >= the length, the code starting at 0x40104F is jumped to, otherwise code execution continues. This is indicative of a while loop.
- It is starting to look like var_4 is an iterative variable, and the code here likely is: var_4 = 0; while (var_4 < strlen(plaintext)) { … } .
- Since var_4 appears to be an iterative variable used to control the loop, I'll rename it as i (a commonly used iterative variable name).
- Since it appears that the code block in 0x401039 is the code in the loop, we'll analyse that next. The memory location containing plaintext is moved to eax, then i (var_4) is added to eax.
- This is commonly how compilers access a specific memory location in an array. In terms of code, it would translate to: plaintext[i] .
- The character at plaintext[i] is then moved to eax.
- This translates in C-code to: eax = plaintext[i]
- The value of i (var_4) is XOR'd with eax (plaintext[i]); the result is stored in eax.
- XOR is used A LOT with encoding routines, so this looks to be the line the password is encoded.
- The code is essentially XOR'ing the plaintext character with its location value. (Remember that array indexes start at 0)
- Translated to C-code, we now have: eax = plaintext[i] ^ i .
- The next three lines find the correct place in new_string and copy the XOR'd value into it. new_string[i] = (byte) eax.
- Puts a null value at the end of new_string.

- This is done as C-strings require the null character at the end to indicate where the string ends.
- Moves new_string into eax and then leaves the function.

The entire fuction can be converted to code below:

```
sub encode_pass(char *plaintext) {
        new_string = malloc(strlen(plaintext) + 1);
        i = 0;
        while (i < strlen(plaintext) ) {
         new_str[i] = plaintext[i] ^ i;
         i++;
         }
        new_str[i] = 0x0;
        return new_str;
        }
```

- The algorithm used to encode the password does so by XOR'ing each character in the password by its location.
- XOR is commonly used by malware because it's easy to implement, but XOR encoding – even with variable length keys – has weaknesses.
- The one weakness we can take advantage of here is part of how XOR works.
- plaintext XOR key = ciphertext
- ciphertext XOR key = plaintext
- In other words, if we XOR the ciphertext by the XOR key, we'll get the plaintext! And we already know the XOR key – it's a sequence of incrementing bytes, starting at 0x0.
- In order to decrypt the ciphertext key -we need to write a small script to xor it against the key.

```
ciphertext ='RnevaKgsafd'
i = 0
plaintext = ''
while (i < len(ciphertext)):
 plaintext = plaintext + chr(ord(ciphertext[i]) ^ i)
 i = i + 1
print(plaintext)
```

main.py

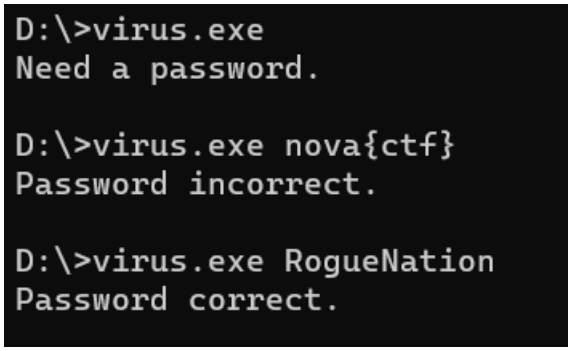
Run

Shell

```
1  ciphertext ='RnevaKgsafd'
2  i = 0
3  plaintext = ''
4  while (i < len(ciphertext)):
5   plaintext = plaintext + chr(ord(ciphertext[i]) ^ i)
6   i = i + 1
7  print(plaintext)
```

```
RogueNation
>
```

When we run the python program ,we get the decoded password , RogueNation.



By running the main program that is virus.exe and giving the password we found , we can verify the password is correct. That will provide you the final flag of this CTF challenge.

Enclosing the Flag:
NOVA{RogueNation}

I hope you learnt something and enjoyed the challenge. Connect me via LinkedIn, https://www.linkedin.com/in/kishoreram-k/

Best of luck in capturing flags ahead!!!