
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.

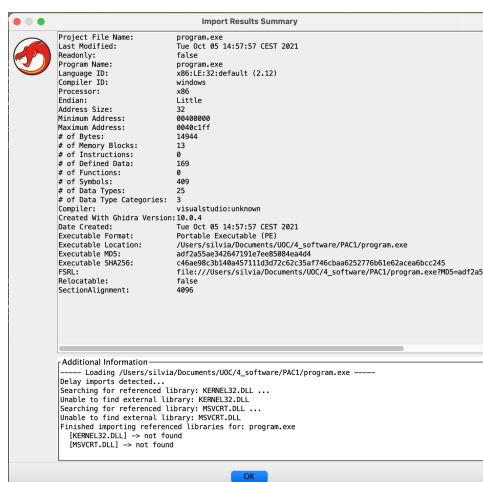


Figura 2: Informació que facilita Ghidra sobre l'executable

Listing: program.exe

```

                *                *
                FUNCTION
*****
int __cdecl _main(int _Argc, char ** _Argv, char ** _Env...)
{
    int     EAX:4   <RETURN>
    int     Stack[0x4]:4  _Argc
    char *   Stack[0x8]:4  _Argv
    char *   Stack[0xc]:4  _Env
    undefined4 Stack[-0x14]:4 local_14
    undefined4 Stack[-0x38]:4 local_38

    _main
004015c9 55      PUSH    EBP
004015ca 89 e5    MOV     EBP,ESP
004015cc 83 e4 10 AND     ECX,0xfffffff0
004015cf 83 ec 20 SUB     ESP,0x20
004015d2 e8 d9 05 CALL    _main
004015d7 c7 44 24 MOV     dword ptr [ESP + local_14],0x3
004015df ff ff    CALL    _show_welcome
004015e4 8b 44 24 MOV     EAX,dword ptr [ESP + local_14]
004015e8 89 24 24 MOV     dword ptr [ESP+local_38],EAX
004015eb eb b4 fe CALL    _request_license_key
004015f0 b8 00 00 MOV     EAX,0x0
004015f5 c9      LEAVE
004015fe c3      RETN
}

```

Decompile: _main - (program.exe)

```

1  int __cdecl _main(int _Argc, char **_Argv, char **_Env)
2  {
3
4      __main();
5      _show_welcome();
6      _request_license_key(3);
7      return 0;
8  }
9
10

```

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 ensamblador 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.

The image displays a Windows desktop with two application windows open side-by-side.

The left window, titled "Listing: program.exe", shows a list of assembly instructions for the "request_license_key" function. The instructions are numbered from 00401444 to 00401561. Key instructions include:

- 00401444: PUSH ESP
- 00401445: MOV EIP, ESP
- 00401447: SUB ESP, 0x08
- 0040144d: MOV dword ptr [EBP + local_10], 0x1
- 00401454: JMP LAB_00401565
- 00401459: MOV dword ptr [ESP+local_8c,s_>_Enter_your_license_key_to_ac_0040...], "\n>> Enter your license key to activate the platform: "
- 00401460: CALL _printf
- 00401465: JZ 0x16
- 0040146a: MOV dword ptr [EIP + local_80], EAX
- 0040146e: MOV dword ptr [ESP + local_80], 0x04
- 0040146e: JZ 0x16
- 0040146e: LEA EAX, local_79, [EBP + -0x75]
- 00401469: MOV dword ptr [ESP+local_8c,EAX]
- 0040146e: CALL _fgets
- 0040146e: LEA EAX, local_79, [EBP + -0x75]
- 00401469: MOV dword ptr [ESP+local_8c,EAX]
- 0040146e: CALL _strlen
- 00401469: MOV EAX, 0x1
- 00401469: MOV dword ptr [EIP + local_14], EAX
- 00401469: MOV EAX, local_79, [EBP + -0x75]
- 00401475: MOV EAX, dword ptr [EBP + local_14]
- 00401478: MOV EAX, EAX
- 0040147a: MOVZX EAX, byte ptr [EAX]
- 0040147d: CMP Al, 0xa
- 0040147f: JNZ LAB_0040159c
- 00401481: LEA EAX, local_79, [EBP + -0x75]
- 00401484: MOV EAX, dword ptr [EBP + local_14]
- 00401487: ADD EAX, EAX
- 00401489: MOV byte ptr [EAX], 0x0
- 00401489: JMP LAB_0040159c
- 0040149c: LEA EAX, local_79, [EBP + -0x75]
- 0040149f: MOV dword ptr [ESP+local_8c,EAX]
- 004014a2: CALL _is_license_key
- 004014a5: MOV EAX, local_15, [EBP + local_15]
- 004014a8: CMP byte ptr [EIP + local_13], 0x0
- 004014ab: JZ LAB_0040154b
- 004014b0: MOV dword ptr [ESP+local_8c,s_[OK]_Valid_license_Key!], 00403882
- 004014b3: CALL _puts
- 004014b6: MOV dword ptr [ESP+local_8c,s_[OK]_Activating...], 0040389b
- 004014b9: CALL _puts
- 004014bc: MOV dword ptr [ESP+local_8c,s_[OK]_Now_your_platform_is_active_0040...], 004038a5
- 004014bf: CALL _puts
- 004014c2: MOV EAX, 0x0
- 004014c5: JMP LAB_00401576
- 0040154b: LEA EAX, local_79, [EBP + -0x75]
- 0040154e: MOV EAX, dword ptr [EBP + param_1]
- 00401551: SUB EAX, dword ptr [EBP + local_10]
- 00401554: MOV dword ptr [EIP + local_80], 0x1
- 00401557: MOV dword ptr [ESP+local_8c,s_[ERROR]_Invalid_license_Key_Rem_0040...], 004038b5
- 0040155a: CALL _printf
- 0040155d: MOV dword ptr [EBP + local_10], 0x1
- 00401561: ADD EAX, 0x0
- 00401565: LEA EAX, local_10, [EBP + local_10]
- 00401568: MOV EAX, dword ptr [EBP + param_1]
- 0040156b: JLE LAB_00401469
- 00401571: MOV EAX, 0x0

The right window, titled "Decompile: _request_license_key - (program.exe)", shows the decompiled C code for the same function. The code is as follows:


```

1  int __cdecl _request_license_key(int param_1)
2  {
3      size_t sVar1;
4      undefined4 uVar2;
5      char acStack122 [101];
6      char local_15;
7      int local_14;
8      int local_10;
9      while( true ) {
10         if (param_1 < local_10) {
11             return 0;
12         }
13         local_10 = 1;
14         while( true ) {
15             if (param_1 < local_10) {
16                 return 0;
17             }
18             _printf("\n>> Enter your license key to activate the platform: ");
19             _fgets(acStack122 + 1, 100, (FILE *)_lob_xrefref);
20             sVar1 = _strlen(acStack122 + 1);
21             local_14 = sVar1 - 1;
22             if (acStack122[sVar1] == '\n') {
23                 acStack122[sVar1] = '\0';
24             }
25             uVar2 = _is_license_key(acStack122 + 1);
26             local_15 = (char)uVar2;
27             if (local_15 == '\0') break;
28             _printf("[ERROR] Invalid license key: Remaining attempts %d\n", param_1 - local_10);
29             local_10 = local_10 + 1;
30         }
31         _puts("[OK] Valid License Key!!!");
32         _puts("[OK] Activating...");
33         _puts("[OK] Now your platform is activated");
34         return 1;
35     }
    
```

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

inserir 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 dona 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 ensamblador i descompilat de les altres funcions mencionades es mostra a l'Apèndix.

```

1  #include <iostream>
2  #include <stdio.h>
3  #include <string.h>
4  using namespace std;
5
6  int sum_numbers(char *param_1) {
7      char local_38 [36];
8      int local_14;
9      int local_10;
10     strcpy(local_38,param_1);
11     local_14 = 0;
12     for (local_10 = 0; local_38[local_10] != '\0'; local_10 = local_10 + 1) {
13         if (('/' < local_38[local_10]) && (local_38[local_10] < ':')) {
14             local_14 = local_14 + local_38[local_10] + -0x30;
15         }
16     }
17     return local_14;
18 }
19
20 int is_divisible_by_a_and_b(int param_1,int param_2,int param_3) {
21     return ((param_1 % param_2 == 0) && (param_1 % param_3 == 0)) ? 1 : 0;
22 }
23
24 int main() {
25     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','A','3' };
26     int var4 = sum_numbers(key);
27     int var5 = is_divisible_by_a_and_b(var4,4,11);
28     cout << var4 << "\n" << var5;
29 }

```

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 dona 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 ensamblador de la figura 6, on es guarda el valor 0x1 al registre EAX. Podem veure instruccions similars al llarg de tot el codi ensamblador.

0040148d	89 45 ec	MOV	dword ptr [EBP + local_18],EAX
00401490	83 7d ec 00	CMP	dword ptr [EBP + local_18],0x0
00401494	74 07	JZ	LAB_0040149d
00401496	b8 01 00 00 00	MOV	EAX,0x1

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:

00401565	8b 45 f4	MOV	EAX,dword ptr [EBP + local_10]
00401568	3b 45 08	CMP	EAX,dword ptr [EBP + param_1]
0040156b	0f 8e 48 ff ff ff	JLE	LAB_004014b9
00401571	b8 00 00 00 00	MOV	EAX,0x0

Figura 7: Instruccions per sortir del bucle

A la figura 20, a l'Apèndix, es mostra el codi assembleador 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 assembleador 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".

		<i>_sum_numbers</i>		XREF[1]:
0040138d	55	PUSH	EBP	
0040138e	89 e5	MOV	EBP,ESP	
00401390	83 ec 48	SUB	ESP,0x48	
00401393	8b 45 08	MOV	EAX,dword ptr [EBP + param_1]	
00401396	89 44 24 04	MOV	dword ptr [ESP + local_48],EAX	
0040139a	8d 45 cc	LEA	EAX=>local_38,[EBP + -0x34]	
0040139d	89 04 24	MOV	dword ptr [ESP+local_4c],EAX	
004013a0	e8 7b 0a 00 00	CALL	_strcpy	
004013a5	c7 45 f0 00 00 00 00	MOV	dword ptr [EBP + local_14],0x0	
004013ac	c7 45 f4 00 00 00 00	MOV	dword ptr [EBP + local_10],0x0	
004013b3	eb 36	JMP	LAB_004013eb	
		LAB_004013b5		XREF[1]:
004013b5	8d 55 cc	LEA	EDX=>local_38,[EBP + -0x34]	
004013b8	8b 45 f4	MOV	EAX,dword ptr [EBP + local_10]	
004013bb	01 d8	ADD	EAX,EDX	
004013bd	0f b6 00	MOVZX	EAX,byte ptr [EAX]	
004013c0	3c 2f	CMP	AL,0x2f	
004013c2	7e 23	JLE	LAB_004013e7	
004013c4	8d 55 cc	LEA	EDX=>local_38,[EBP + -0x34]	
004013c7	8b 45 f4	MOV	EAX,dword ptr [EBP + local_10]	
004013ca	01 d8	ADD	EAX,EDX	
004013cc	0f b6 00	MOVZX	EAX,byte ptr [EAX]	
004013cf	3c 39	CMP	AL,0x39	
004013d1	7f 14	JG	LAB_004013e7	
004013d3	8d 55 cc	LEA	EDX=>local_38,[EBP + -0x34]	
004013d6	8b 45 f4	MOV	EAX,dword ptr [EBP + local_10]	
004013d9	01 d8	ADD	EAX,EDX	
004013db	0f b6 00	MOVZX	EAX,byte ptr [EAX]	
004013de	0f be c0	MOVSB	EAX,AL	
004013e1	83 e8 30	SUB	EAX,0x30	
004013e4	01 45 f0	ADD	dword ptr [EBP + local_14],EAX	
		LAB_004013e7		XREF[2]:
004013e7	83 45 f4 01	ADD	dword ptr [EBP + local_10],0x1	
		LAB_004013eb		XREF[1]:
004013eb	8d 55 cc	LEA	EDX=>local_38,[EBP + -0x34]	
004013ee	8b 45 f4	MOV	EAX,dword ptr [EBP + local_10]	
004013f1	01 d8	ADD	EAX,EDX	
004013f3	0f b6 00	MOVZX	EAX,byte ptr [EAX]	
004013f6	84 c0	TEST	AL,AL	
004013f8	75 bb	JNZ	LAB_004013b5	

Figura 8: Codi assembleador 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.

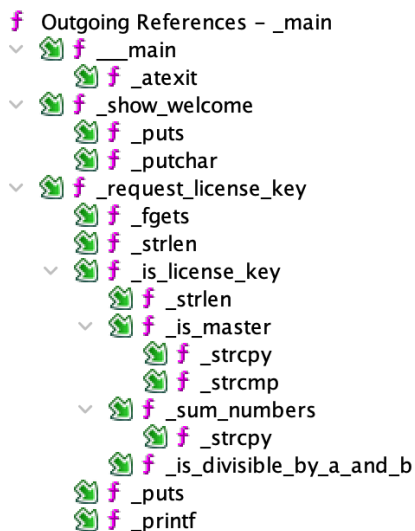


Figura 9: Funcions del programa analitzat

Heu detectat algun punt vulnerable?

Sí. Aquesta aplicació és per a validar la clau i 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 ensamblador de la funció *is_master* es pot veure que es crida a la funció *strcpy* i *strcmp*. La funció *strcpy* copia 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.

```

      _is_master
00401350 55          PUSH     EBP
00401351 89 e5      MOV      EBP,ESP
00401353 83 ec 48   SUB      ESP,0x48
00401356 8b 45 08   MOV      EAX,dword ptr [EBP + param_1]
00401359 89 44 24 04 MOV      dword ptr [ESP + local_48],EAX
0040135d 8d 45 d4   LEA      EAX=>local_30,[EBP + -0x2c]
00401360 89 04 24   MOV      dword ptr [ESP+local_4c],EAX
00401363 e8 b8 0a   CALL     _strcpy
00401368 c7 44 24   MOV      dword ptr [ESP + local_48],.rdata
00401370 8d 45 d4   LEA      EAX=>local_30,[EBP + -0x2c]
00401373 89 04 24   MOV      dword ptr [ESP+local_4c],EAX
00401376 e8 ad 0a   CALL     _strcmp
0040137b 85 c0      TEST     EAX,EAX
0040137d 75 07      JNZ      LAB_00401386
0040137f b8 01 00   MOV      EAX,0x1
00401384 eb 05      JMP      LAB_0040138b
```

Figura 10: Funcions *strcpy* i *strcmp*

Quins condicionals has trobat?

Al llarg del codi ensamblador 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 ensamblador 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)		
char *	AL:1	<RETURN>	
undefined1	Stack[0x4]:4	param_1	XREF
undefined4	Stack[-0x30]:1	local_30	XREF
undefined4	Stack[-0x40]:4	local_40	XREF
undefined4	Stack[-0x4c]:4	local_4c	XREF
	.text		XREF[1]:
00401350 55	_is_master		
	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.

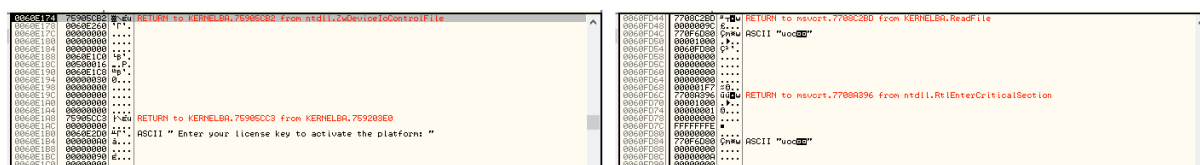


Figura 13: Fragment de la pila durant l'execució del programa

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 ensamblador.

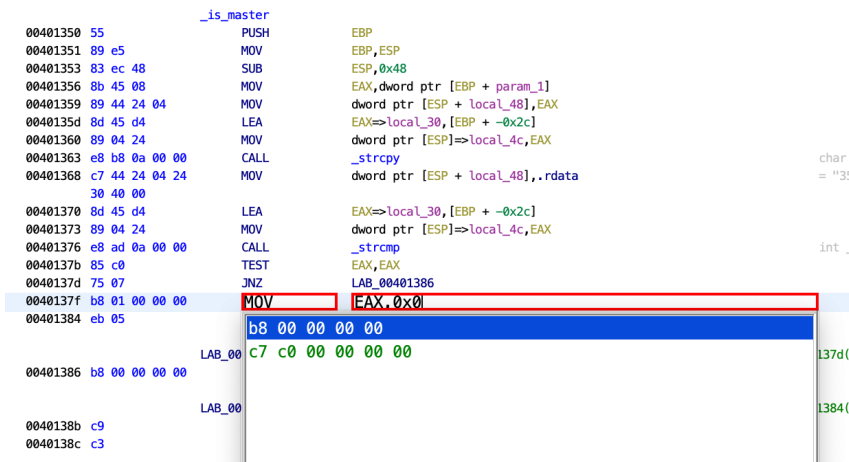


Figura 14: Modificació del codi ensamblador

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

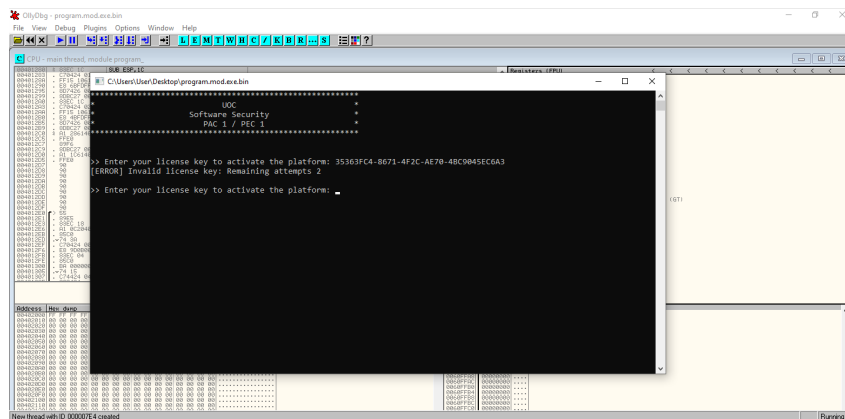


Figura 15: En executar no accepta la clau per defecte

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 ensamblador 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 ensamblador i tornar a exportar el programa, demostrant la importància de signar els programes amb algun algorisme 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

The screenshot displays the Ghidra interface with two main panes. The left pane, titled 'Listing: program.exe', shows the assembly code for the function `_is_license_key`. The code includes instructions like `PUSH EBP`, `MOV ESP, EBP`, `MOV EAX, dword ptr [EBP + param_1]`, and various conditional jumps and calls. The right pane, titled 'Decompile: _is_license_key - (program.exe)', shows the corresponding decompiled C code. The decompiled code defines a function `__cdecl _is_license_key(char *param_1)` that returns a `bool`. It uses local variables `bool bVar1`, `size_t sVar2`, `int iVar4`, and `uint uVar3`. The logic involves checking the length of the input string, calling `_is_master`, and performing a series of conditional checks and arithmetic operations before returning `uVar3`.

Figura 16: Llenguatge ensamblador i codi descompilat de `is_license_key`

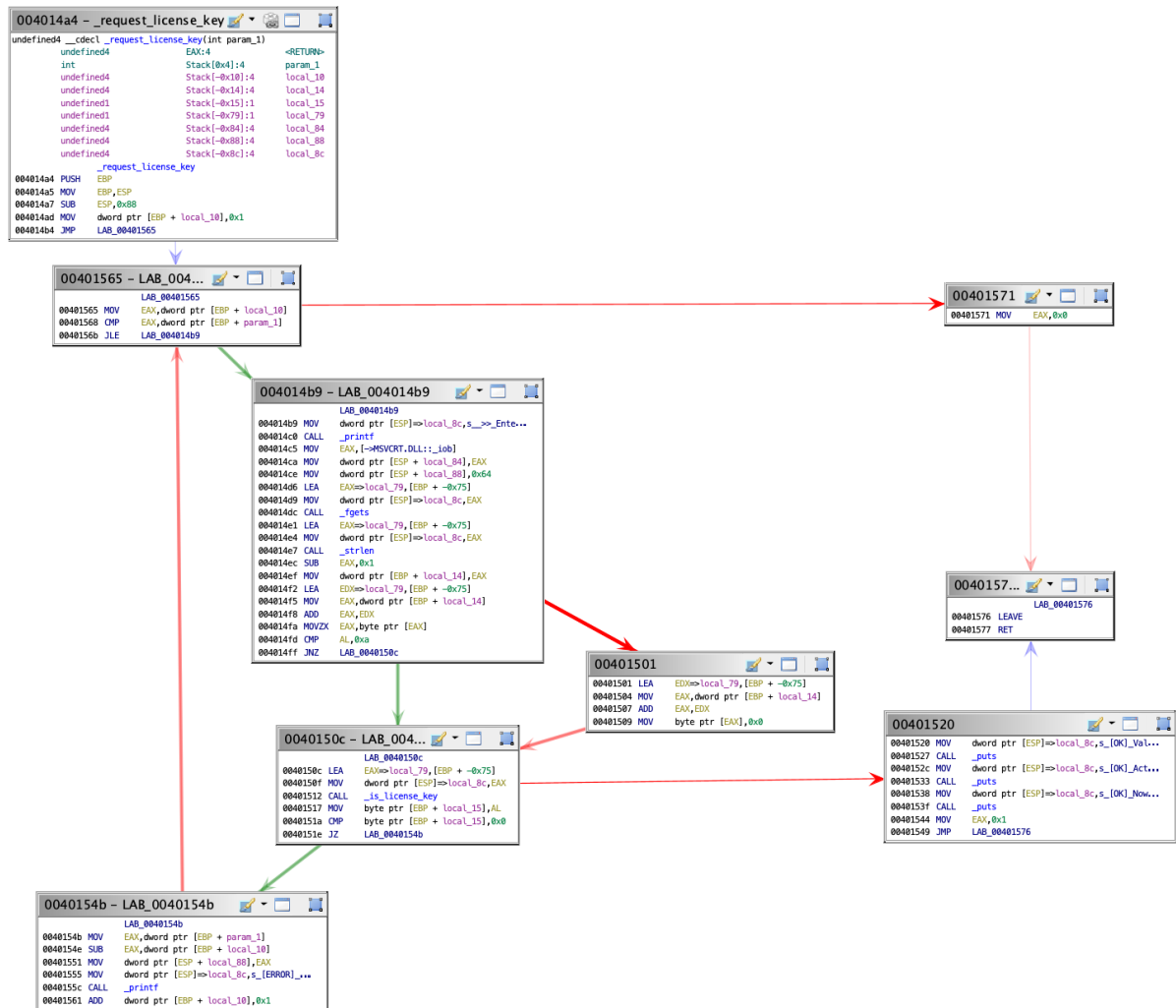


Figura 20: Codi ensamblador de la funció *request_license_key*