PAC 1. Debugging i reversing d'una aplicació

Sílvia Sanvicente García

10 octubre 2021

Índex

Reverse engineering **2**
Aplicació escollida i eines emprades 2
Procés detallat de la realització del reversing 2

Preguntes **4**
Què heu deduït què fa l'aplicació a partir del reversing realitzat? 4
Utilitza constants, literals? 4
Quins bucles hi ha en l'aplicació? 5
Quins mètodes o funcions té l'aplicació? 6
Heu detectat algun punt vulnerable? 6
Utilitza alguna funció vulnerable? 6
Quins condicionals has trobat? 7
Pots detectar quines variables fa servir? 7
Mostra l'ús de la Pila en algun punt de l'execució 7
Es pot alterar el flux del codi? 7

Conclusions **8**

Referències **8**

Apèndix **9**

Reverse engineering

Aplicació escollida i eines emprades

Per dur a terme aquesta PAC i fer el reversing s'ha utilitzat l'executable "program.exe" facilitat per la docència d'aquesta assignatura juntament amb l'eina de debugging Ghidra. S'ha escollit aquesta eina en comptes d'altres com Immunity o IDA Pro atès que Ghidra és una eina lliure i de codi obert, desenvolupada per l'Agència de Seguretat Nacional (NSA).

Procés detallat de la realització del reversing

Per analitzar l'aplicació a la qual es vol fer reversing, en primer lloc s'ha executat per veure el seu funcionament. Atès que aquesta aplicació podria contenir codi maliciós s'ha creat una màquina virtual amb un sistema operatiu Windows 10 per executar-la.

Tal com es veu a la figura 1, aquesta aplicació mostra una capçalera amb informació i demana inserir un codi d'activació. L'aplicació permet tres intents abans de tancar-se automàticament.

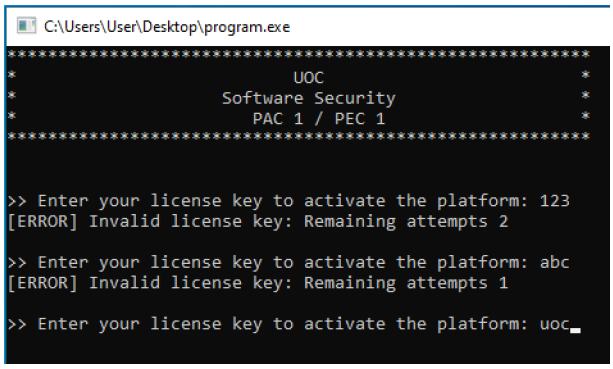

Figura 1: Captura de l'execució de l'aplicació

A continuació s'ha creat un nou projecte de Ghidra per analitzar l'aplicació. La figura 2 mostra la informació que facilita Ghidra sobre l'executable. Aquest té una arquitectura de 32 bits amb format Little Endian i ha estat compilat amb un sistema operatiu Windows.

Figura 2: Informació que facilita Ghidra sobre l'executable

Després d'analitzar el binari, a Symbol Tree es mostren les funcions i es poden filtrar buscant la funció *main*, la qual es mostra a la figura 3.

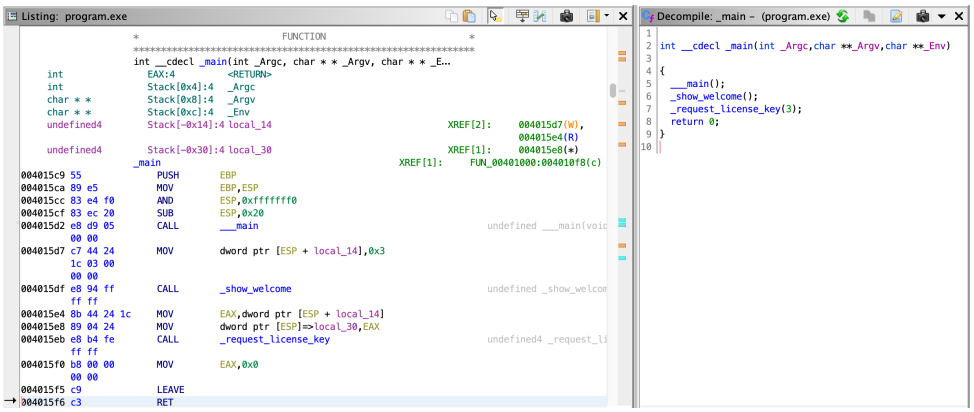

Figura 3: Funció *main* de l'executable

La funció interessant és *request_license_key*, ja que *show_welcome* simplement mostra el text de la capçalera. La figura 4 mostra el codi en llenguatge assemblador de la funció *request_license_key* i el codi descompilat. Aquesta funció té un paràmetre el qual indica el nombre d'intents que té l'usuari per inserir la clau. Dins de la funció s'inicialitzen diverses variables i hi ha un bucle que demana la clau a l'usuari mentre que no arribi al límit marcat pel paràmetre de la funció. Per comprovar si la clau és correcta crida a la funció *is_license_key* i en cas d'encert finalitza el bucle.

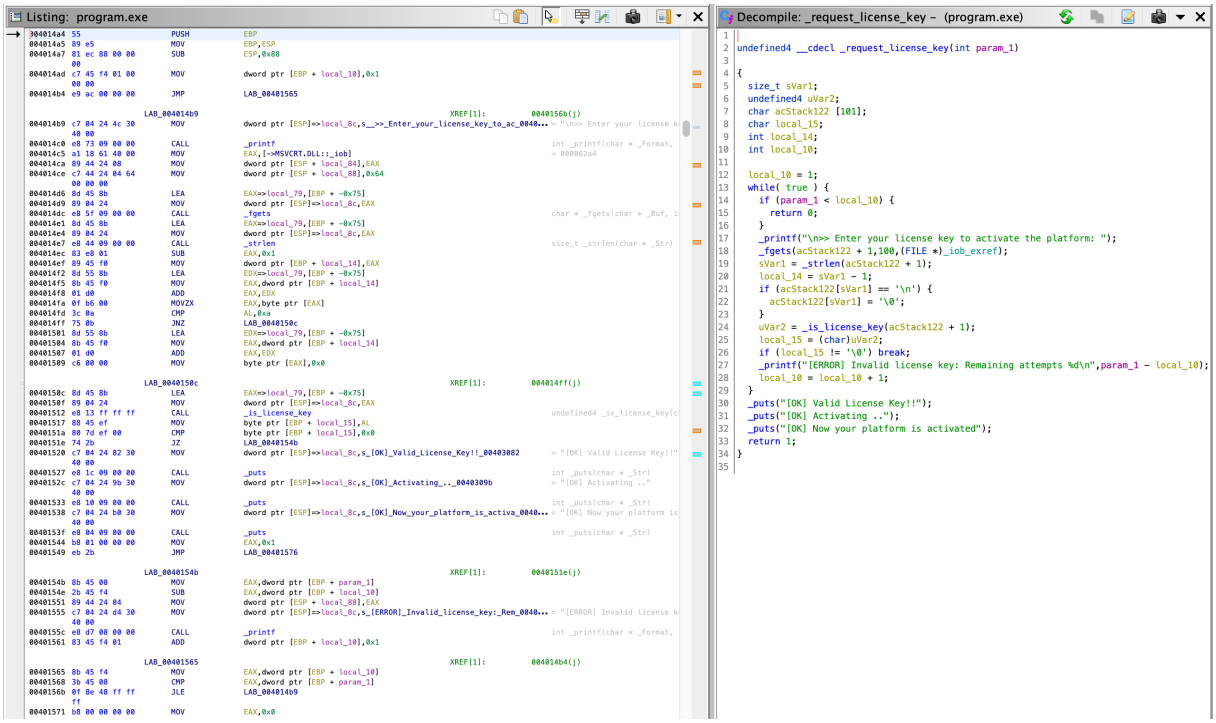

Figura 4: Llenguatge assemblador i codi descompilat de *request_license_key*

La funció *is_license_key* comprova si l'usuari ha inserit la clau correcta. Hi ha dues formar de trencar aquesta verificació. En primer lloc, en la funció *is_master*, es comprova si la clau

inserida per l'usuari és igual a una clau per defecte. Es pot veure aquesta clau observant la llista de strings que retorna l'eina Ghidra, on es mostra que la clau és "35363FC4-8671-4F2C-AE70-4BC9045EC6A3". Si no és la clau per defecte ha de complir els requisits de les funcions *sum_numbers* i *is_divisible_by_a_and_b*. La funció *is_divisible_by_a_and_b* comprova que si fem el mòdul del resultat de *sum_numbers* per 4 (0x4 en hex) i per 11 (0xb en hex) aquest dóna 0 en ambdós casos. Això vol dir que el resultat de *sum_numbers* ha de donar per exemple 44, 88, 132, etc. A partir de la clau per defecte és fàcil modificar algun valor perquè doni un d'aquests resultats, com per exemple la clau "15363FC4-8671-4F2C-AE70-4BC9045EC6A3". Per entendre el funcionament de les funcions i poder fer proves amb diferents claus s'ha escrit el codi que es mostra a la figura 5 a partir del codi obtingut de la descompilació. Les captures del codi en assemblador i descompilat de les altres funcions mencionades es mostra a l'Apèndix.

```cpp
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;

int sum_numbers(char *param_1) {
    char local_38 [36];
    int local_14;
    int local_10;
    strcpy(local_38,param_1);
    local_14 = 0;
    for (local_10 = 0; local_38[local_10] != '\0'; local_10 = local_10 + 1) {
        if (('/' < local_38[local_10]) && (local_38[local_10] < ':')) {
            local_14 = local_14 + local_38[local_10] + -0x30;
        }
    }
    return local_14;
}

int is_divisible_by_a_and_b(int param_1,int param_2,int param_3) {
    return ((param_1 % param_2 == 0) && (param_1 % param_3 == 0)) ? 1 : 0;
}

int main() {
    char key[] = { '1','5','3','6','3','F','C','4','-','8','6','7','1','-','4','F','2','C','-','A','E','7','0','-','4','B','C','9','0','4','5','E','C',
    int var4 = sum_numbers(key);
    int var5 = is_divisible_by_a_and_b(var4,4,11);
    cout << var4 << "\n" << var5;
}
```

```
C++ - key.cpp:18 ✓
88
1
[Finished in 0.738s]
```

Figura 5: Codi per entendre el funcionament de les funcions i poder fer proves

Preguntes

A continuació es procedeix a respondre les preguntes plantejades un cop realitzat el procés de reversing.

Què heu deduït què fa l'aplicació a partir del reversing realitzat?

Aquesta aplicació dóna tres intents a un usuari per inserir una clau per activar la llicència d'una plataforma. Podem deduir això a partir dels strings que mostra l'eina de debugging Ghidra.

Utilitza constants, literals?

Aquest programa si utilitza literals com es pot veure en el codi assemblador de la figura 6, on es guarda el valor 0x1 al registre EAX. Podem veure instruccions similars al llarg de tot el codi assemblador.

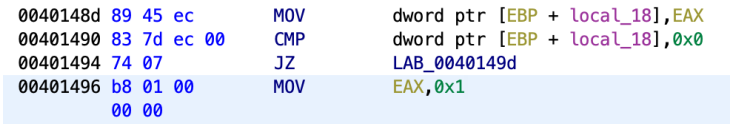

Figura 6: Guarda el valor 0x1 al registre EAX

Quins bucles hi ha en l'aplicació?

En la funció *request_license_key* trobem un bucle while. Es pot determinar que no és un bucle for perquè no hi ha una instrucció que incrementi un registre i no és un bucle do-while perquè es fa el salt directament en comptes de moure una variable inicial a un registre. Aquest bucle demana a l'usuari que insereixi la clau i se surt d'ell en les següents instruccions:

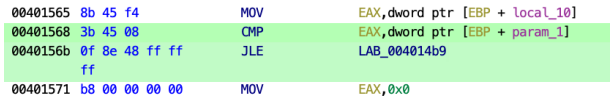

Figura 7: Instruccions per sortir del bucle

A la figura 20, a l'Apèndix, es mostra el codi assemblador d'aquesta funció.

En la funció *sum_numbers* hi ha un bucle de tipus for. Es pot determinar que és de tipus for atès que en el codi assemblador hi ha instruccions que indiquen que hi ha una variable incremental, ja que el registre EBP s'inicialitza a 0 i s'incrementa posteriorment, instruccions que no són necessàries en els altres tipus de bucle. Aquest bucle recorre la clau inserida per l'usuari i suma els elements que la componen, sortint del bucle en trobar l'element "\0".

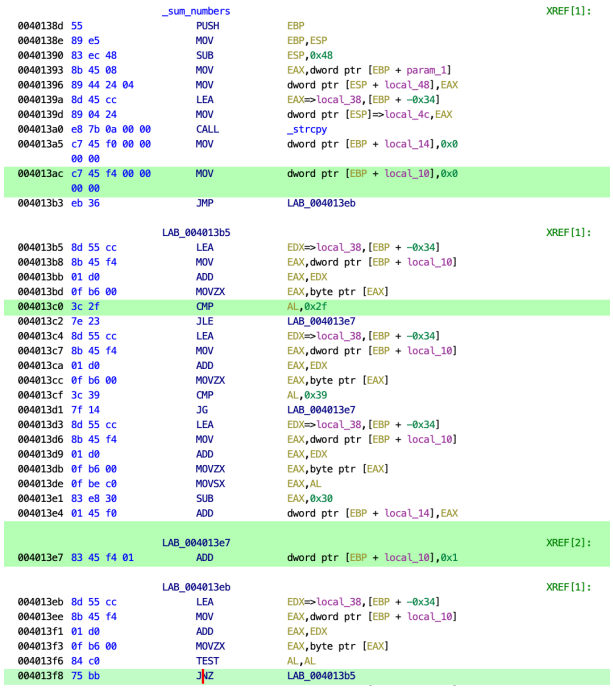

Figura 8: Codi assemblador de la funció *sum_numbers*

Quins mètodes o funcions té l'aplicació?

A la figura 9 es llisten les funcions del programa analitzat a partir de la funció main.

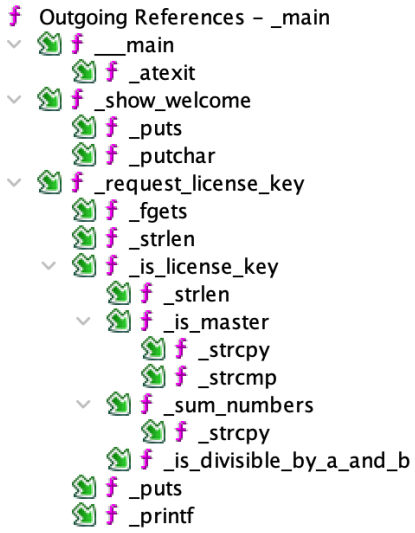

Figura 9: Funcions del programa analitzat

Heu detectat algun punt vulnerable?

Sí. Aquesta aplicació és per a validar la clau l'activació d'una plataforma i fent reversing es troben dues formes d'obtenir aquesta clau. En primer lloc, observant els strings es pot obtenir una clau per defecte i d'altra banda, es poden crear noves claus a partir d'aquesta sempre que compleixin els requisits de les funcions *sum_numbers* i *is_divisible_by_a_and_b*.

Utilitza alguna funció vulnerable?

Sí, en el codi assemblador de la funció *is_master* es pot veure que es crida a la funció *strcpy* i *strcmp*. La funció *strcpy* còpia la cadena apuntada per origen en la cadena apuntada per destinació, però és millor utilitzar la funció *snprintf* per evitar problemes amb el buffer. La funció *strcmp* compara dues cadenes i retorna 0 si són iguals. En aquest cas és millor utilitzar *strncmp*, ja que rep com a paràmetre també la mida de les cadenes.

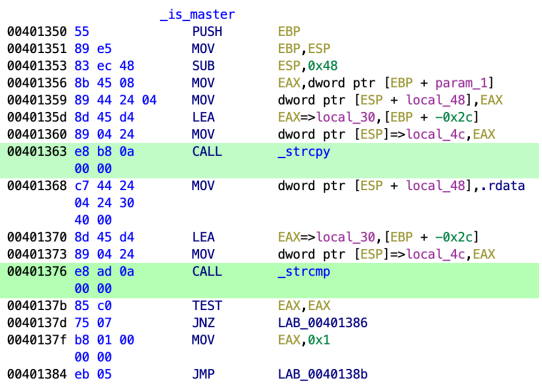

Figura 10: Funcions *strcpy* i *strcmp*

Quins condicionals has trobat?

Al llarg del codi assemblador hi ha diferents condicionals. Aquests es poden detectar perquè hi ha una instrucció de comparació seguida d'una instrucció de salt. A la figura 11 es mostra un fragment de codi assemblador que segurament correspon a un condicional if-else. Es pot determinar el tipus atès que hi ha una instrucció de comparació i una de salt i entre elles un MOV d'una variable en un registre. Aquesta instrucció MOV té lloc si la comparació no és vàlida, equivalent a una instrucció else en C.

```
0040143e 83 7d f4 24        CMP     dword ptr [EBP + local_10],0x24
00401442 74 07              JZ      LAB_0040144b
00401444 b8 00 00 00 00     MOV     EAX,0x0
00401449 eb 57              JMP     LAB_004014a2

                        LAB_0040144b
0040144b 8b 45 08           MOV     EAX,dword ptr [EBP + param_1]
```

Figura 11: Fragment de codi corresponent a un if-else

Pots detectar quines variables fa servir?

Es poden veure les variables que s'estan utilitzant mirant la Stack. Per exemple, en la funció *is_master*, hi ha quatre variables. Dues d'elles són de tipus double word de 4 bytes de longitud, una és de 1 byte i una altra és un punter de tipus char. El punter de tipus char apunta la clau inserida per l'usuari i les altres variables permeten guardar variables temporals de la funció.

```
                        bool __cdecl _is_master(char * param_1)
bool                      AL:1              <RETURN>
char *                    Stack[0x4]:4      param_1                 XREF
undefined1                Stack[-0x30]:1    local_30                XREF
undefined4                Stack[-0x48]:4    local_48                XREF
undefined4                Stack[-0x4c]:4    local_4c                XREF
                        .text                                       XREF[1]:
                        _is_master
00401350 55                 PUSH    EBP
```

Figura 12: Variables en la funció *is_master*

Mostra l'ús de la Pila en algun punt de l'execució

Per visualitzar la pila durant l'execució del programa s'ha utilitzat l'eina OllyDbg. A la figura 13 a l'esquerra, es mostra la pila quan s'executa el programa i demana a l'usuari que insereixi una clau. A la dreta de la figura es mostra la pila quan l'usuari ha inserit la clau.

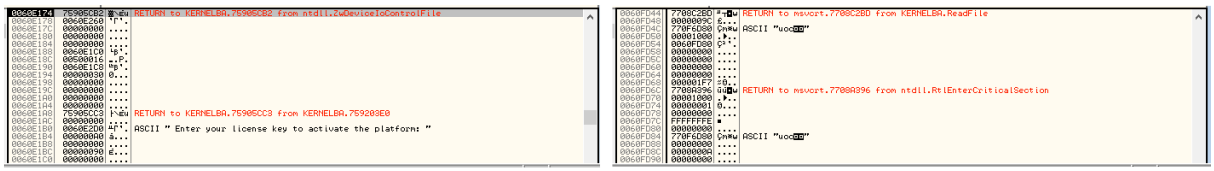

Figura 13: Fragment de la pila durant l'execució del programa

Es pot alterar el flux del codi?

Ghidra permet modificar el flux del programa. La modificació que s'ha fet és no acceptar la clau per defecte com a vàlida, retornant sempre false com a retorn de la funció. A la figura 14 es mostra la modificació del codi assemblador.

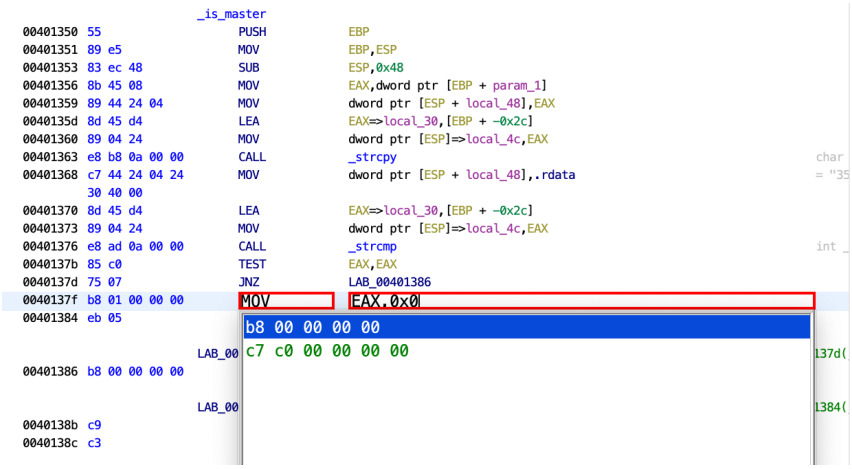

Figura 14: Modificació del codi assemblador

Després d'exportar el codi modificat, en executar no accepta la clau per defecte, tal com es pot veure a la figura 15.

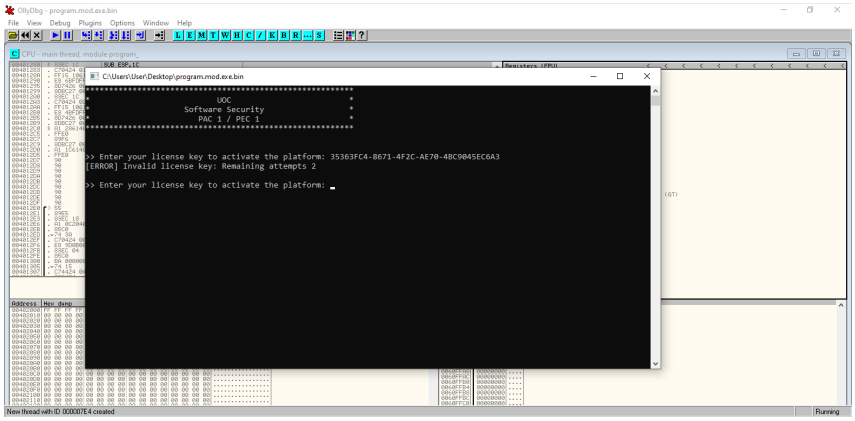

Figura 15: En executar no accepta la clau per defecte

Conclusions

El reverse engineering permet analitzar l'executable d'un programa i comprovar que no té vulnerabilitats que un atacant pugui explotar simplement disposant de l'arxiu executable. En l'executable utilitzat per a aquesta PAC s'ha pogut observar que estudiant el codi assemblador era possible descobrir una clau per defecte per activar una plataforma i que era possible crear noves claus a partir d'aquesta. A part d'aquesta vulnerabilitat, l'executable utilitza funcions vulnerables com *strcpy* i *strcmp*. Amb eines com Ghidra és possible modificar una part del codi assemblador i tornar a exportar el programa, demostrant la importància de signar els programes amb algun algoritme criptogràfic com SHA-256 per comprovar que el programa és oficial i no ha estat modificat.

Referències

[1] Josep Vañó Chic, Herramientas de pentesting de software (setembre 2021).

[2] Josep Vañó Chic, Reverse engineering (setembre 2021).

[3] Josep Vañó Chic, Exploits (setembre 2014). Disponible a: https://materials.campus.uoc.edu/daisy/Materials/PID_00217402/pdf/PID_00217345.pdf

[4] Josep Vañó Chic, Codi segur (setembre 2014). Disponible a: https://materials.campus.uoc.edu/daisy/Materials/PID_00217403/pdf/PID_00217346.pdf

[5] Miquel Albert Orenga i Gerard Enrique Manonellas, Programación en ensamblador (x86-64) (setembre 2011). Disponible a: http://openaccess.uoc.edu/webapps/o2/bitstream/10609/12743/12/Estructura%20de%20computadores_M%C3%B3dulo6_Programaci%C3%B3n%20en%20ensamblador%28x86-64%29.pdf

[6] "Ghidra Software Reverse Engineering Framework". 2021. Ghidra. https://ghidra-sre.org/

[7] "Ghidra quickstart & tutorial: Solving a simple crackme - stacksmashing". 2021. YouTube. https://www.youtube.com/watch?v=fTGTnrgjuGA

[8] "OllyDbg v1.10". 2021. OllyDbg. https://www.ollydbg.de/

Apèndix

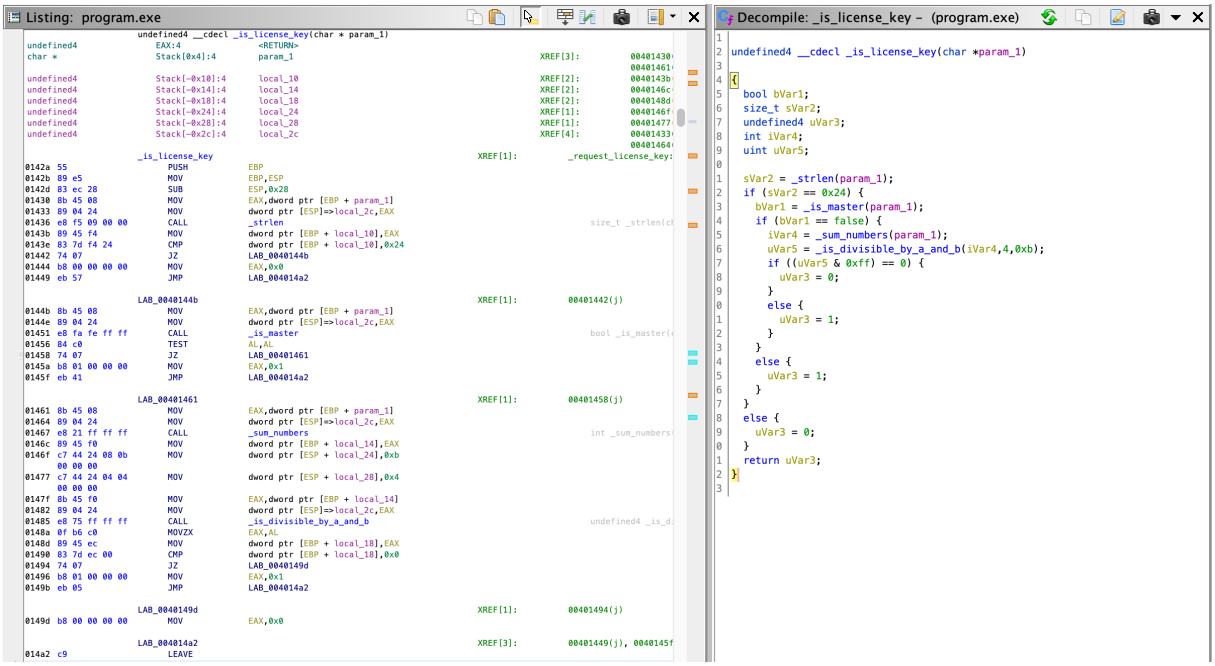

Figura 16: Llenguatge assemblador i codi descompilat de *is_license_key*

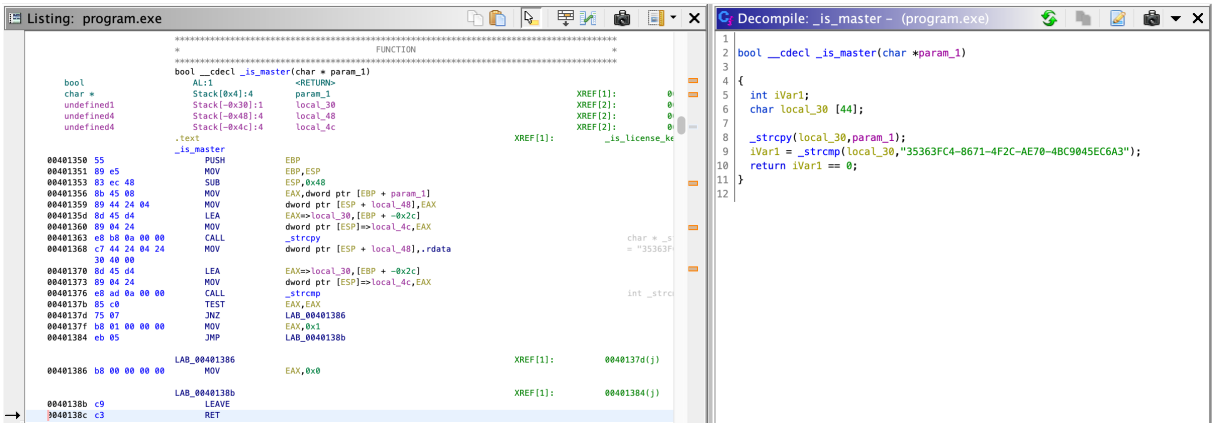

Figura 17: Llenguatge assemblador i codi descompilat de *is_master*

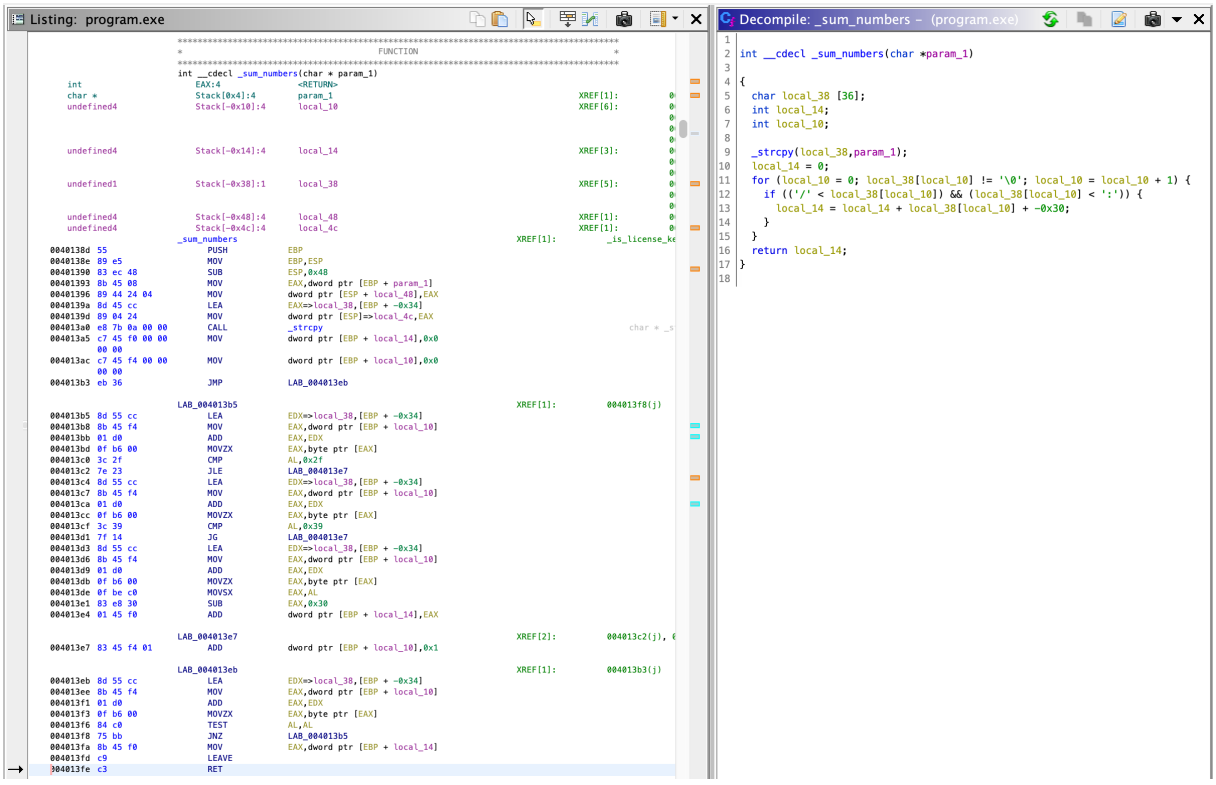

Figura 18: Llenguatge assemblador i codi descompilat de *sum_numbers*

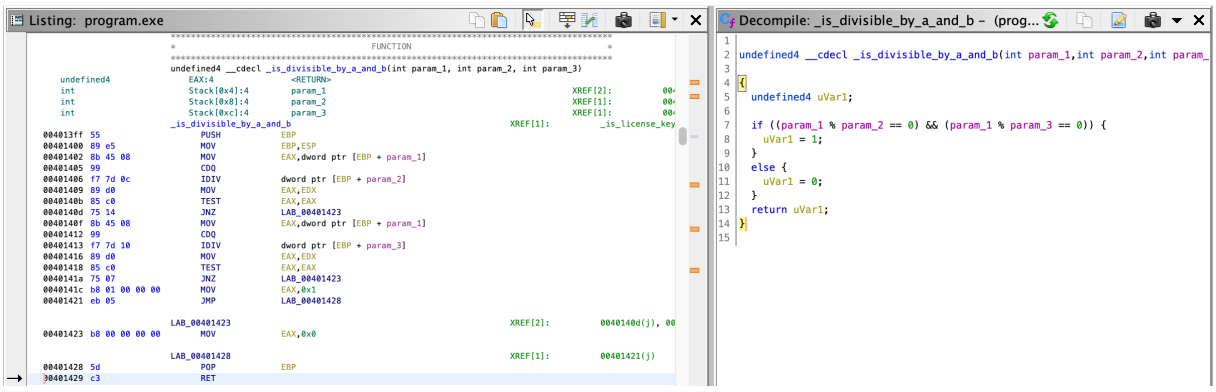

Figura 19: Llenguatge assemblador i codi descompilat de *is_divisible_by_a_and_b*

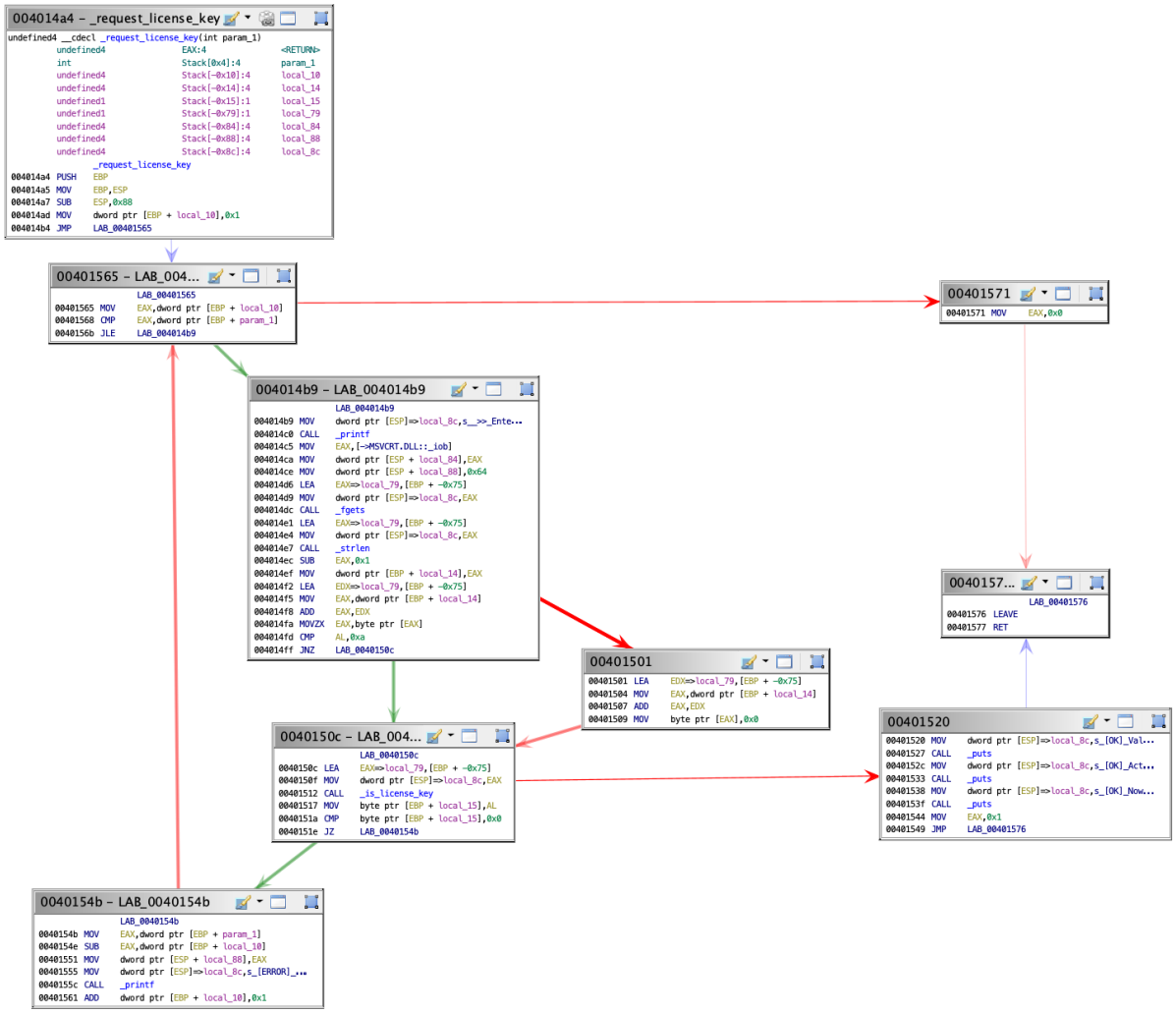

Figura 20: Codi assemblador de la funció *request_license_key*