Adithyaa Sivamal

Reverse Engineering Exercise Writeup

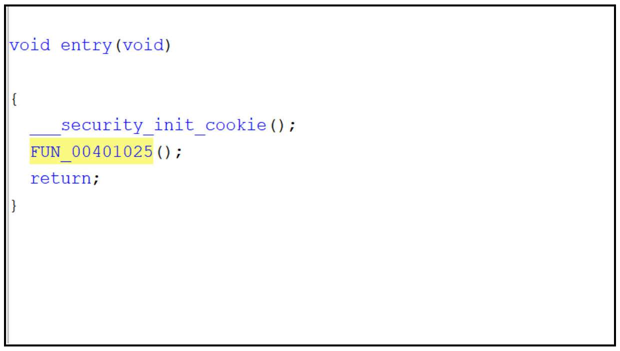
Exer2_Julia

Sample: Exer2_Julia.exe

Link: https://www.begin.re/julia

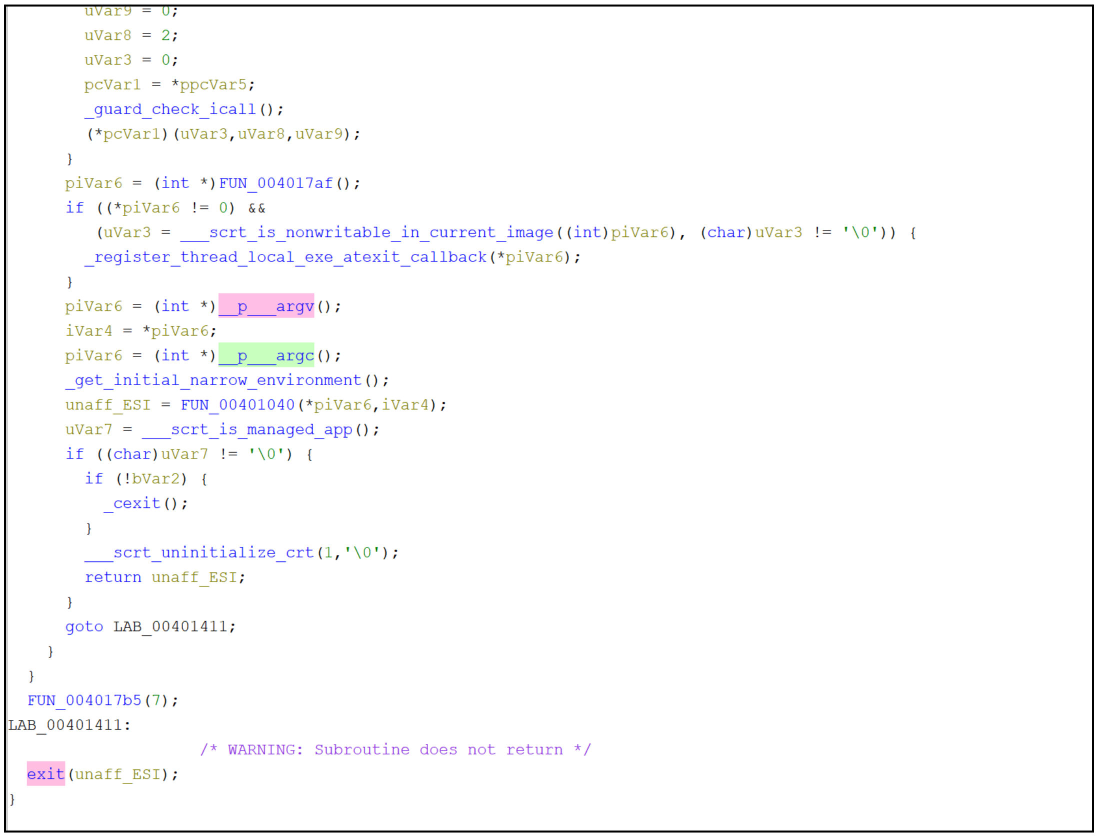
Ghidra Static Analysis

We have a **Windows Executable** which calls to a single function from entry.



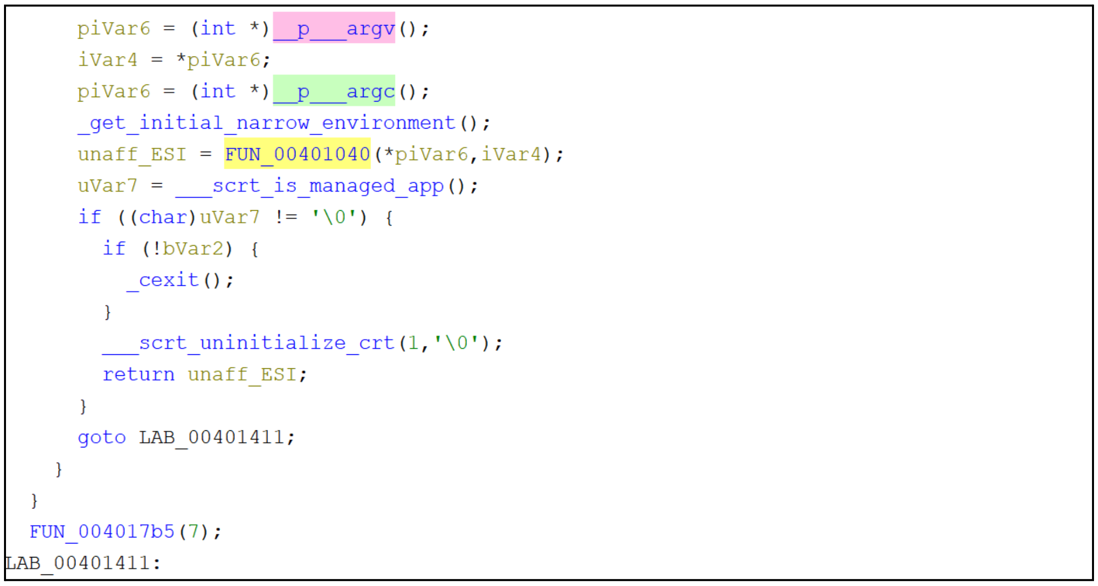
```
void entry(void)

{
  ___security_init_cookie();
  FUN_00401025();
  return;
}
```

We can ensure that it's the **main()** function as it can call **argv()** and **argc()** (in this case by their references). Since the function also doesn't return and instead exits after executing, we can assume it's the main().



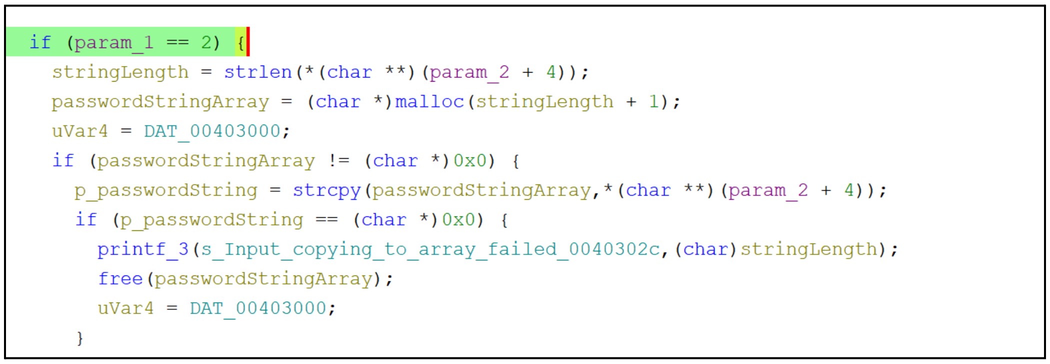
```
        uVar9 = 0;
        uVar8 = 2;
        uVar3 = 0;
        pcVar1 = *ppcVar5;
        _guard_check_icall();
        (*pcVar1)(uVar3,uVar8,uVar9);
      }
      piVar6 = (int *)FUN_004017af();
      if ((*piVar6 != 0) &&
         (uVar3 = ___scrt_is_nonwritable_in_current_image((int)piVar6), (char)uVar3 != '\0')) {
        _register_thread_local_exe_atexit_callback(*piVar6);
      }
      piVar6 = (int *)__p___argv();
      iVar4 = *piVar6;
      piVar6 = (int *)__p___argc();
      _get_initial_narrow_environment();
      unaff_ESI = FUN_00401040(*piVar6,iVar4);
      uVar7 = ___scrt_is_managed_app();
      if ((char)uVar7 != '\0') {
        if (!bVar2) {
          _cexit();
        }
        ___scrt_uninitialize_crt(1,'\0');
        return unaff_ESI;
      }
      goto LAB_00401411;
    }
  }
  FUN_004017b5(7);
LAB_00401411:
                    /* WARNING: Subroutine does not return */
  exit(unaff_ESI);
}
```

We can see references to __p__argv() and __p__argc() are being set to **piVar6** and **iVar4**, which are then passed as parameters to the **FUN_00401040()**.

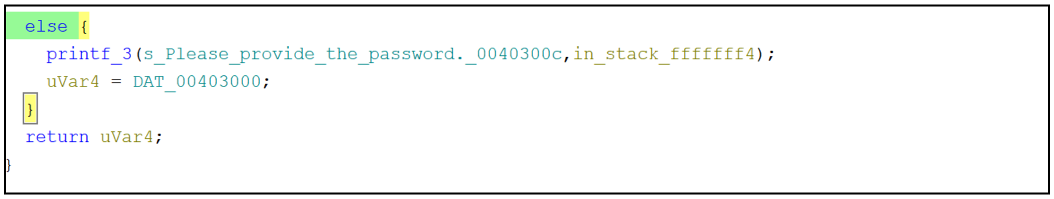


```
      piVar6 = (int *)__p___argv();
      iVar4 = *piVar6;
      piVar6 = (int *)__p___argc();
      _get_initial_narrow_environment();
      unaff_ESI = FUN_00401040(*piVar6,iVar4);
      uVar7 = ___scrt_is_managed_app();
      if ((char)uVar7 != '\0') {
        if (!bVar2) {
          _cexit();
        }
        ___scrt_uninitialize_crt(1,'\0');
        return unaff_ESI;
      }
      goto LAB_00401411;
    }
  }
  FUN_004017b5(7);
LAB_00401411:
```

Within **FUN_00401040()**, we find the first line to be checking if argc() == 2, in other words whether two arguments were inputted into the command line or not. Scrolling down will tell us that if argc() != 2, the message “Please provide the password.” is returned and displayed.

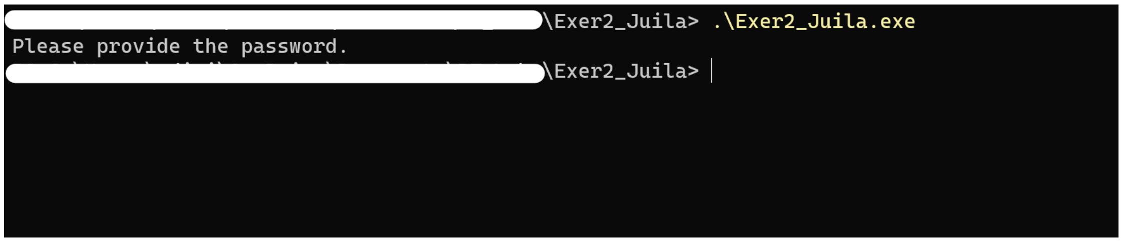


```
  if (param_1 == 2) {
    stringLength = strlen(*(char **)(param_2 + 4));
    passwordStringArray = (char *)malloc(stringLength + 1);
    uVar4 = DAT_00403000;
    if (passwordStringArray != (char *)0x0) {
      p_passwordString = strcpy(passwordStringArray,*(char **)(param_2 + 4));
      if (p_passwordString == (char *)0x0) {
        printf_3(s_Input_copying_to_array_failed_0040302c,(char)stringLength);
        free(passwordStringArray);
        uVar4 = DAT_00403000;
      }
```



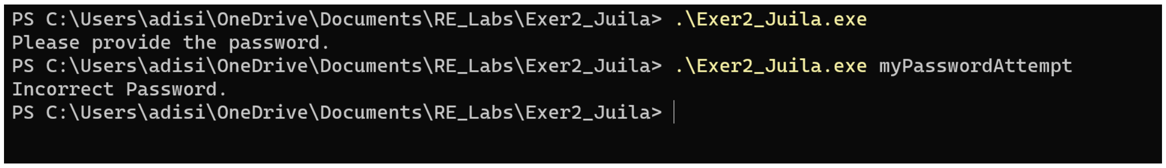
```
  else {
    printf_3(s_Please_provide_the_password._0040300c,in_stack_ffffffff4);
    uVar4 = DAT_00403000;
  }
  return uVar4;
}
```

Here's the function in action.



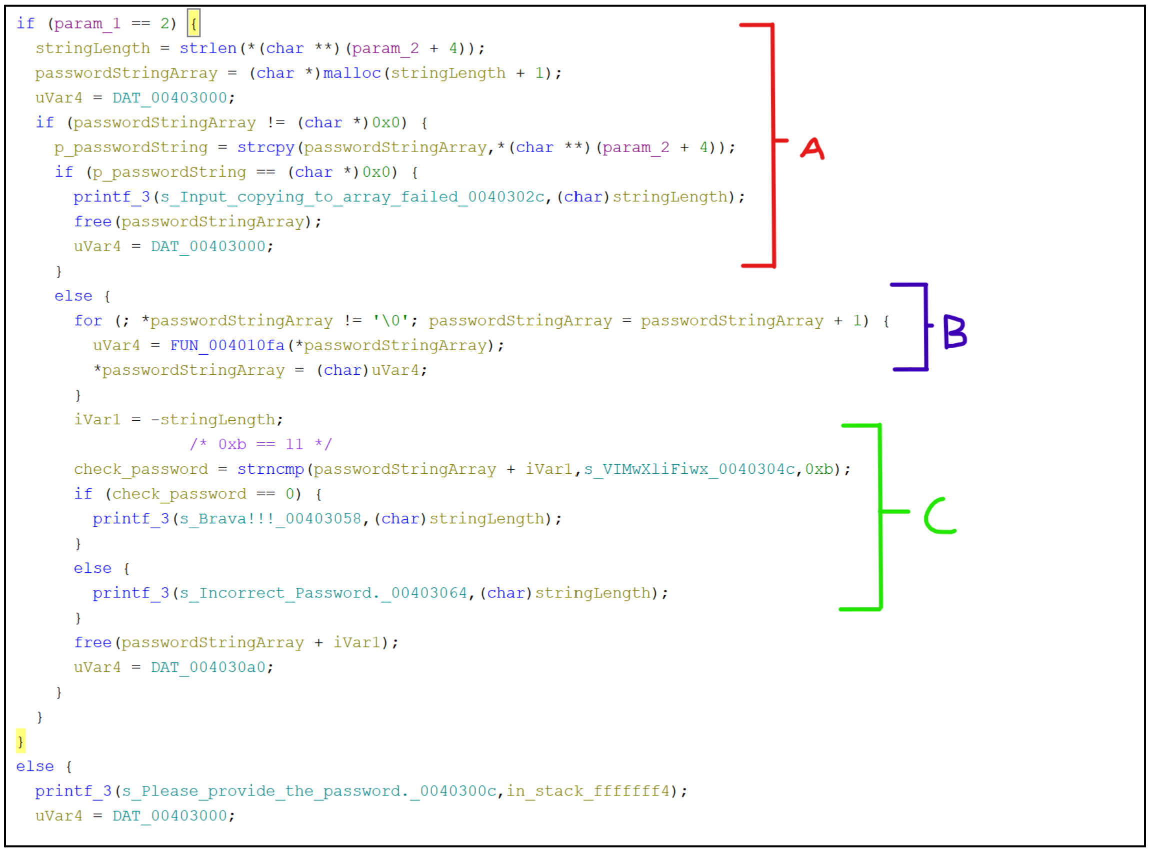
```
[redacted]\Exer2_Juila> .\Exer2_Juila.exe
Please provide the password.
[redacted]\Exer2_Juila>
```

We can also see that if I provide a second argument as the incorrect password, it will return and display "Incorrect password."



```
PS C:\Users\adisi\OneDrive\Documents\RE_Labs\Exer2_Juila> .\Exer2_Juila.exe
Please provide the password.
PS C:\Users\adisi\OneDrive\Documents\RE_Labs\Exer2_Juila> .\Exer2_Juila.exe myPasswordAttempt
Incorrect Password.
PS C:\Users\adisi\OneDrive\Documents\RE_Labs\Exer2_Juila>
```

After looking around the function and renaming some variables, we can better understand what's going on within the if-else statement.



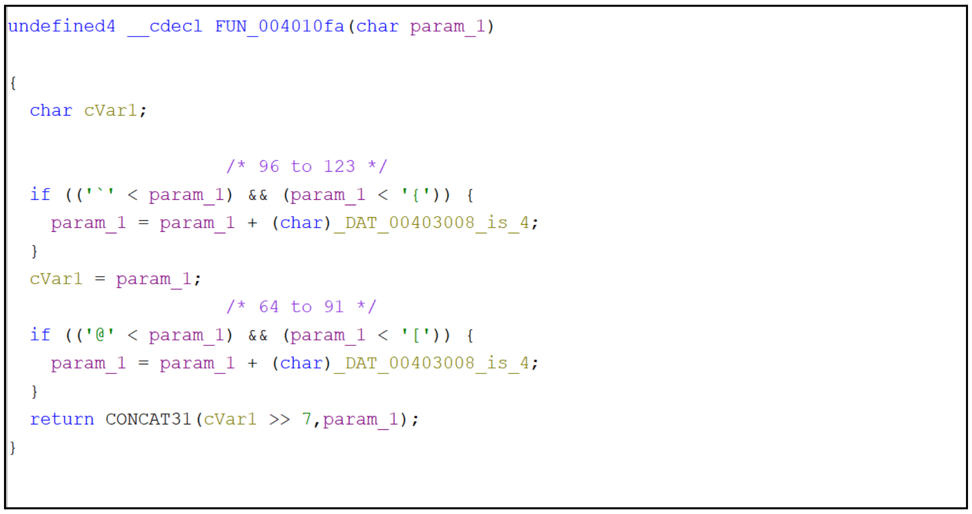
```
if (param_1 == 2) {
  stringLength = strlen(*(char **)(param_2 + 4));
  passwordStringArray = (char *)malloc(stringLength + 1);
  uVar4 = DAT_00403000;
  if (passwordStringArray != (char *)0x0) {
    p_passwordString = strcpy(passwordStringArray,*(char **)(param_2 + 4));
    if (p_passwordString == (char *)0x0) {
      printf_3(s_Input_copying_to_array_failed_0040302c,(char)stringLength);
      free(passwordStringArray);
      uVar4 = DAT_00403000;
    }
    else {
      for (; *passwordStringArray != '\0'; passwordStringArray = passwordStringArray + 1) {
        uVar4 = FUN_004010fa(*passwordStringArray);
        *passwordStringArray = (char)uVar4;
      }
      iVar1 = -stringLength;
                  /* 0xb == 11 */
      check_password = strncmp(passwordStringArray + iVar1,s_VIMwXliFiwx_0040304c,0xb);
      if (check_password == 0) {
        printf_3(s_Brava!!!_00403058,(char)stringLength);
      }
      else {
        printf_3(s_Incorrect_Password._00403064,(char)stringLength);
      }
      free(passwordStringArray + iVar1);
      uVar4 = DAT_004030a0;
    }
  }
}
else {
  printf_3(s_Please_provide_the_password._0040300c,in_stack_fffffff4);
  uVar4 = DAT_00403000;
```

In *section A*, a char-array **'passwordStringArray'** with its memory set to equal the length of param_2. It'll then check if it's empty to output "input copying to array failed" and break out the loop.

If the passwordStringArray isn't empty, it'll move onto se*ction B*, where each char is processed by the **FUN_004010fa()** function.

In *section C,* he processed PasswordStringArray is then checked against the string "**VIMwXliFiwx**" with strncmp(). If they're equal, the program will print "Brava!!", otherwise "Incorrect Password".

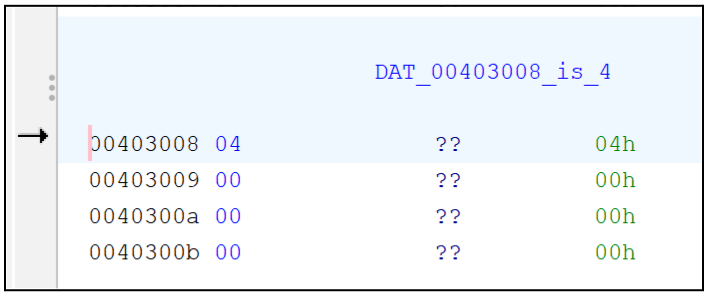
We can see that FUN_004010fa() is doing some transformation to param_2, then checked against the string “**VIMwXliFiwx**”. We can see the transformation code within the function here:



```
undefined4 __cdecl FUN_004010fa(char param_1)

{
  char cVar1;
  
                    /* 96 to 123 */
  if (('`' < param_1) && (param_1 < '{')) {
    param_1 = param_1 + (char)_DAT_00403008_is_4;
  }
  cVar1 = param_1;
                    /* 64 to 91 */
  if (('@' < param_1) && (param_1 < '[')) {
    param_1 = param_1 + (char)_DAT_00403008_is_4;
  }
  return CONCAT31(cVar1 >> 7,param_1);
}
```

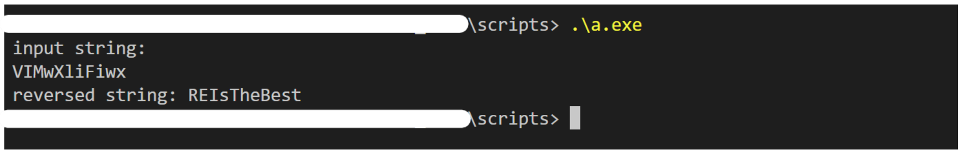
The function takes a character as input and transforms it based on two specified rules. If the character falls within the ASCII range of 'a' to 'z' or 'A' to 'Z', it shifts the character by a certain amount (**_DAT_00403008**). Then, it returns the transformed character. Essentially, it’s just a simple Caesar cypher.

The reference tells us that _DAT_00403008 is equal to int 4.



```
                        DAT_00403008_is_4
00403008 04             ??          04h
00403009 00             ??          00h
0040300a 00             ??          00h
0040300b 00             ??          00h
```

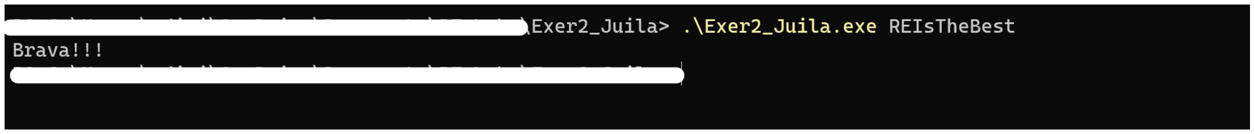
From here, we can write our own C++ program identical to FUN_004010fa(), but instead sets _DAT00403008 as –4.



```
[redacted]\scripts> .\a.exe
input string:
VIMwXliFiwx
reversed string: REIsTheBest
[redacted]\scripts>
```

We know that the transformed param_2 is checked against the string **“VIMwXliFiwx”**. So reverse Caesar-shifting this string should give us the correct password. Doing so with the script outputs “**REIsTheBest**”

We then verify if it’s the correct password by running ‘**Exer2_Julia.exe REIsTheBest**’.



```
[redacted]Exer2_Juila> .\Exer2_Juila.exe REIsTheBest
Brava!!!
[redacted]
```

“Brava!!!” is returned, indicating “REIsTheBest” is the correct password.

---- END ----