МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра Інформаційної Безпеки

Захист програмного забезпечення

Лабораторна робота 4

Аналіз механізмів захисту додатку та їх блокування

Мета роботи: навчитися використовувати засоби статичного і динамічного аналізу програм. Отримати навички модифікації бінарного коду додатка.

Перевірив:

_____

Виконав:
студент III курсу
групи ФБ-01
Сахній Н.Р.

Київ 2023

Завдання:

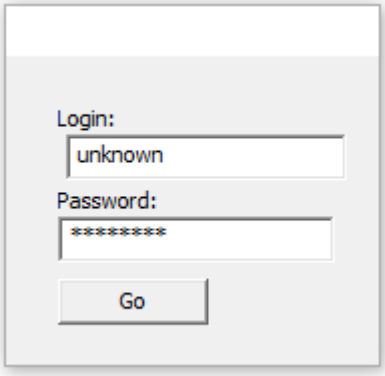
Дослідимо програму із наданого файлу, який має назву **`Crack_me_up!.exe`**, що володіє захистом від свого несанкціонованого використання і зламаємо захист різними способами.

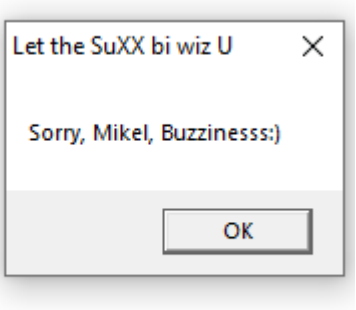
У процесі виконання лабораторної роботи виконаємо наступне:

1. ----------

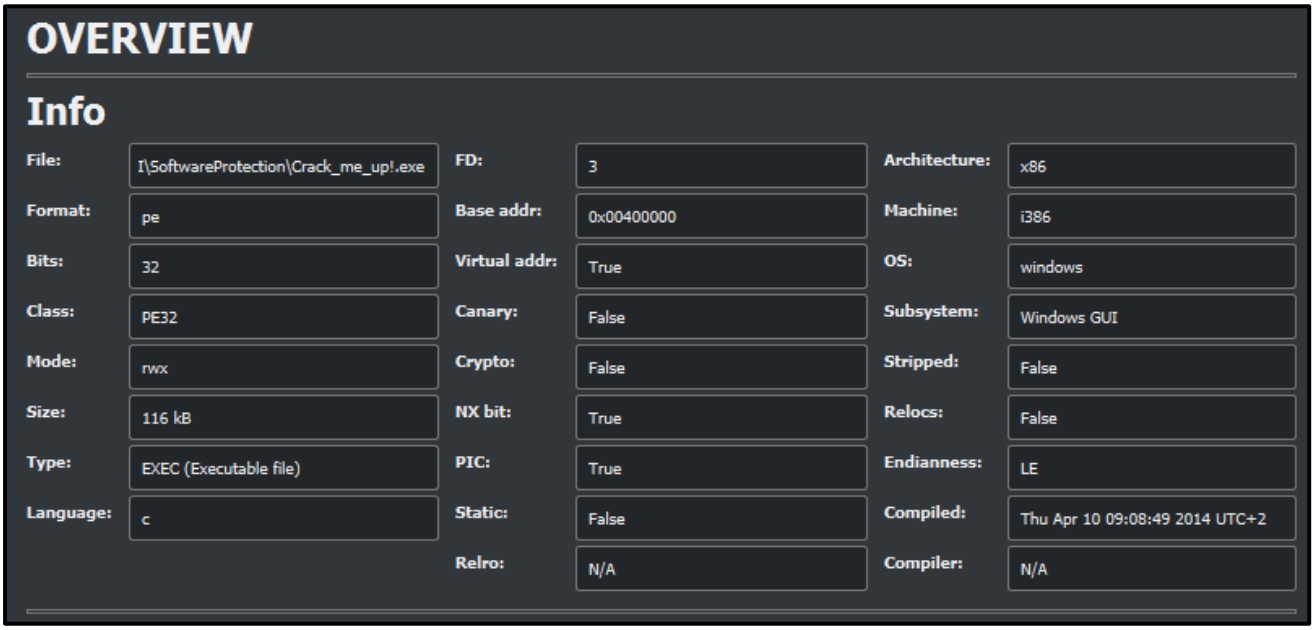
- Виділимо в досліджуваній програмі ділянку коду, що виконує функцію прийняття рішення про коректність введеного пароля:

– Запускаємо перший раз програму на виконання, і після введення довільних облікових даних бачимо, що у вікні із назвою “**Let the SuXX bi wiz U**” відображається текст “**Sorry, Mikel, Buzzinesss:)???**”.

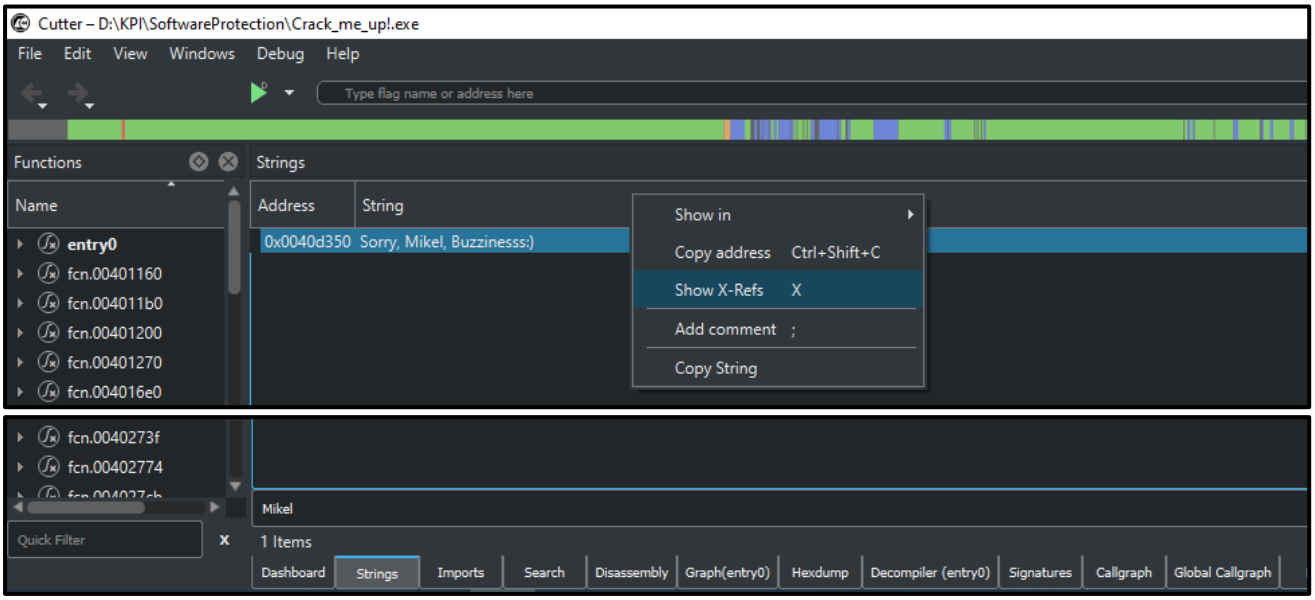




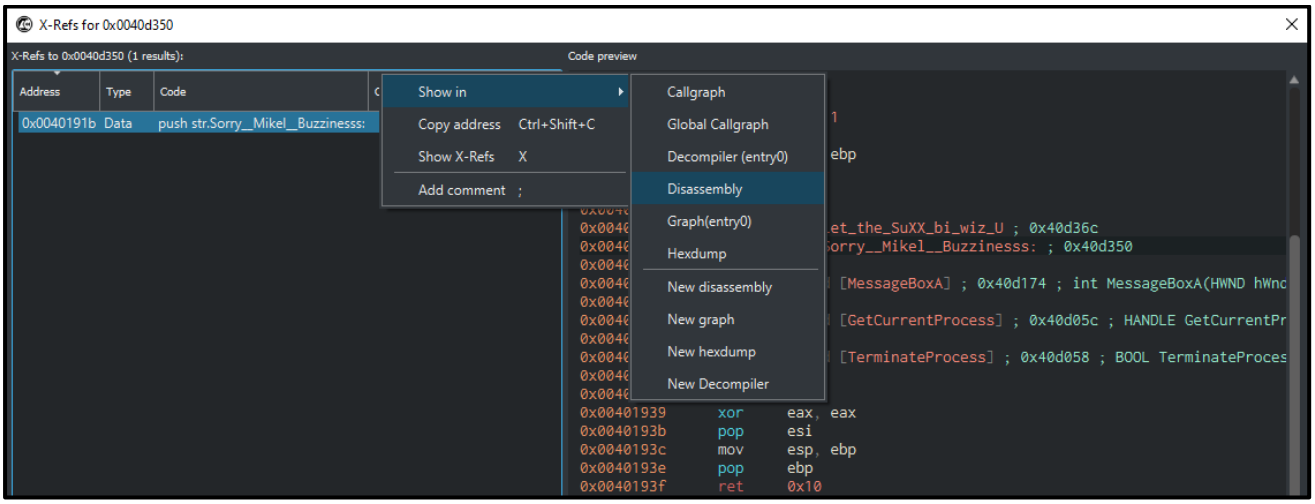
– Відкриваємо наш файл у відповідному застосунку **`cutter.exe`** (досить важливий, ефективний та багатофункціональний засіб для зворотної розробки програмного забезпечення).



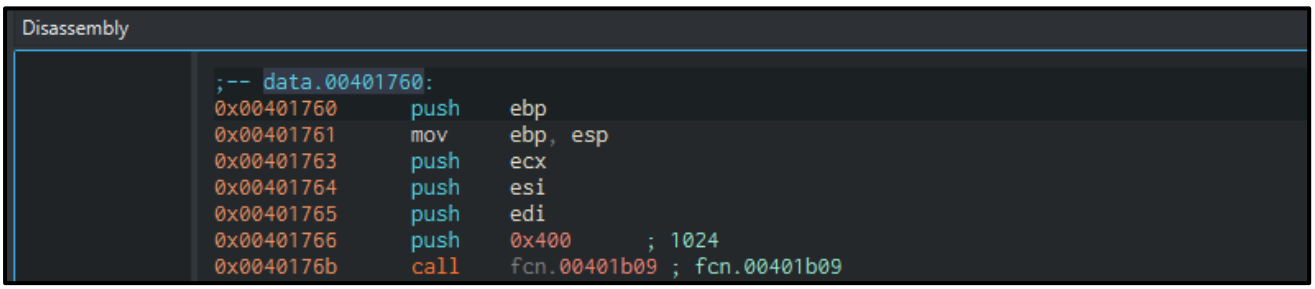
– Далі шукаємо у вікні “**Strings**” попередньо згаданий рядок тексту та пробуємо переглянути, у якому місці коду програма посилається на цей запис.



– Знайдемо цей рядок в дизасемблері та будемо шукати, за допомогою якої функції він викликається.

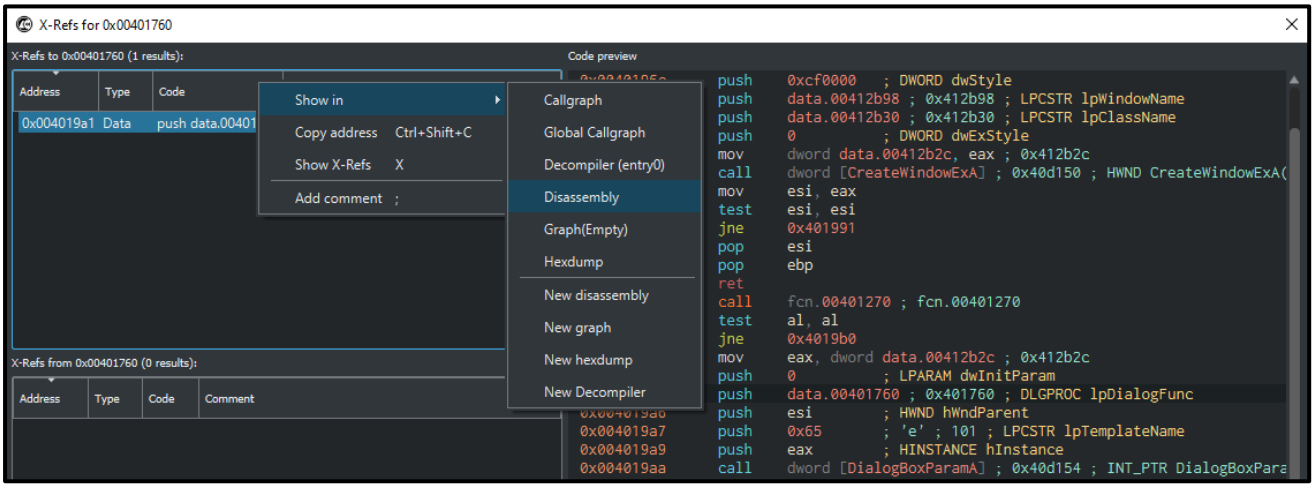


– Простежуємо у асемблерному коді, що запис знаходиться у функціональній секції “**data.00401760**”, а тому необхідно детальніше розглянути логіку її виконання.



```
0x00401914      push    0
0x00401916      push    str.Let_the_SuXX_bi_wiz_U ; 0x40d36c
0x0040191b      push    str.Sorry__Mikel__Buzzinesss: ; 0x40d350
0x00401920      push    0
0x00401922      call    dword [MessageBoxA] ; 0x40d174 ; int MessageBoxA(HWND hWnd, LPCST
```

– Додатково переглянемо, де все ж таки відбувається посилання на цю пам’ять.



– І бачимо, що попередній фрагмент коду, що зображений на фото зверху, міститься у виклику “**fcn.00401950**” функції.

```
fcn.00401950 (HINSTANCE hInstance, int nCmdShow);
; arg HINSTANCE hInstance @ stack + 0x4
; arg int nCmdShow @ stack + 0x8
0x00401950      push    ebp
0x00401951      mov     ebp, esp
```

```
0x00401998      jne     0x4019b0
0x0040199a      mov     eax, dword data.00412b2c ; 0x412b2c
0x0040199f      push    0           ; LPARAM dwInitParam
0x004019a1      push    data.00401760 ; 0x401760 ; DLGPROC lpDialogFunc
0x004019a6      push    esi         ; HWND hWndParent
0x004019a7      push    0x65        ; 'e' ; 101 ; LPCSTR lpTemplateName
0x004019a9      push    eax         ; HINSTANCE hInstance
0x004019aa      call    dword [DialogBoxParamA] ; 0x40d154 ; INT_PTR DialogBoxParamA(HIN
0x004019b0      mov     ecx, dword [nCmdShow]
```

– У свою чергу, вищезгадана функція викликається у головній функції “**main**”

```
int main (int argc, char **argv, char **envp);
; var unknown_t *lpMsg @ stack - 0x20
; var int32_t var_18h @ stack - 0x18
; arg char **argv @ stack + 0x4
; arg int32_t arg_10h @ stack + 0x10
0x004019d0      push    ebp
0x004019d1      mov     ebp, esp
```

```
0x00401a02      push    eax         ; int nCmdShow
0x00401a03      push    esi         ; HINSTANCE hInstance
0x00401a04      call    fcn.00401950 ; fcn.00401950
```

– Отже, тепер ми розуміємо, як саме буде викликана відповідна секція пам’яті, а тому простежуємо, як чином програма вирішує, чи вірно був введений пароль.

```
;-- data.00401760:
0x00401760      push    ebp
0x00401761      mov     ebp, esp
0x00401763      push    ecx
0x00401764      push    esi
0x00401765      push    edi
0x00401766      push    0x400       ; 1024
0x0040176b      call    fcn.00401b09 ; fcn.00401b09
```

.....

```
0x004017df      call    fcn.00401200 ; fcn.00401200
0x004017e4      add     esp, 0xc
0x004017e7      test    al, al
0x004017e9      je      0x401914
0x004017ef      push    0x40d3b8
0x004017f4      push    str.reg.reg ; 0x40d258
0x004017f9      call    flirt.gets_s ; flirt.gets_s
0x004017fe      mov     ebx, eax
0x00401800      add     esp, 8
0x00401803      test    ebx, ebx
0x00401805      je      0x4018d4
```

.....

```
0x00401914      push    0
0x00401916      push    str.Let_the_SuXX_bi_wiz_U ; 0x40d36c
0x0040191b      push    str.Sorry__Mikel__Buzzinesss: ; 0x40d350
0x00401920      push    0
0x00401922      call    dword [MessageBoxA] ; 0x40d174 ; int MessageBoxA(HWND hWnd, LPC
```

– Із наведених вище фото, можна зрозуміти, що після виклику функції “**`fcn.00401200`**”, буде зрозуміло, чи правильно були внесені облікові дані, так як далі буде виконуватися в “стрибок”, якщо вони були неправильними.

Decompiler (fcn.00401200) (unsynced)

```
/* jsdec pseudo code output */
/* D:\KPI\SoftwareProtection\Crack_me_up!.exe @ 0x401200 */
#include <stdint.h>

uint32_t fcn_00401200 (int32_t arg_4h, int32_t arg_8h, int32_t arg_ch) {
    int32_t var_8h;
    eax = fcn_00401b09 (8);
    edx = arg_4h;
    esi = eax;
    eax = arg_ch;
    fcn_00401160 (edx, var_8h, eax);
    eax = var_8h;
    fcn_004011b0 (eax, esi);
    eax = arg_8h;
    edi = esi;
    ecx = 0;
    edi -= eax;
    do {
        dl = *((edi + eax));
        if (dl != *(eax)) {
            goto label_0;
        }
        ecx++;
        eax++;
    } while (ecx < 8);
```

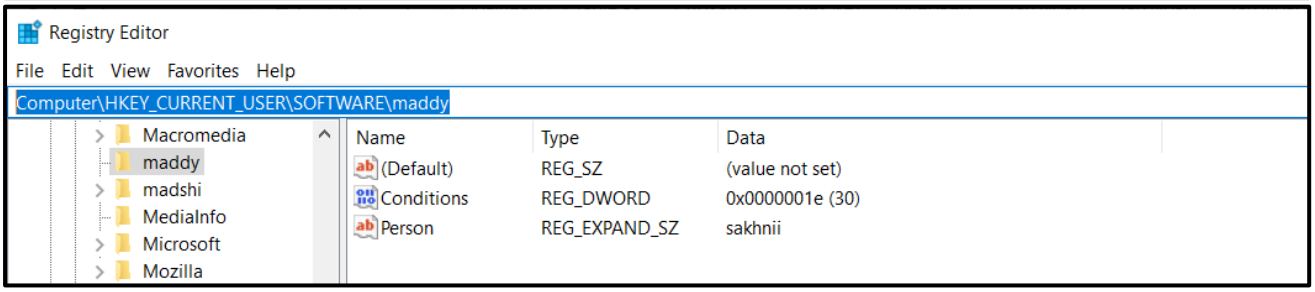
```
    fcn_00401aaf ();
    al = 1;
    return eax;
label_0:
    al = fcn_00401aaf ();
    al = 0;
    return eax;
}
```

/* На фото було продемонстровано дану функції в декомпіляторі */

- Визначимо файл або файли, в яких зберігається зашифрований пароль:

– Можна помітити, що після вдалої авторизації, будуть внесені відповідні записи до регістру із такою самою назвою, що й авторизований користувач, у папку, посилання якої “**Computer\HKEY_CURRENT_USER\SOFTWARE\maddy**”.

```
0x00401816      push    str.Software_maddy ; 0x40d340
0x0040181b      push    0x80000001
0x00401820      call    dword [RegOpenKeyExA] ; 0x40d010 ; LSTATUS RegOpenKeyExA(HKEY hKey, LPCS
0x00401826      test    eax, eax
0x00401828      jne     0x401853
0x0040182a      mov     eax, edi
0x0040182c      lea     edx, [eax + 1]
0x0040182f      nop
0x00401830      mov     cl, byte [eax]
0x00401832      inc     eax
0x00401833      test    cl, cl
0x00401835      jne     0x401830
0x00401837      mov     ecx, dword [ebp - 4]
0x0040183a      sub     eax, edx
0x0040183c      inc     eax
0x0040183d      push    eax
0x0040183e      push    edi
0x0040183f      push    2           ; 2
0x00401841      push    0
0x00401843      push    str.Person ; 0x40d2d8
0x00401848      push    ecx
0x00401849      call    dword [RegSetValueExA] ; 0x40d004 ; LSTATUS RegSetValueExA(HKEY hKey, LP
```

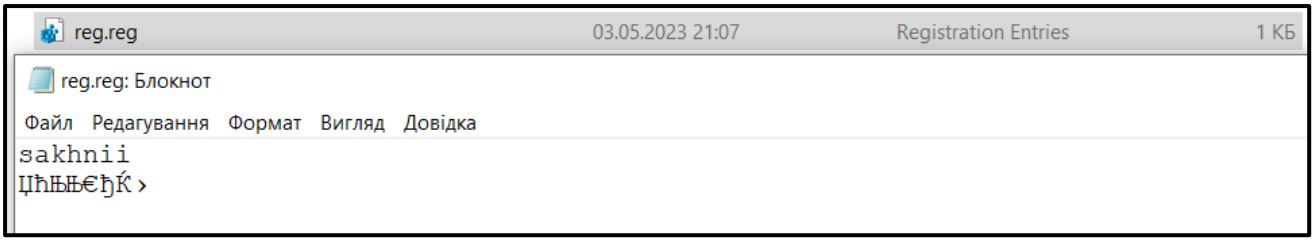


– Дізнаємося, що у файлі з ім’ям “**reg.reg**” зберігається зашифрований пароль.

```
0x004017f4      push    str.reg.reg ; 0x40d258
```

……

```
0x0040155b      push    data.00412320 ; 0x412320 ; int32_t arg_4h
0x00401560      push    str.s__s    ; 0x40d250 ; int32_t arg_8h
0x00401565      push    esi         ; int32_t arg_ch
0x00401566      call    flirt.fwscanf ; flirt.fwscanf
0x0040156b      push    esi         ; int32_t arg_8h
0x0040156c      call    flirt.fclose ; flirt.fclose
```



2. ----------

- Здійснимо блокування встановленого засобу захисту, реалізувавши відключення захисного механізму, шляхом модифікації функції прийняття рішення про коректність введеного пароля:

– Щоби програма не виконувала перевірка на правильність введених облікових даних, необхідно і достатньо зробити зміни в асемблерному коді за адресою "**0x004017e9**", а саме зробити зворотнім "стрибок" (**`jn → jne`**).

3. ----------

- Виділимо в програмі ділянку коду, відповідальну за формування коректного пароля, відповідно введеному імені користувача:

– Помічаємо в програмному коді виклик WinAPI-функції “**GetDlgItemTextA**”, а отже, робимо висновок, що отримані нею параметри (у нашому випадку логін, пароль та довжина логіну) будуть передані до функції “**fcn.00401200**”.

```
0x004017a9      mov     ebx, dword [GetDlgItemTextA] ; 0x40d160
0x004017af      push    0x400       ; 1024
0x004017b4      push    edi
0x004017b5      push    0xc8        ; 200
0x004017ba      push    ecx
0x004017bb      call    ebx
0x004017bd      mov     edx, dword [ebp + 8]
0x004017c0      push    0x400       ; 1024
0x004017c5      push    esi
0x004017c6      push    0xc9        ; 201
0x004017cb      push    edx
0x004017cc      call    ebx
0x004017ce      mov     eax, edi
0x004017d0      lea     edx, [eax + 1]
0x004017d3      mov     cl, byte [eax]
0x004017d5      inc     eax
0x004017d6      test    cl, cl
0x004017d8      jne     0x4017d3
0x004017da      sub     eax, edx
0x004017dc      push    eax
0x004017dd      push    esi
0x004017de      push    edi
0x004017df      call    fcn.00401200 ; fcn.00401200 ;  fcn.00401200(int32_t arg_4h, int32_t arg_ch, int32_t arg_8h)
```

– Далі переглядаємо дану функцію в декомпіляторі та пробуємо знайти наступні виклики функцій, які необхідні для формування пароля.

```
/* jsdec pseudo code output */
/* D:\KPI\SoftwareProtection\Crack_me_up!.exe @ 0x401208 */
#include <stdint.h>

uint32_t fcn_00401200 (int32_t arg_4h, int32_t arg_8h, int32_t arg_ch) {
    int32_t var_8h;
    /* A function of allocating some buffer  */
    eax = fcn_00401b09 (8);
    edx = arg_4h;
    esi = eax;
    eax = arg_ch;
    /* The first part of the password generation function */
    fcn_00401160 (edx, var_8h, eax);
    eax = var_8h;
    /* The second part of the password generation function */
    fcn_004011b0 (eax, esi);
    eax = arg_8h;
    edi = esi;
    ecx = 0;
    edi -= eax;
    do {
        dl = *((edi + eax));
        if (dl != *(eax)) {
            goto label_0;
        }
        ecx++;
        eax++;
    } while (ecx < 8);
    /* It's an auxiliary function */
    fcn_00401aaf ();
    al = 1;
    return eax;
label_0:
    /* The same auxiliary function */
    al = fcn_00401aaf ();
    al = 0;
    return eax;
}
```

– Отже, першою функціональною частиною генерації відповідного пароля є функція “**fcn_00401160**”, що в якості аргументів приймає логін, деяку змінну та довжину логіну, а сам результат її виконання заносить до регістру **eax**.

```
#include <stdint.h>

uint32_t rotate_left32 (uint32_t value, uint32_t count) {
    const uint32_t mask = (CHAR_BIT * sizeof (value)) - 1;
    count &= mask;
    return (value << count) | (value >> (-count & mask));
}

int32_t fcn_00401160 (int32_t arg_4h, int32_t arg_8h, int32_t arg_ch) {
    int32_t var_8h;
    eax = arg_8h;
    esi = arg_ch;
    edx = 0;
    *(eax) = 0xfacc0fff;
    var_8h = edx;
    if (esi <= 0) {
        goto label_0;
    }
    while (edx < esi) {
        eax = arg_4h;
        eax += var_8h;
        ecx = 0;
        cl = *(eax);
        eax = arg_8h;
        ebx = *(eax);
        ecx ^= ebx;
        ecx = rotate_left32 (ecx, 8);
        eax = arg_8h;
        *(eax) = ecx;
        ecx = ebx;
        eax = ebx;
        edx++;
        var_8h = edx;
    }
label_0:
    esi = ebx;
    return eax;
}
```

– Другою ж частиною виступає функція “**fcn_004011b0**”, яка на вхід приймає ту саму змінну, що вже пройшла відповідні перетворення у першій частині, та попередньо створений буфер, у який буде записаний вихідний результат (пароль), що у свою чергу повернеться назад у якості значення регістра **eax**.

```
/* jsdec pseudo code output */
/* D:\KPI\SoftwareProtection\Crack_me_up!.exe @ 0x4011b0 */
#include <stdint.h>

int32_t fcn_004011b0 (int32_t arg_4h, int32_t arg_8h) {
    int32_t var_8h;
    ebx = arg_8h;
    eax = 0;
    edi = arg_4h;
    var_8h = ax;
    esi = 0;
    do {
        eax = edi;
        eax &= 0xf;
        edi >>= 4;
        if (eax > 9) {
            eax = 9;
        }
        fcn_0040c06b (eax, var_8h, 0xa);
        dl = var_8h;
        *((esi + ebx)) = dl;
        esi++;
    } while (esi < 8);
    return eax;
}
```

- Дослідимо даний код і формально запишемо алгоритм формування коректного пароля:

❖ Для кожного символу введеного логіну необхідно виконати наступні операції:
- ➢ Застосувати операцію “**XOR**” між ASCII-кодом даного символу та останнім байтом відповідного числа (тут для першого кроку береться шістнадцяткове значення (0xfacc0fff);
- ➢ Виконати обертання вліво на 1 байт (т.б на 8 біт) для отриманого результату із попереднього кроку.

❖ Отримане число необхідно передати в наступну функцію, яка також для кожної цифри цього числа, починаючи з кінця, виконуватиме наступне:
- ➢ Братиме останню цифру числа за допомогою порозрядного оператора “**AND**” (x & 0x0f), що перетворить її у шістнадцятковий формат;
- ➢ Зсуватиме значення даного числа праворуч на 4 біти;
- ➢ Виконуватиме порівняння тої останньої цифри із 9, і якщо вона більше, то додасть до пароля 9, а якщо менше, то додасть цю ж саму цифру.

- Використовуючи код програми, що відповідає за формування правильного пароля, створимо генератор паролів:

Run | Debug | Stop | Share | Save | { } Beautify

main.py

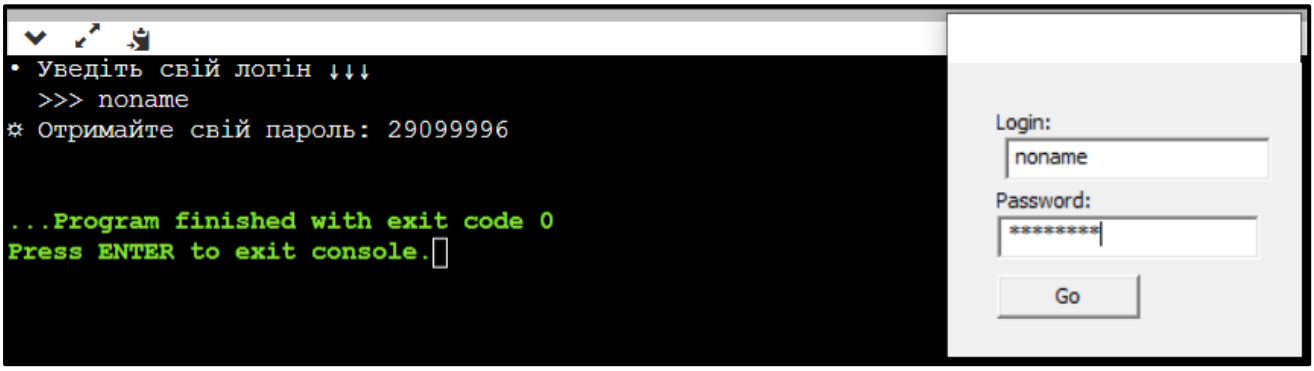
```
1  login = input("• Уведіть свій логін ↓↓↓  \n  >>> ")
2
3  num = 0xfacc0fff
4  for i in login:
5      num = num ^ ord(i)
6      num = ((num << 8) & 0xffffff00) | (num >> 24)
7
8  password = ""
9  for i in range(8):
10     last = num & 0x0f
11     num = num >> 4
12     if last > 9:
13         last = 9
14     password += str(last)
15
16 print(f"☼ Отримайте свій пароль: {password}")
```

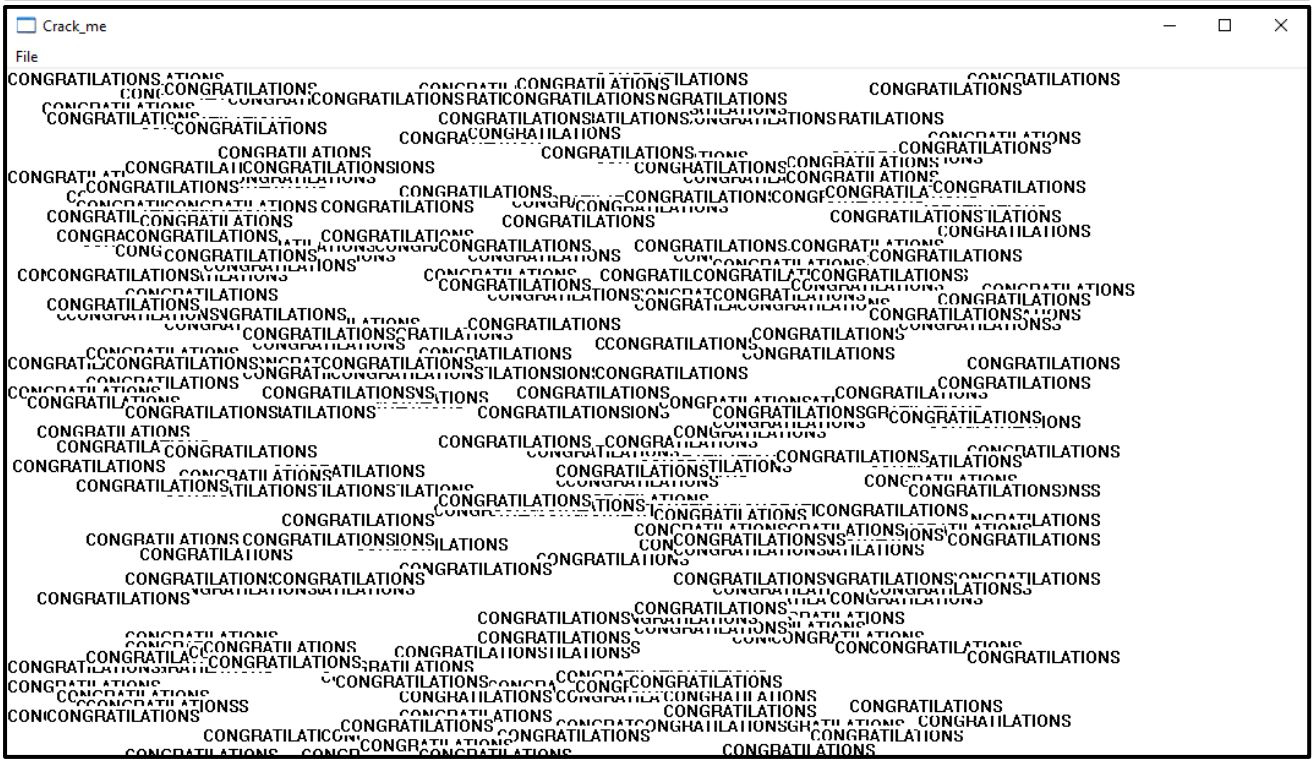
input

.....

4. ----------

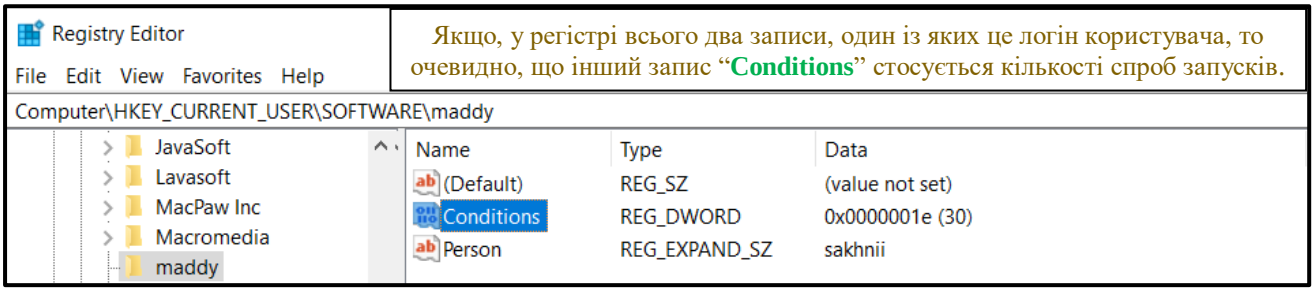
- Здійснимо злом встановленого засобу захисту, використовуючи деякий користувальницький ідентифікатор (ім'я користувача) і відповідний йому коректний пароль, сформований по знайденому в п.3 алгоритму:





5. ----------

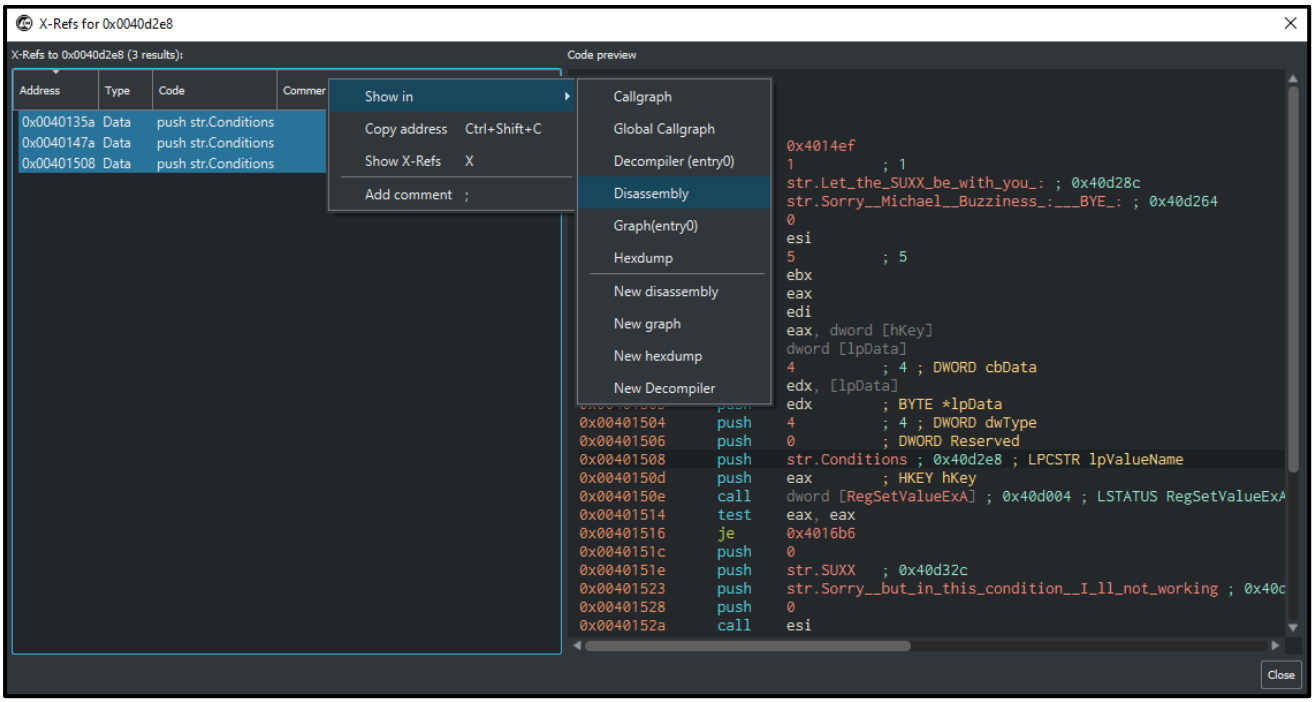
- Визначимо ключі реєстру, у яких зберігається лічильник спроб запусків:



Якщо, у регістрі всього два записи, один із яких це логін користувача, то очевидно, що інший запис "**Conditions**" стосується кількості спроб запусків.

- Виділимо в досліджуваній програмі ділянку коду, що виконує функцію прийняття рішення про перевищення встановленої межі спроб запусків:

– Як бачимо, рядок “**Conditions**” згадується декілька разів у програмному коді.



– Однак, якщо детальніше подивитися на яке місце в асемблерному коді посилається даний запис, то можна зрозуміти, що використовується він лише у функції “**fcn.00401270**”, яка у свою чергу викликається із “**fcn_00401950**”.

```
Decompiler (fcn.00401270)
/* jsdec pseudo code output */
/* D:\KPI\SoftwareProtection\Crack_me_up!.exe @ 0x401270 */
#include <stdint.h>

int32_t fcn_00401270 (void) {
    int32_t var_81ch;
    LPDWORD lpdwDisposition;
```

.....

```
        ecx = hKey;
        eax = RegCreateKeyExA (ecx, "maddy", 0, 0, 0, 0xf003f, 0, phkResult, lpdwDisposition);
        if (eax != 0) {
            void (*esi)(uint32_t, char*, char*, uint32_t) (0, "Sorry, but in this condition, I'll not working", "SUXX", 0);
            eax = void (*ebx)(uint32_t) (1);
            void (*edi)(uint32_t) (eax);
        }
        eax = phkResult;
        eax = RegSetValueExA (eax, "Conditions", 0, 4, lpData, 4, 0x1e);
        if (eax != 0) {
            void (*esi)(uint32_t, char*, char*, uint32_t) (0, "Sorry, but in this condition, I'll not working", "SUXX", 0);
            eax = void (*ebx)(uint32_t) (3);
            void (*edi)(uint32_t) (eax);
        }
        ecx = phkResult;
        eax = RegSetValueExA (ecx, "Person", 0, 2, "noname", 8);
        if (eax != 0) {
            void (*esi)(uint32_t, char*, char*, uint32_t) (0, "Sorry, but in this condition, I'll not working", "SUXX", 0);
            eax = void (*ebx)(uint32_t) (3);
            void (*edi)(uint32_t) (eax);
        }
        edx = phkResult;
```

.....

```
    if (eax == 0) {
        ecx = hKey;
        eax = void (*esi)(uint32_t, char*, uint32_t, uint32_t, uint32_t, uint32_t) (ecx, "Conditions", 0, var_81ch, lpData, lpdwDisposition);
        esi = imp.MessageBoxA;
        edi = imp.TerminateProcess;
        ebx = imp.GetCurrentProcess;
        if (eax != 0) {
            void (*esi)(uint32_t, char*, char*, uint32_t) (0, "Sorry, but in this condition, I'll not working", "SUXX", 0);
            eax = void (*ebx)(uint32_t) (3);
            void (*edi)(uint32_t) (eax);
        }
        if (lpData > 0x1e) {
            void (*esi)(uint32_t, char*, char*, uint32_t) (0, "Is there anybody hacking me?", "I'm confused", 0);
            eax = void (*ebx)(uint32_t) (3);
            void (*edi)(uint32_t) (eax);
        }
        if (lpData == 0) {
            void (*esi)(uint32_t, char*, char*, uint32_t) (0, "Sorry, Michael, Buzziness :), BYE :)", "Let the SUXX be with you :)", 1);
            eax = void (*ebx)(uint32_t) (5);
            void (*edi)(uint32_t) (eax);
        }
```

```
        eax = hKey;
        lpData--;
        eax = RegSetValueExA (eax, "Conditions", 0, 4, lpData, 4);
        if (eax == 0) {
            goto label_0;
        }
        void (*esi)(uint32_t, char*, char*, uint32_t) (0, "Sorry, but in this condition, I'll not working", "SUXX", 0);
        eax = void (*ebx)(uint32_t) (3);
        void (*edi)(uint32_t) (eax);
        goto label_0;
    }
```

6. ----------

- Здійснимо блокування встановленого засобу захисту підміною функції прийняття рішення про перевищення встановленої межі спроб запусків:

– Як можна було вже помітити, змінна “**lpData**” відповідає за контроль кількості спроб запуску, тобто її значення зменшується та порівнюється при кожному відкритті програми, а отже треба буде щось з ним зробити :)

```
        eax = hKey;
        lpData--;
        eax = RegSetValueExA (eax, "Conditions", 0, 4, lpData, 4);
```

– Отже, для блокування встановленого засобу захисту необхідно змінити рядок наступним чином: {**dec dword [lpData]**} → {**nop**}

```
0x004014ef      mov     eax, dword [hKey]
0x004014f5      dec     dword [lpData]
```

↓ ↓

```
0x004014ef      mov     eax, dword [hKey]
0x004014f5      nop
```

– Звідси маємо, що через те, що немає змінної, у якій би зберігалось значення кількості спроб запусків програми, то значення регістру змінюватись не буде.

```
        eax = hKey;
        eax = RegSetValueExA (eax, "Conditions", 0, 4, lpData, 4);
```