

# **C.7 DEZASAMBLARE ȘI PATCHING CU IDA**

**PAUL A. GAGNIUC**



**Academia Tehnică Militară „Ferdinand I”**

# PRINCIPALELE PĂRȚI ALE PREZENTĂRII

## C.7 Dezasamblare și Patching cu **IDA free**:

- C.7.1 COMPILARE (FASM) ȘI DEZASAMBLARE EXECUTABIL (PE)
- C.7.2 OPTIMIZARI AUTOMATE FACUTE DE COMPILATOR (C++)
- C.7.3 ENCODARE DIACRITICE (ASCII VS UTF-8)
- C.7.4 DEZASAMBLARE ȘI PETICIRE EXECUTABIL (NOP)
- C.7.5 DEZASAMBLARE ȘI PETICIRE EXECUTABIL (JMP)
- C.7.6 DISTRUGEREA FUNCȚIONALITĂȚII CU INSTRUȚIUNI NOP
- C.7.7 DETECTAREA UNEI FUNCȚII DE CRIPTARE
- C.7.8 INGINERIA INVERSĂ DE COD MALWARE & IDA



C.7.1

# COMPILARE (FASM) ȘI DEZASAMBLARE EXECUTABIL (PE)



# Compilarea unui exemplu (**mic.asm**) în: FASM

	<pre>format PE GUI 4.0    ; specifică formatul executabilului ca fiind PE (Portable Executable) pentru interfața grafică Windows, versiunea 4.0 entry start          ; definește punctul de intrare al programului, unde execuția va începe</pre>
	<pre>include 'INCLUDE/win32a.inc' ; include fișierul de antet 'win32a.inc' care conține macro-uri și definiții pentru interfața cu API-ul Windows</pre>
.data	<pre>section '.data' data readable writeable ; începe o secțiune de date care poate fi citită și scrisă     title db 'Inginerie Inversa',0      ; definește un șir de caractere pentru titlu, terminat cu null (0) pentru a fi folosit în mesaj     message db 'Malware',0             ; definește un șir de caractere pentru mesaj, terminat cu null (0)</pre>
.text	<pre>section '.text' code readable executable ; începe secțiunea de cod, care poate fi citită și executată start:                                   ; eticheta 'start', care este punctul de intrare al programului     push 0                               ; pune valoarea 0 pe stivă, reprezentând handle-ul pentru fereastra părinte (niciuna în acest caz)     push title                           ; pune adresa șirului de titlu pe stivă     push message                         ; pune adresa șirului de mesaj pe stivă     push 0                               ; pune valoarea 0 pe stivă, care reprezintă stilul cutiei de mesaj (MB_OK)     call [MessageBoxA]                  ; apelează funcția MessageBoxA din user32.dll      push 0                               ; pune valoarea 0 pe stivă, argument pentru ExitProcess care indică statusul de ieșire     call [ExitProcess]                  ; apelează funcția ExitProcess din kernel32.dll pentru a încheia programul</pre>
.idata	<pre>section '.idata' import data readable writeable ; începe secțiunea de date pentru importuri, care poate fi citită și scrisă  library kernel32,'KERNEL32.DLL',\        ; specifică că vom folosi funcții din KERNEL32.DLL     user32,'USER32.DLL'                  ; și din USER32.DLL  import kernel32,\                          ; începe definițiile de import pentru kernel32.dll     ExitProcess,'ExitProcess'            ; specifică că dorim să folosim funcția ExitProcess din acest DLL  import user32,\                            ; începe definițiile de import pentru user32.dll     MessageBoxA,'MessageBoxA'            ; specifică că dorim să folosim funcția MessageBoxA din acest DLL</pre>



# COMPILARE PENTRU DEZASAMBLARE

## COMPILARE CU FASM

- `cd cale_dir_FASM`
- `FASM.EXE mic.asm mic.exe`
- FASM vine ca o arhivă, nu trebuie instalat.

mic.exe (2Kb)

```
Command Prompt
Microsoft Windows [Version 10.0.19043.2364]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Elitebook>cd C:\Users\Elitebook\Desktop\fasmw17332

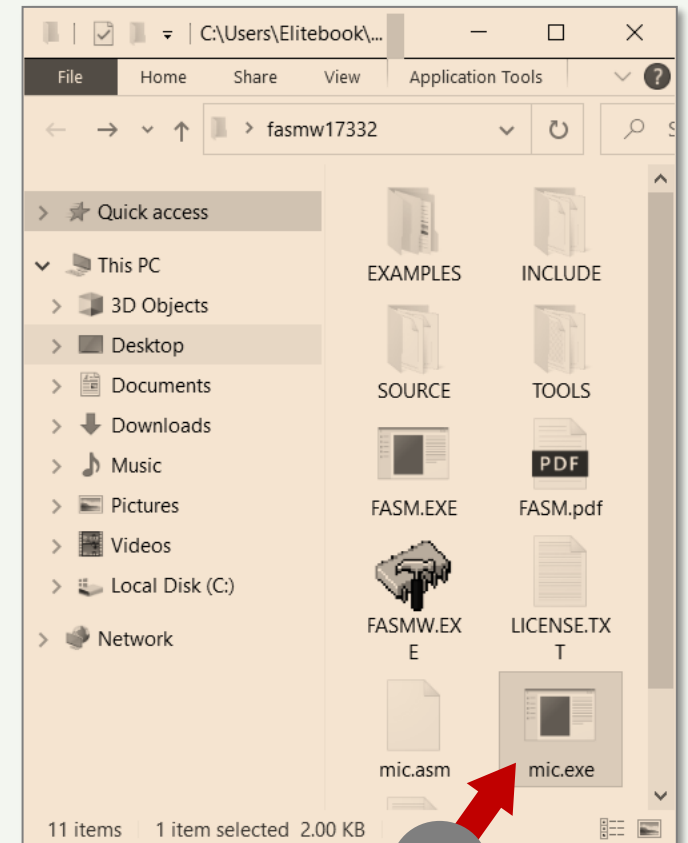
C:\Users\Elitebook\Desktop\fasmw17332>
```

```
Command Prompt

C:\Users\Elitebook>cd C:\Users\Elitebook\Desktop\fasmw17332

C:\Users\Elitebook\Desktop\fasmw17332>FASM.EXE mic.asm mic.exe
flat assembler version 1.73.32 (1048576 kilobytes memory)
3 passes, 2048 bytes.

C:\Users\Elitebook\Desktop\fasmw17332>
```



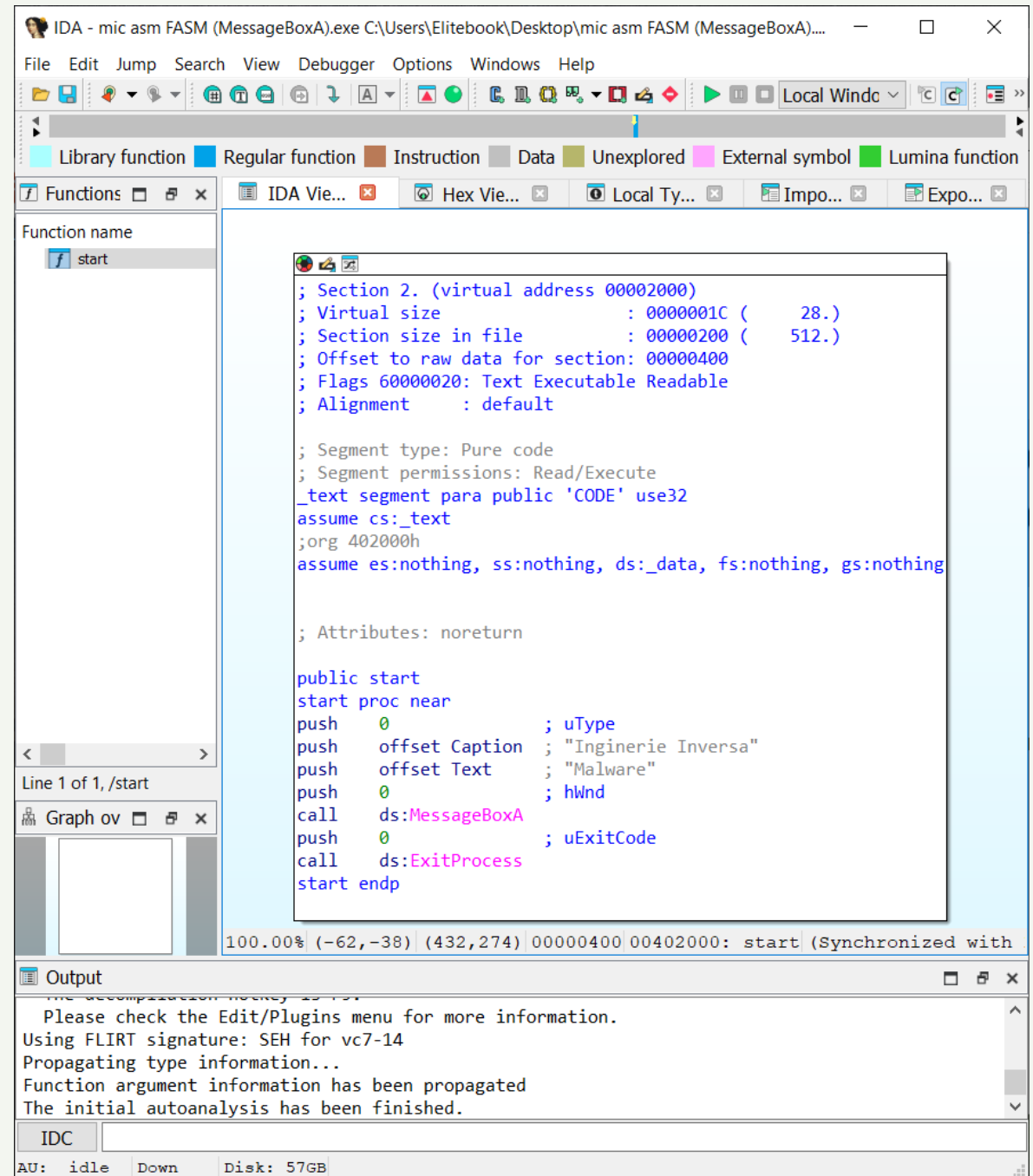
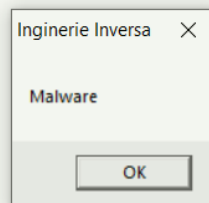
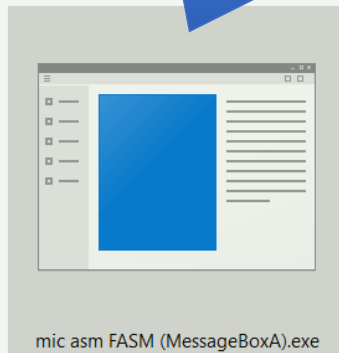
# Executabilul peticit

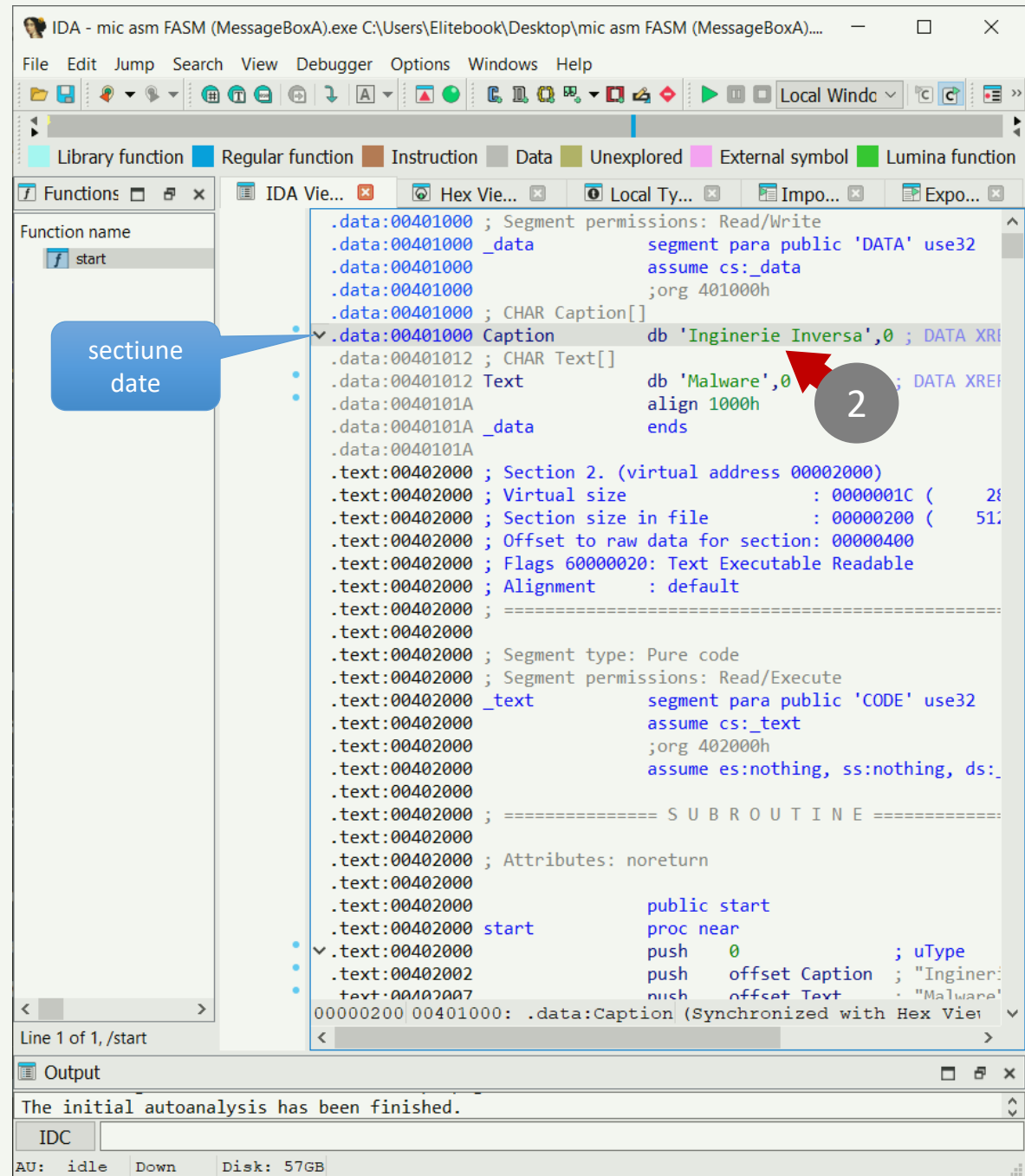
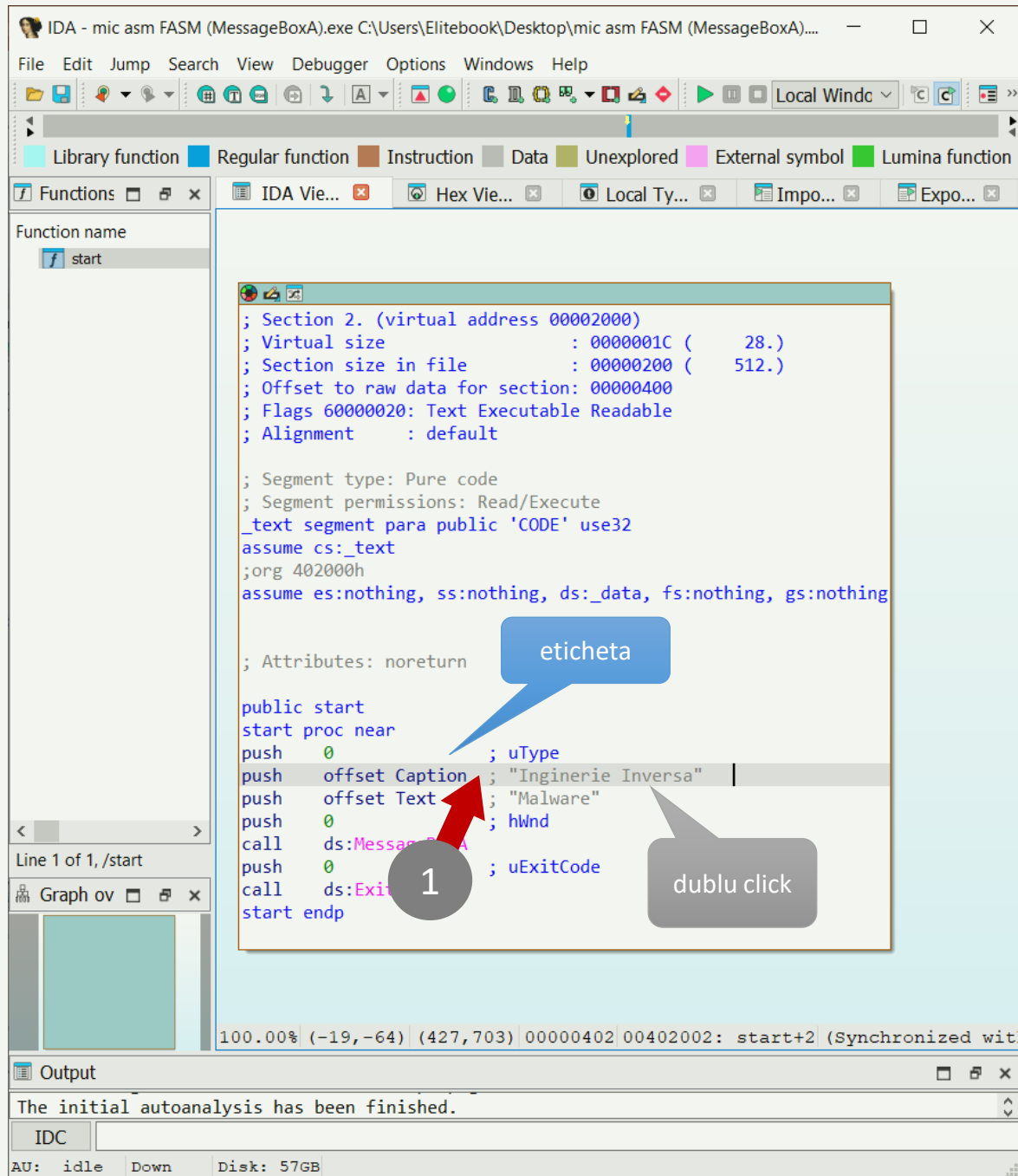
sau modificat/carpit/peticit

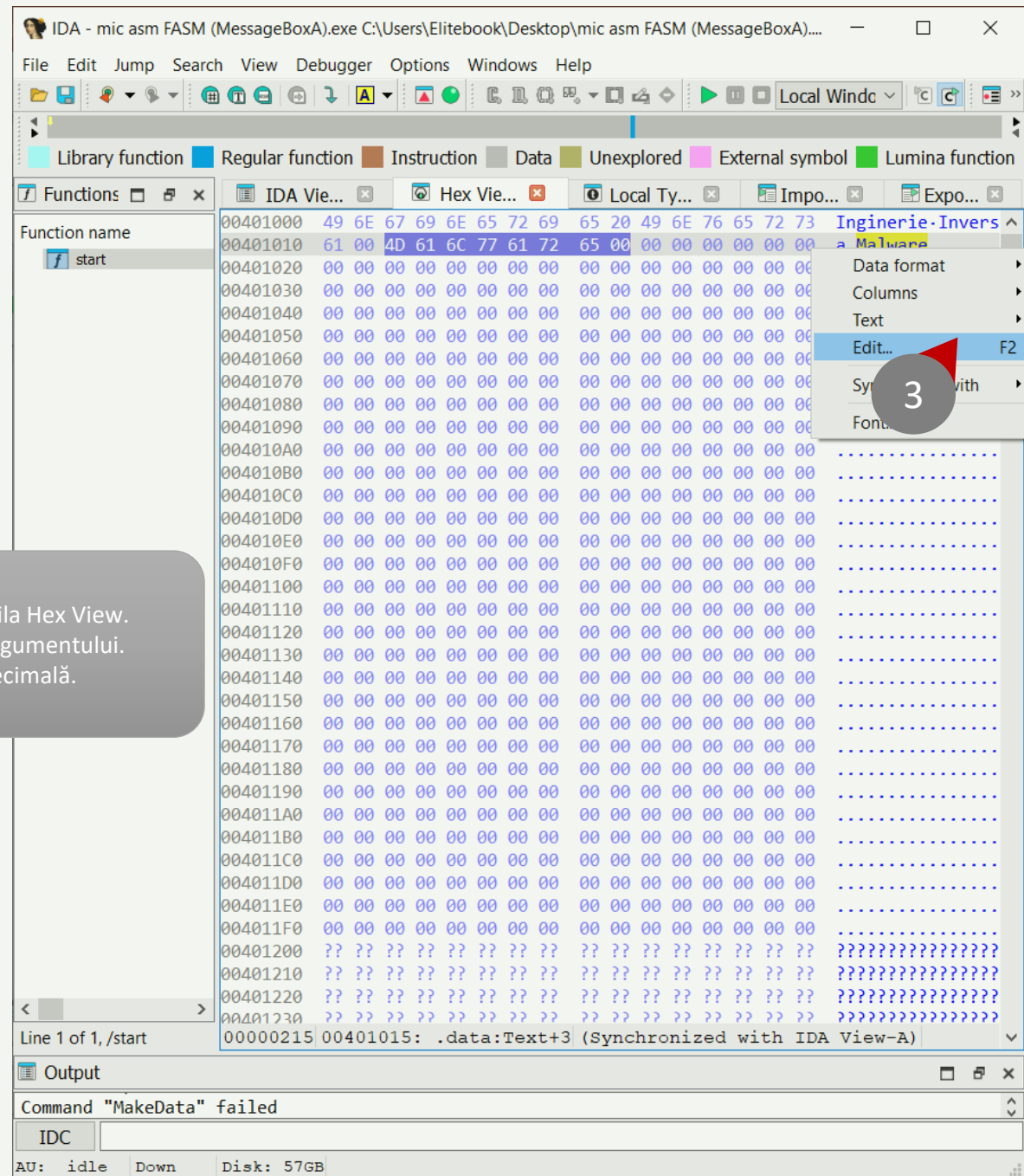
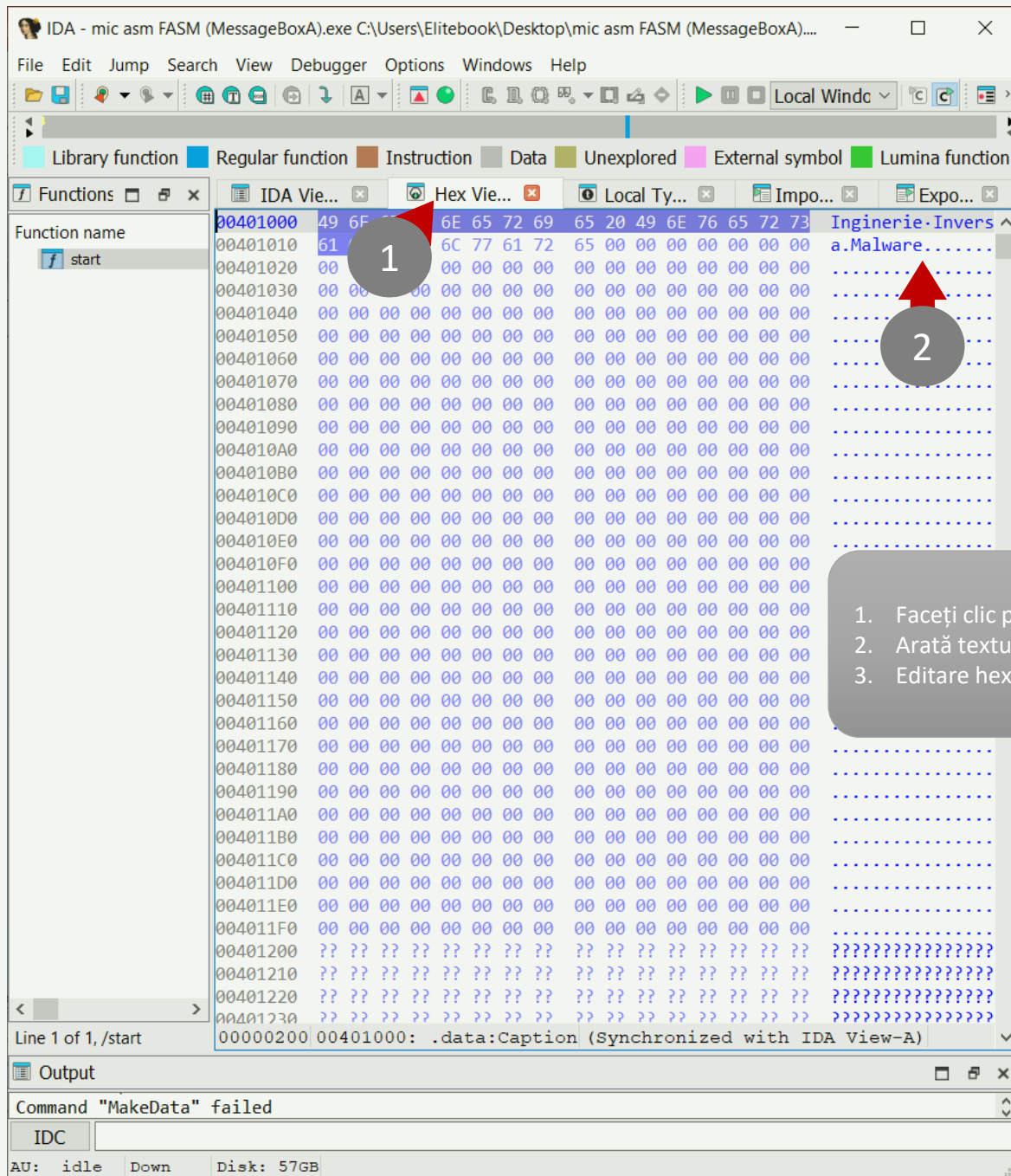
PATCHING

Redenumiți „mix.exe” în „mic asm FASM (MessageBoxA).exe” pentru a fi sincronizați cu exemplele.  
Deschideți fișierul executabil cu IDA.

Executabil original !

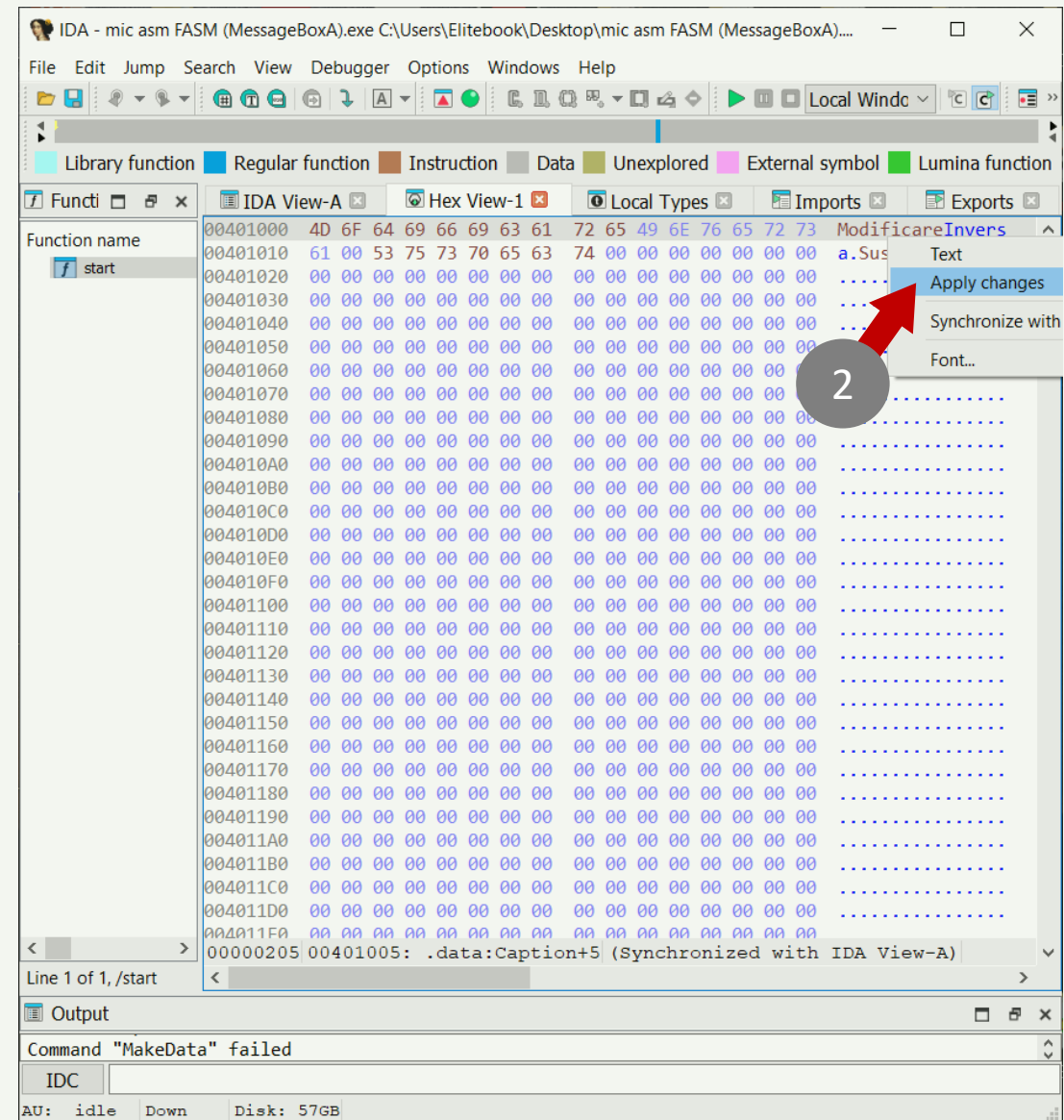
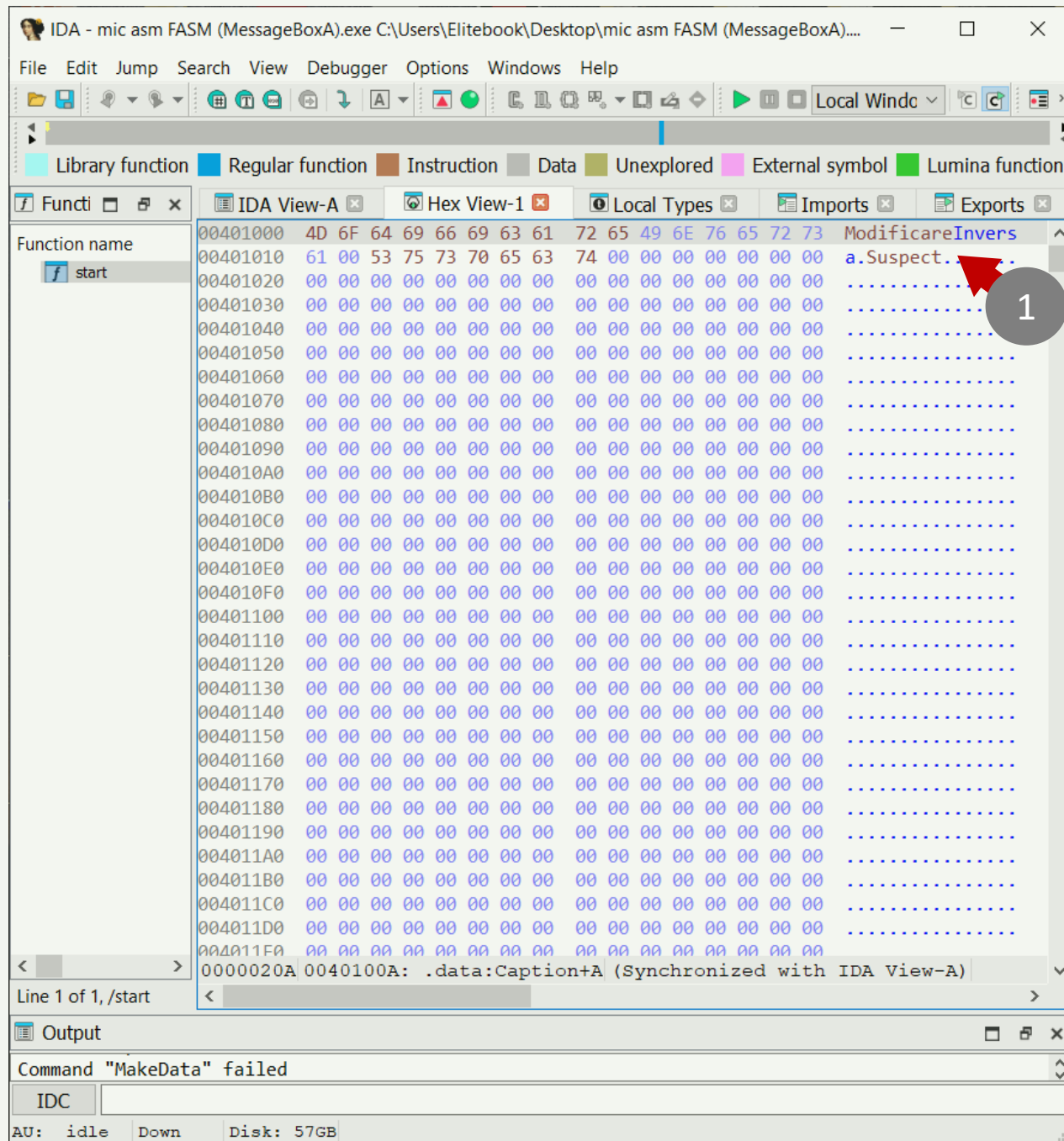




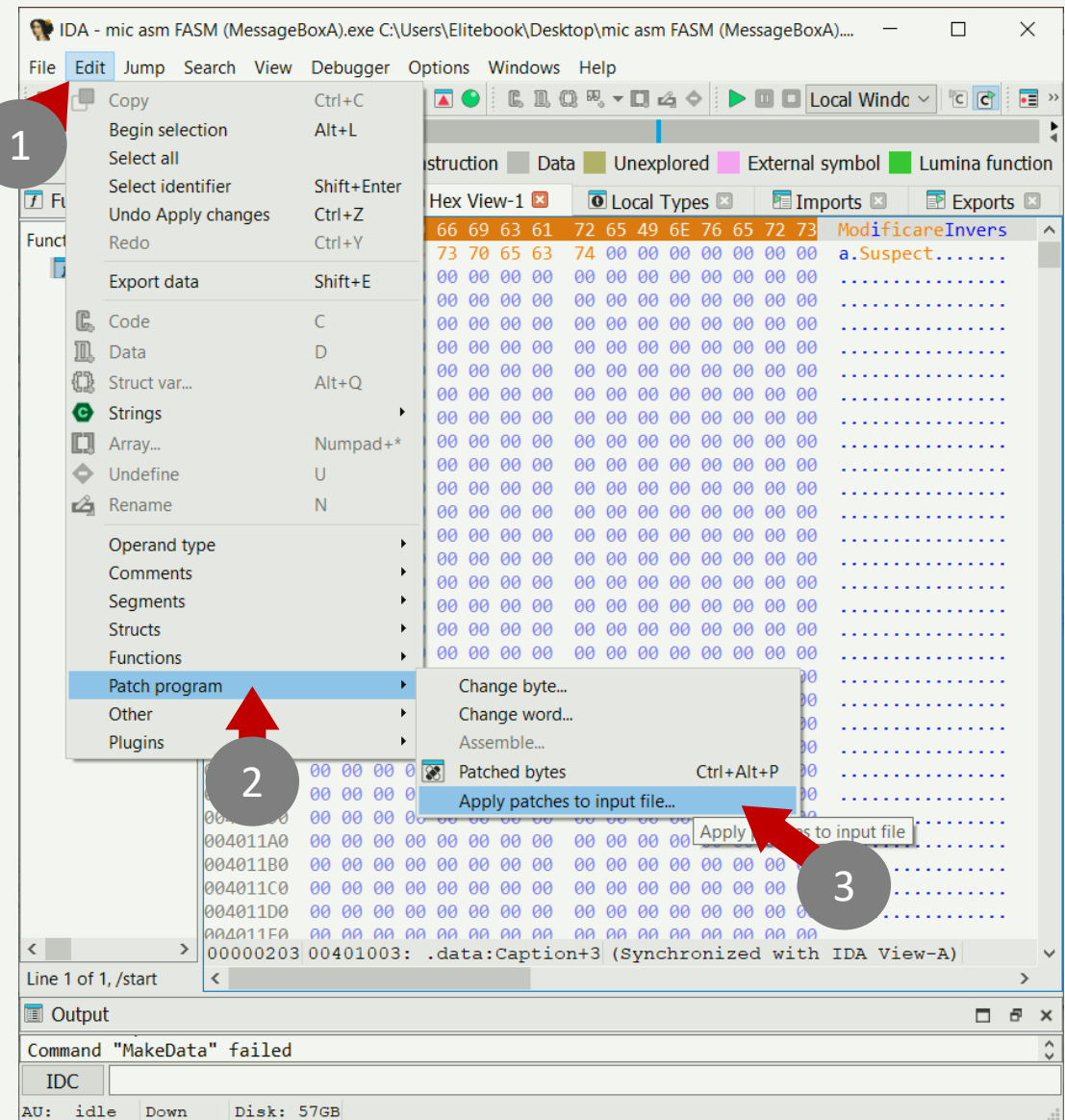
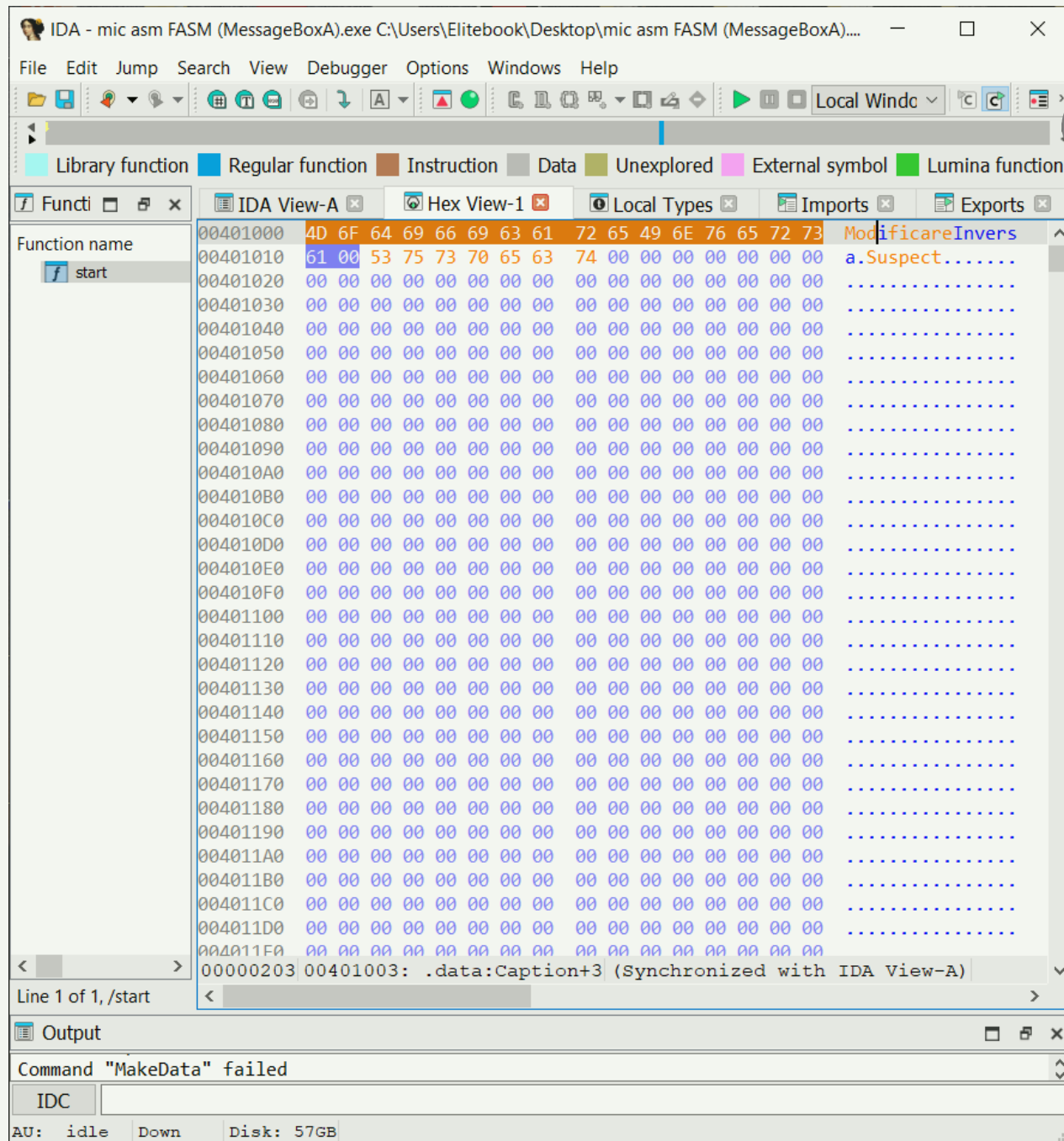


1. Faceți clic pe fila Hex View.
2. Arată textul argumentului.
3. Editare hexazecimală.

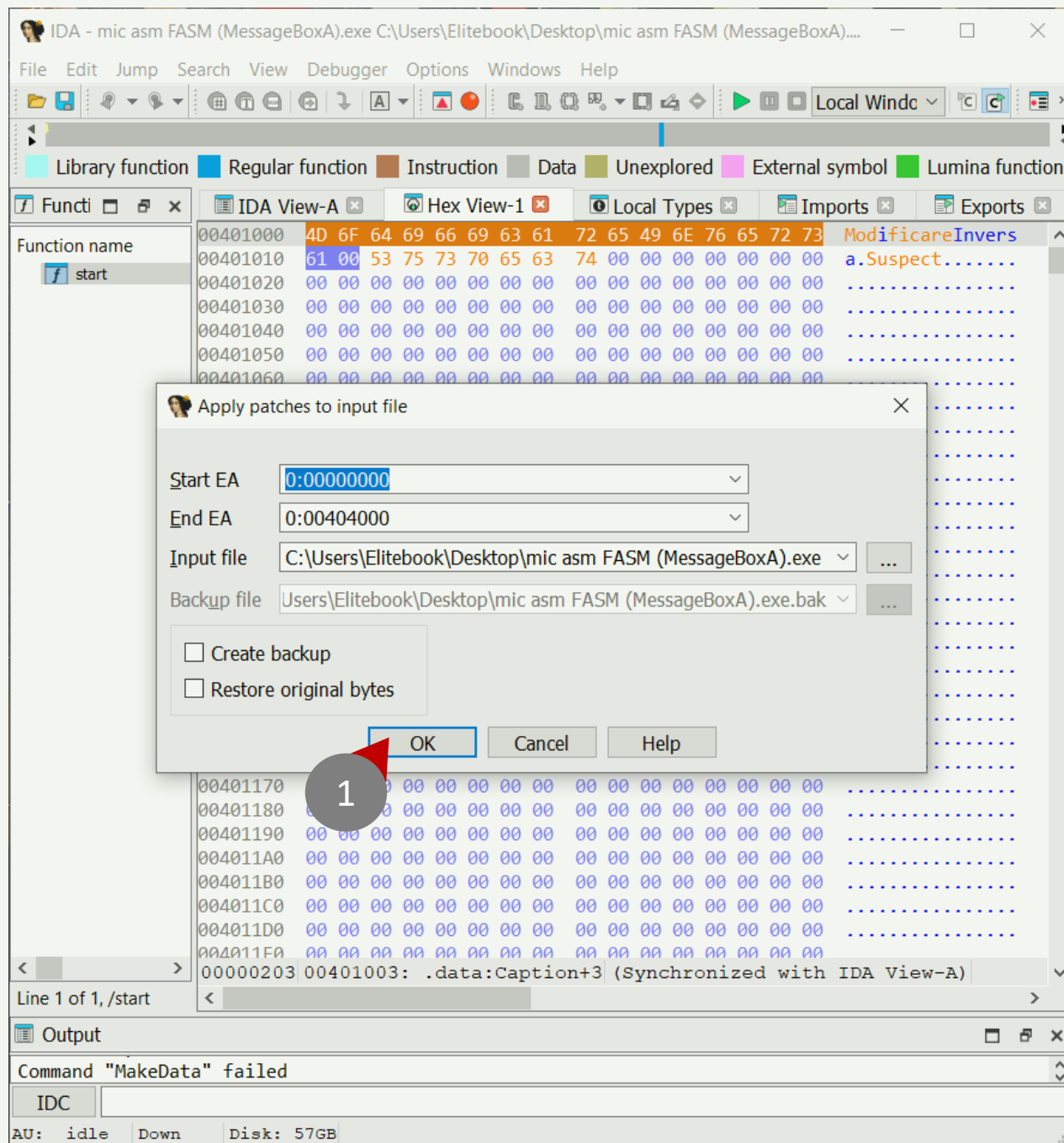




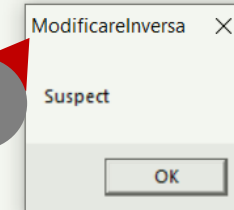
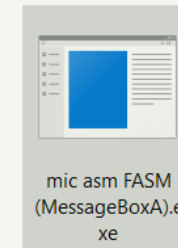
1. Modificări directe ale textului de la tastatură.
2. Aplicați modificările.



1. Edit.
2. Peticire executabil.
3. Aplicare peticire.



Executabil modificat !



2



# Pentru a modifica un șir de caractere și a aplica un patch unui executabil în IDA Free:

**1. Găsiți Șirul.** Utilizați funcția de căutare din IDA pentru a localiza șirul pe care doriți să îl modificați. Acest lucru poate fi făcut uitându-vă în fereastra "Strings" sau navigând către secțiunea de date unde este definit șirul.

**2. Editați Șirul.** Odată ce ați găsit șirul:

- Faceți dublu clic pe acesta pentru a deschide linia în vizualizarea dezasamblării sau în vizualizarea hex.
- Dacă este în vizualizarea dezasamblării, puteți tasta peste șirul existent cu cel nou pe care doriți să îl utilizați. Asigurați-vă că noul șir nu este mai lung decât cel vechi, deoarece acest lucru ar putea suprascrie datele sau codul adiacent.
- Dacă este în vizualizarea hex, editați valorile hexadecimale care corespund caracterelor ASCII ale șirului.

**3. Aplicați Patch-ul la Binare.**

- În meniul IDA, mergeți la **Edit -> Patch program -> Apply patches to input file...** pentru a scrie schimbările înapoi în fișierul executabil.
- S-ar putea să fie nevoie să specificați un nou nume pentru fișierul patch-uit sau să alegeți să suprascrieți fișierul existent.

**4. Salvați Executabilul Patch-uit.** După ce ați făcut modificările și ați aplicat patch-ul fișierului de intrare, asigurați-vă că salvați noul binar pentru ca schimbările dvs. să fie păstrate. Utilizați **File -> Produce file -> Create EXE...** pentru a salva executabilul patch-uit, dacă este necesar.



C.7.2

# OPTIMIZARI AUTOMATE FĂCUTE DE COMPILATOR (C++)





g++ -Os -s -nostdlib -fno-exceptions -fno-rtti -Wl,--subsystem,windows minimal.cpp -o minimal.exe -luser32

Găsiți șirul „Ai sarit peste conditie!” în executabilul compilat?

# OPTIMIZAREA COMPILATORULUI C++

(DETECTEAZĂ CĂ `bool showMessageBox = true;` ESTE ADEVĂRAT, DECI CONSTANT)

Compilatorul detectează că `bool showMessageBox = true;` este constant prin urmare compilatorul îl trece în secțiunea `.rdata` (read only data)

3

Compilatorul înțelege că `bool showMessageBox = true;` este adevărat, deci constant.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    bool showMessageBox = true;

    MessageBox(NULL, "Ar trebui sa urmeze inca un message box !", "Inginerie Inversa ATM", MB_OK);

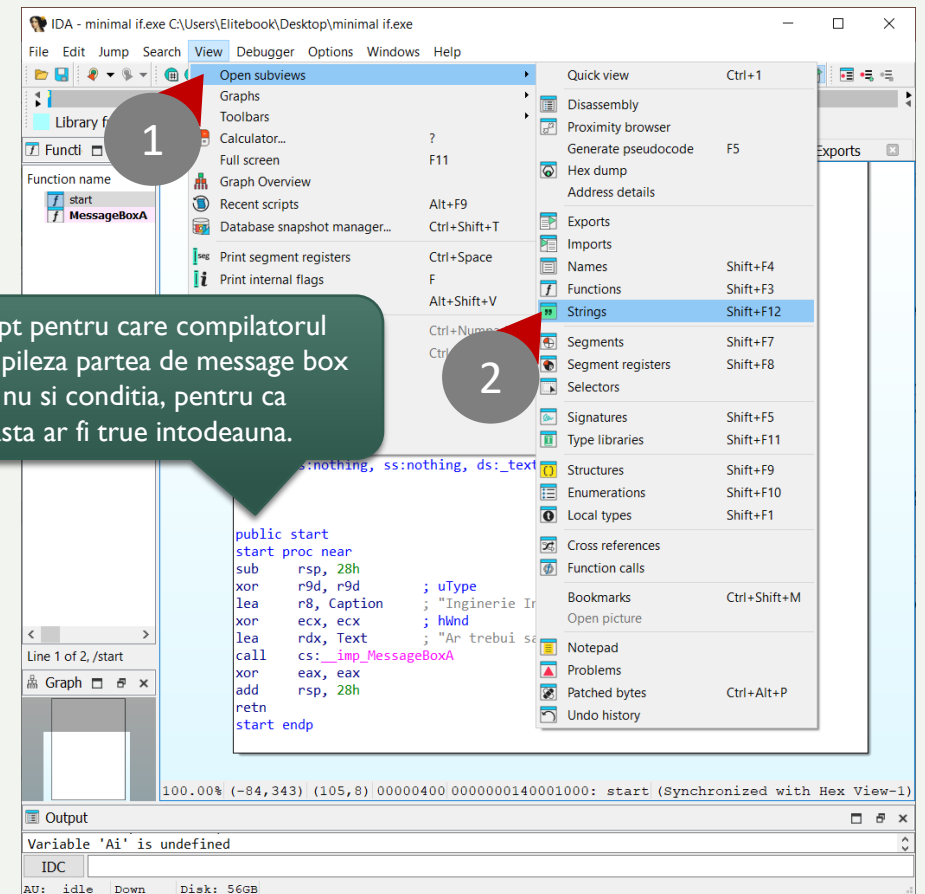
    if (!showMessageBox) {
        MessageBox(NULL, "Ai sarit peste conditie!", "Inginerie Inversa ATM", MB_OK);
    }

    return 0;
}
```

Drept pentru care compilatorul compilează partea de message box însă nu și condiția, pentru că aceasta ar fi `true` întotdeauna.

2

1



# DACĂ CONDIȚIA ESTE ADEVĂRATĂ...

MESSAGE BOX-UL SECUND ESTE INCLUS, DAR CONDIȚIA NU EXISTĂ DIN CAUZA OPTIMIZĂRII

Structurile de cod pot exista în codul sursă, dar nu și în versiunea compilată. Compilatorul elimină în timpul optimizării părțile de cod care nu sunt niciodată executate, deoarece introducerea lor este ca un balast.

Message box-ul secund este inclus, dar condiția nu există în forma compilată din cauza optimizării făcută automat de către compilator (deoarece `showMessageBox` este `true`).

```
public start
start proc near
push    rbp
sub     rsp, 20h
mov     rbp, cs:__imp_MessageBoxA
xor     r9d, r9d
xor     ecx, ecx
lea     r8, Caption
lea     rdx, Text
call    rbp
xor     r9d, r9d
lea     r8, Caption
xor     ecx, ecx
lea     rdx, aShowmessagebox
call    rbp
add     esp, 20h
pop     rbp
ret
start endp
```

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    bool showMessageBox = true;

    MessageBox(NULL, "Ar trebui sa urmeze inca un message box !", "Inginerie Inversa ATM", MB_OK);

    if (showMessageBox) {
        MessageBox(NULL, "showMessageBox = true", "Inginerie Inversa ATM", MB_OK);
    }

    return 0;
}
```

# C.7.3 ENCODARE DIACRITICHE (ASCII VS UTF-8)



# STOCAREA CARACTERELOR ȘI ÎNCERCAREA DE A AFIȘA DIACRITICE

ASCII (American Standard Code for Information Interchange)

UTF-8 este o codare variabilă:  
caracterele ASCII (A-Z, 0-9, etc.) → 1 byte  
caracterele cu accent, diacritice → 2 sau 3 bytes

```
.text:00000014:00000014: start+14 (Synchronized with Hex View-1)
public start
proc near ; DATA XREF: .pdata
push rbp
sub rsp, 20h
mov r9d, 4 ; uType
lea r8, Caption ; lpCaption
xor ecx, ecx ; hWnd
lea rdx, Text ; "E"
mov rbp, rdx ; __imp_MessageBoxA
call __imp_MessageBoxA
xor r9d, r9d ; uType
lea rax, "Alegere" ; "Alegere"
lea rcx, aAiAlesDa ; "Ai ales DA."
cmp eax, 6
nop
```

Line 1 of 2, /start

Output

The initial autoanalysis has been finished.  
Applied 2/2 patch(es)

IDC

AU: idle Down Disk: 56GB

```
.rdata:00000014:0000200B: const CHAR Text[1]
Text
db 'E' ; DATA XREF: star
db 0C8h
db 99h
db 74h ; t
db 69h ; i
db 20h
db 64h ; d
db 65h ; e
db 20h
db 61h ; a
db 63h ; c
db 6Fh ; o
db 72h ; r
db 64h ; d
db 3Fh ; ?
db 0
.rdata:00000014:0000201B: const CHAR aAlegere[1]
aAlegere
db "Alegere"
```

Line 1 of 2, /start

Output

The initial autoanalysis has been finished.  
Applied 2/2 patch(es)

IDC

AU: idle Down Disk: 56GB

0xC8 0x99 → este codul UTF-8 pentru litera „ș” (U+0219) stocat pe 2 bytes!

# C.7.4 DEZASAMBLARE ȘI PETICIRE EXECUTABIL (NOP)





# EXAMINAREA FUNCȚIONALITĂȚII EXECUTABILULUI FOLOSIND NOP



IDA - minimal.exe C:\Users\Elitebook\Desktop\minimal.exe

File Edit Jump Search View Debugger Options Windows Help

Library function Regular function Instruction Data Unexplored External symbol Lumina function

Functions

Function name

start

MessageBoxA

start

```
public start
proc near ; DATA XREF: .pdata
push rbx
sub rsp, 20h
mov rbx, cs:__imp_MessageBoxA
xor r9d, r9d ; uType
xor ecx, ecx ; hWnd
lea r8, Caption ; "Inginerie Invers
lea rdx, Text ; "Ar trebui sa urm
call rbx ; __imp_MessageBoxA
xor r9d, r9d ; uType
lea r8, Caption ; "Inginerie Invers
xor ecx, ecx ; hWnd
lea rdx, aShowmessagebox ; "showMessage
call rbx ; __imp_MessageBoxA
xor eax, eax
add rsp, 20h
pop rbx
retn
endp
```

00000434 00000000140001034: start+34

Line 1 of 2, /start

Output

Propagating type information...  
Function argument information has been propagated.  
The initial autoanalysis has been finished.  
Command "EditFunction" failed

IDC

AU: idle Down Disk: 56GB

IDA - minimal.exe C:\Users\Elitebook\Desktop\minimal.exe

File Edit Jump Search View Debugger Options Windows Help

Library function Regular function Instruction Data Unexplored External symbol Lumina function

Functions

Function name

start

MessageBoxA

start

```
public start
proc near ; DATA XREF: .pdata
push rbx
sub rsp, 20h
mov rbx, cs:__imp_MessageBoxA
xor r9d, r9d ; uType
xor ecx, ecx ; hWnd
lea r8, Caption ; "Inginerie Invers
lea rdx, Text ; "Ar trebui sa urm
call rbx ; __imp_MessageBoxA
xor r9d, r9d ; uType
lea r8, Caption ; "Inginerie Invers
xor ecx, ecx ; hWnd
lea rdx, aShowmessagebox ; "showMessage
call rbx ; __imp_MessageBoxA
xor eax, eax
add rsp, 20h
pop rbx
retn
endp
```

00000434 00000000140001034: start+34

Line 1 of 2, /start

Output

Propagating type information...  
Function argument information has been propagated.  
The initial autoanalysis has been finished.  
Command "EditFunction" failed

IDC

AU: idle Down Disk: 56GB

# NOUL EXECUTABIL

## AFIŞEAZĂ UN SINGUR MESAJ



**Patch Bytes**

Address: 0x140001034  
File offset: 0x434  
Original value: FF D3 31 C0 48 83 C4 20 5B C3 90 90 FF 25 F2 3F  
Values: **FF D3 31 C0 48 83 C4 20 5B C3 90 90 FF 25 F2 3F**

1

OK Cancel Help

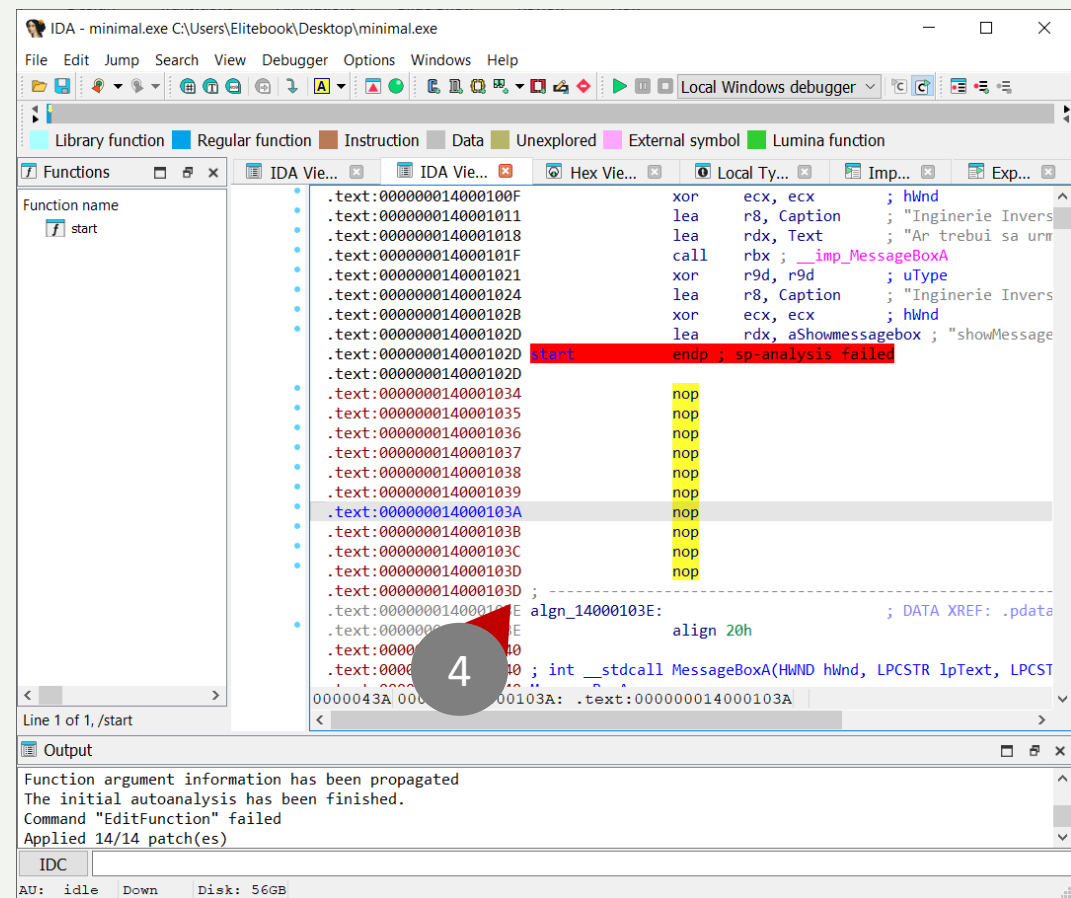
**Patch Bytes**

Address: 0x140001034  
File offset: 0x434  
Original value: FF D3 31 C0 48 83 C4 20 5B C3 90 90 FF 25 F2 3F  
Values: **90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90**

2

3

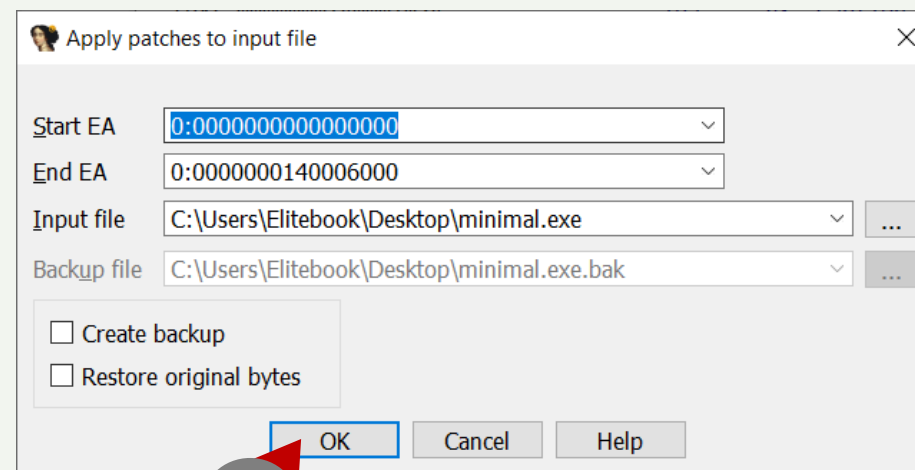
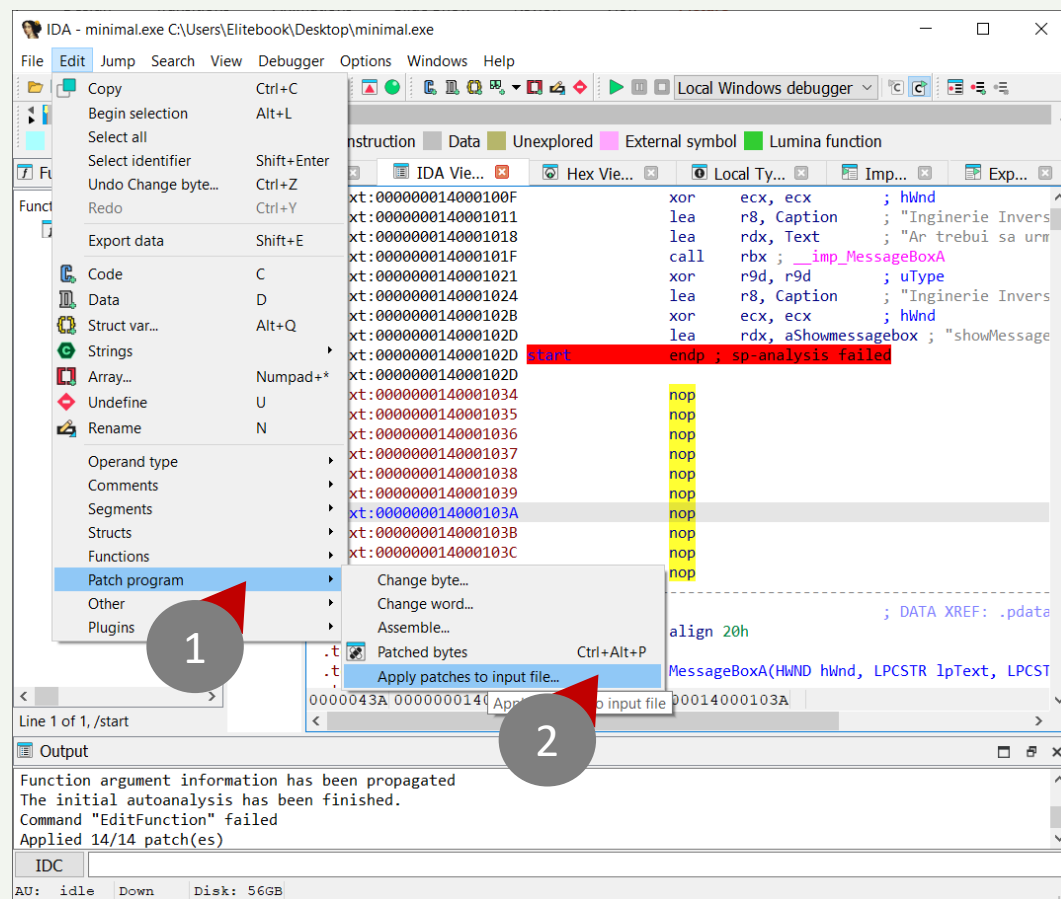
OK Cancel Help



# PETICIRE EXECUTABIL

ZONA **NOP** UNDE POT FI FĂCUTE ADĂUGĂRI DE COD

**Start EA** (Effective Address) și **End EA** nu se modifică.



**Start EA.** Este adresa efectivă la care începe secvența de cod pe care dorești să o modifice. Aceasta este adresa de memorie unde sunt stocate instrucțiunile sau datele pe care intenționezi să le schimbi.

**End EA.** Este adresa efectivă la care se termină secvența de cod pe care vrei să o modifice. Practic, definești intervalul de memorie care va fi afectat de patch-urile aplicate, de la "start EA" până la "end EA" (exclusiv).

C.7.5

# DEZASAMBLARE ȘI PETICIRE EXECUTABIL (JMP)



# SE ÎNLOCUIEȘTE JZ CU JMP

## OCOLIȚI CONDIȚIILE FOLOSIND INSTRUCȚIUNILE DE SALT

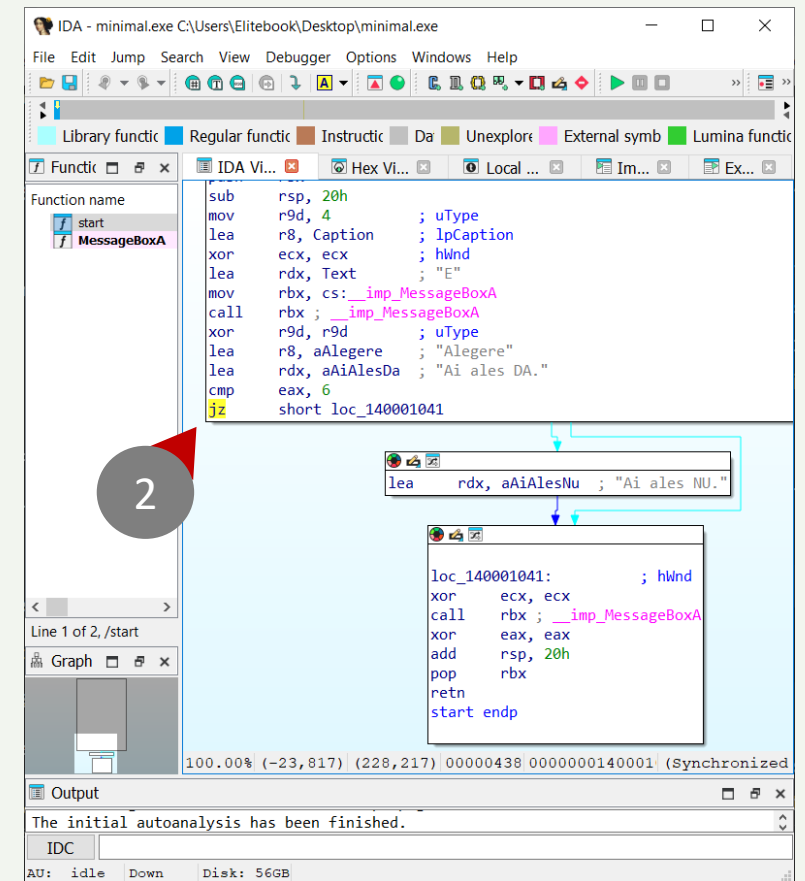
Cmp eax, 6  
Jz short loc\_140001041

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    int answer = MessageBox(NULL, "Esti de acord?", "Întrebare", MB_YESNO);
    if (answer == IDYES) {
        MessageBox(NULL, "Ai ales DA.", "Alegere", MB_OK);
    } else {
        MessageBox(NULL, "Ai ales NU.", "Alegere", MB_OK);
    }
    return 0;
}
```

1

Puteți înlocui direct instrucțiunea de salt condițional (**jz**) cu un salt necondițional (**jmp**). Acest lucru va cauza ca programul să execute întotdeauna blocul de cod care urmează după eticheta specificată în instrucțiunea de salt, ignorând orice condiție.



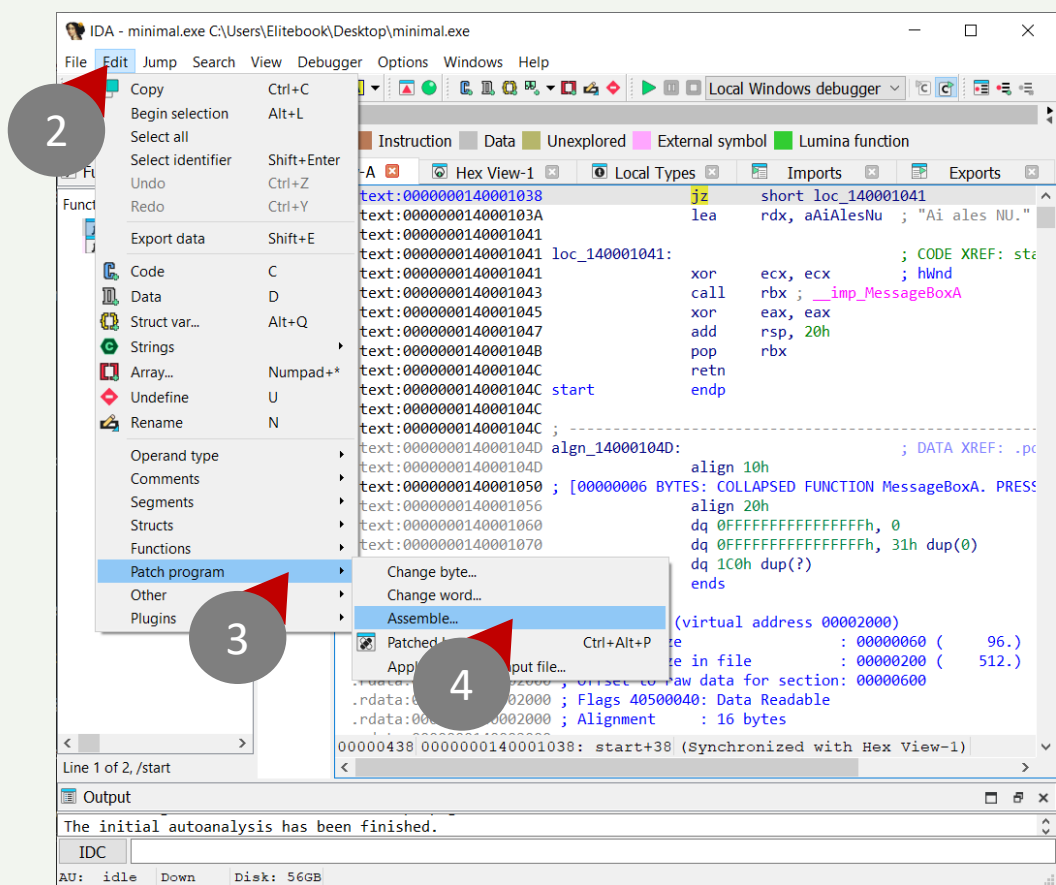
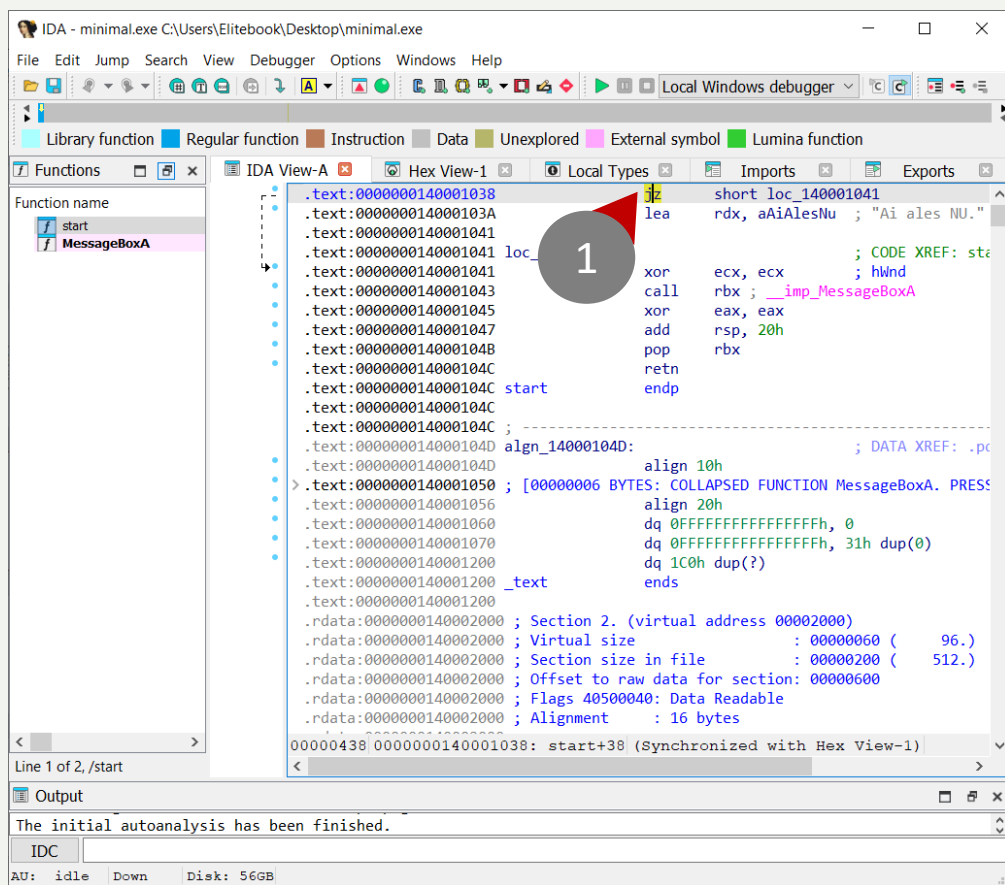
2



# ASAMBLARE

## ÎNLOCUIREA JZ CU JMP

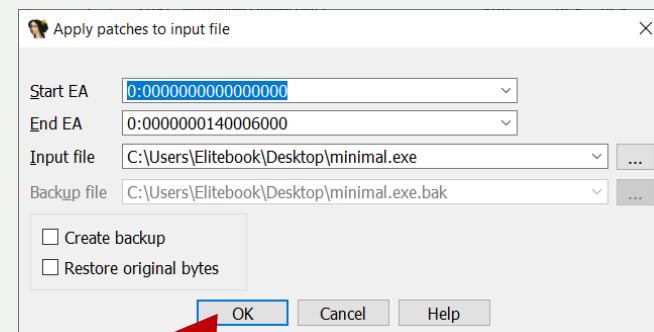
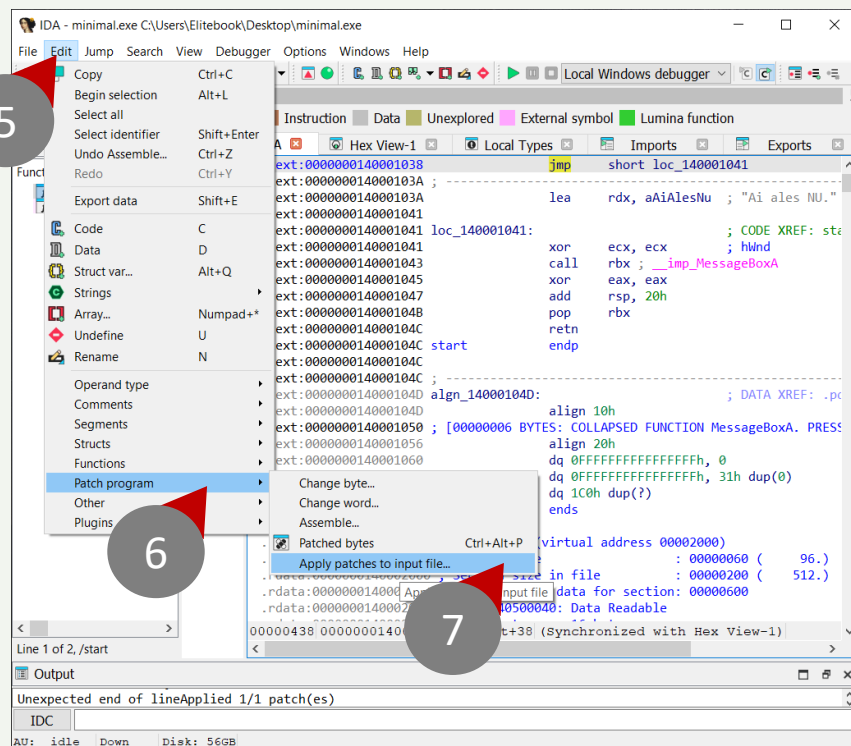
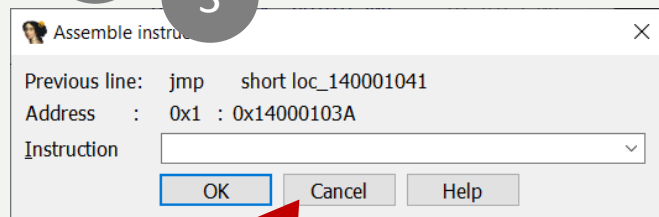
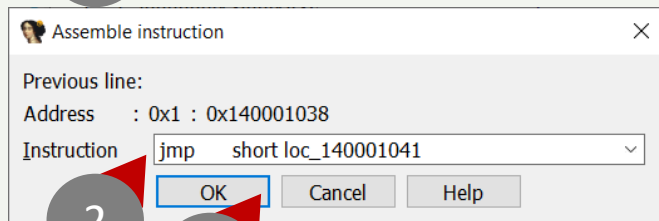
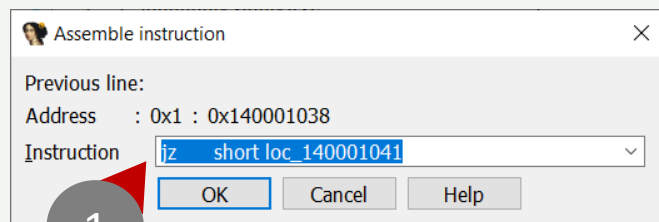
1. Localizează instrucțiunea jz pe care vrei să o modifici.
2. Deschide: **Edit > Patch program > Assemble** sau prin dublu-click pe instrucțiunea jz.



# ASAMBLARE

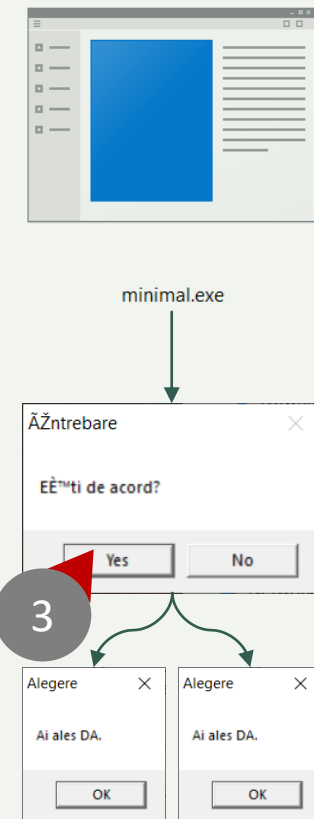
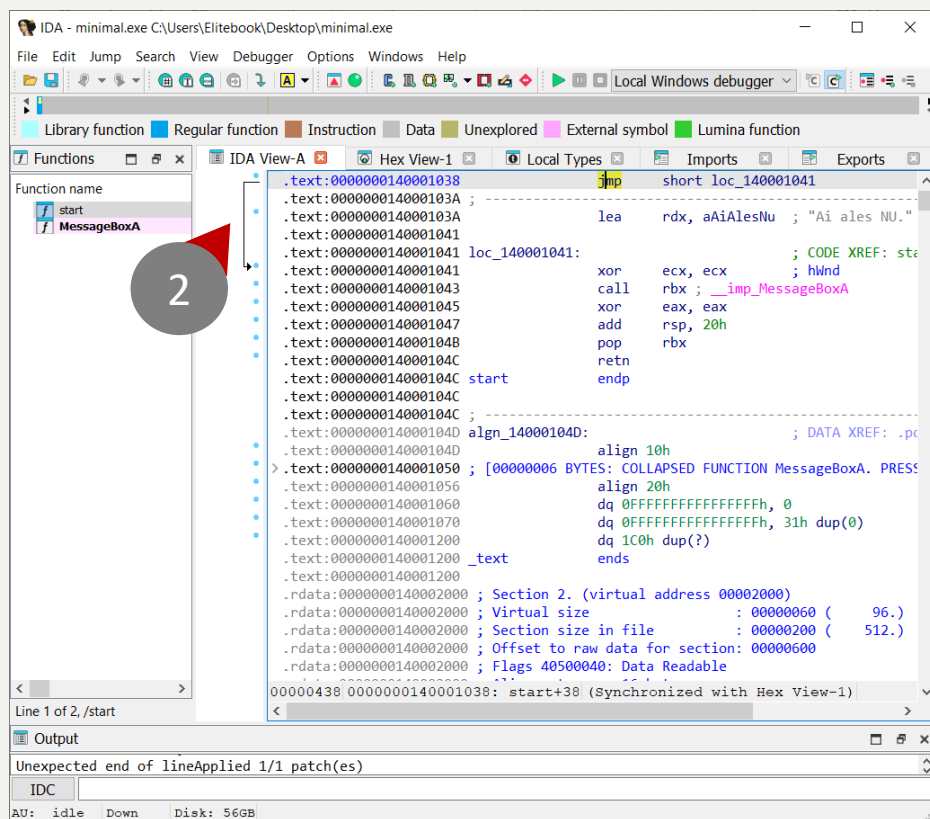
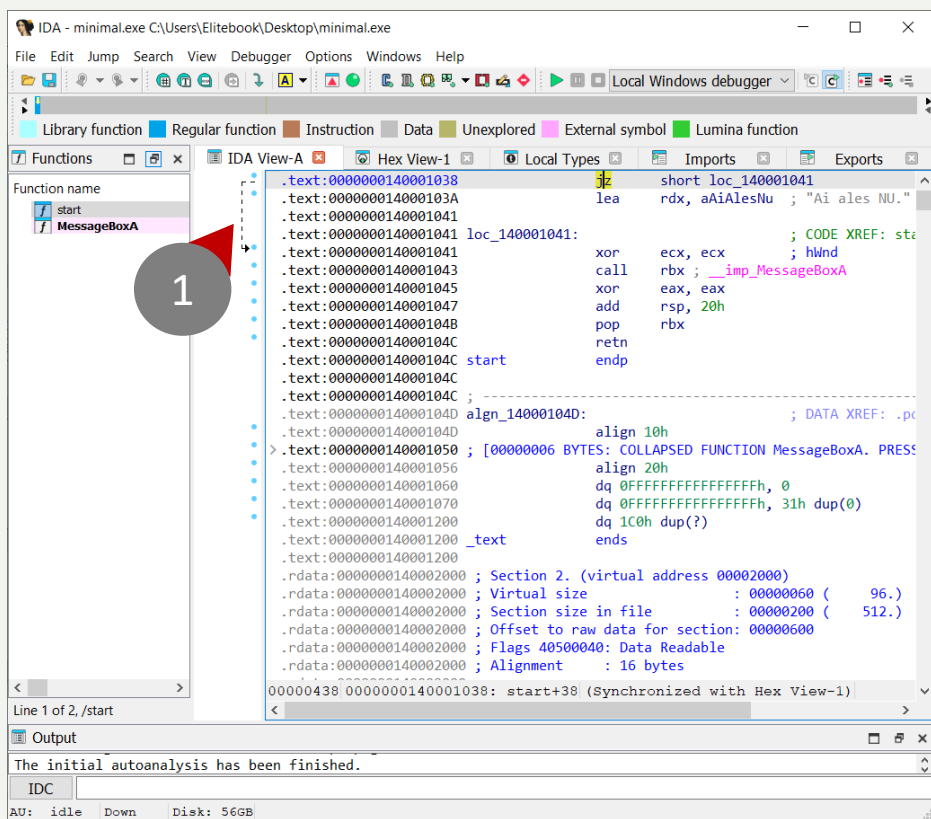
## ÎNLOCUIREA JZ CU JMP

3. Scrie jmp urmat de eticheta sau adresa la care vrei să sară programul.
4. Aplică modificarea și asigură-te că lungimea octeților pentru noua instrucțiune corespunde cu cea veche pentru a menține alinierea codului.
5. Aplică patch-ul la executabil și testează-l pentru a verifica modificările.



# EFECTE DUPĂ ÎNLOCUIREA JZ CU JMP

Indiferent de răspunsul utilizatorului, aplicația ajunge în același punct.



# PE SCURT !

## OCOLIREA CONDIȚIILOR



Puteți înlocui direct instrucțiunea de salt condițional (**jz**) cu un salt necondițional (**jmp**). Acest lucru va cauza ca programul să execute întotdeauna blocul de cod care urmează după eticheta specificată în instrucțiunea de salt, ignorând orice condiție.

1. Localizați instrucțiunea **jz** pe care doriți să o modificați.
2. Deschideți modul de editare de patch-uri în IDA folosind **Edit > Patch program > Assemble** sau prin dublu-click pe instrucțiunea pe care doriți să o schimbați.
3. Scrieți **jmp** urmat de eticheta sau adresa la care doriți să sară programul.
4. Aplicați modificarea și asigurați-vă că lungimea octeților pentru noua instrucțiune corespunde cu cea veche pentru a menține alinierea codului.
5. Aplicați patch-ul la executabil și testați pentru a verifica modificările.

De exemplu, dacă **jz short loc\_140001041** este instrucțiunea originală și doriți să faceți întotdeauna saltul: **jmp short loc\_140001041**

Nu uitați că după ce înlocuiți o instrucțiune de salt **condițional** cu un salt **necondițional**, trebuie să vă asigurați că logica de mai jos este consistentă și nu va cauza efecte secundare.

C.7.6

# DISTRUGEREA FUNCTIONALITĂȚII CU INSTRUCȚIUNI NOP





# TRECEREA CA “PRIN BRÂNZĂ”

## ÎNLOCUIREA **JMP** CU **NOP**

Tip de JMP	Opcode	Bytes	Exemplu
Short jump	EB xx	2	EB 07
Near jump (relativ)	E9 xx xx xx xx	5	E9 12 34 56 00
Absolute indirect	FF 25 xx xx xx xx	6	Folosit pt. IAT, hook-uri



- Dacă primii doi bytes **EB 07** reprezintă codul operațional și offset-ul pentru instrucțiunea **jmp short**, înlocuirea lor cu **90 90** va transforma acea instrucțiune în două operațiuni **NOP** (No Operation), ceea ce efectiv “dezactivează” acea săritură în cod fără a schimba adresele sau fluxul codului ulterior.

- Deci, dacă aveți o secvență originală de bytes care arată ca:

- EB 07 48 8D 15 EE OF 00 00 31 C9 FF D3 31 C0 48**

- și doriți să scoateți **jmp short**, care sunt primii doi bytes, schimbarea arată ca:

- 90 90 48 8D 15 EE OF 00 00 31 C9 FF D3 31 C0 48**

- În acest mod, primele două instrucțiuni care formau **jmp short** vor fi acum **NOP**, iar restul codului va rămâne neschimbat. După această modificare, când programul se execută, va “trece peste” aceste NOP-uri fără efect și va continua execuția cu următoarele instrucțiuni.

Index	Byte	Hex	Instrucțiune
0	1	EB	jmp short (opcode)
1	2	07	offset: +7
2	3	48	← aici ar fi fost lea
3	4	8D	
4	5	15	
5	6	EE	
6	7	OF	
7	8	00	
8	9	00	
9	10	31	● ← AICI aterizează jmp EB 07
10	11	C9	
11	12	FF	
12	13	D3	
13	14	31	
14	15	C0	
15	16	48	

# COMPLETARE

## SHORT JUMP

Cod	Înseamnă	Detalii
EB 07	jmp + offset +7	Sari 7 bytes înainte
EB FE	jmp -2	Bucă infinită ( jmp la sine)
EB 00	jmp +0	Sari la instrucțiunea următoare (inutile)
EB D6	jmp -42	Salt înapoi

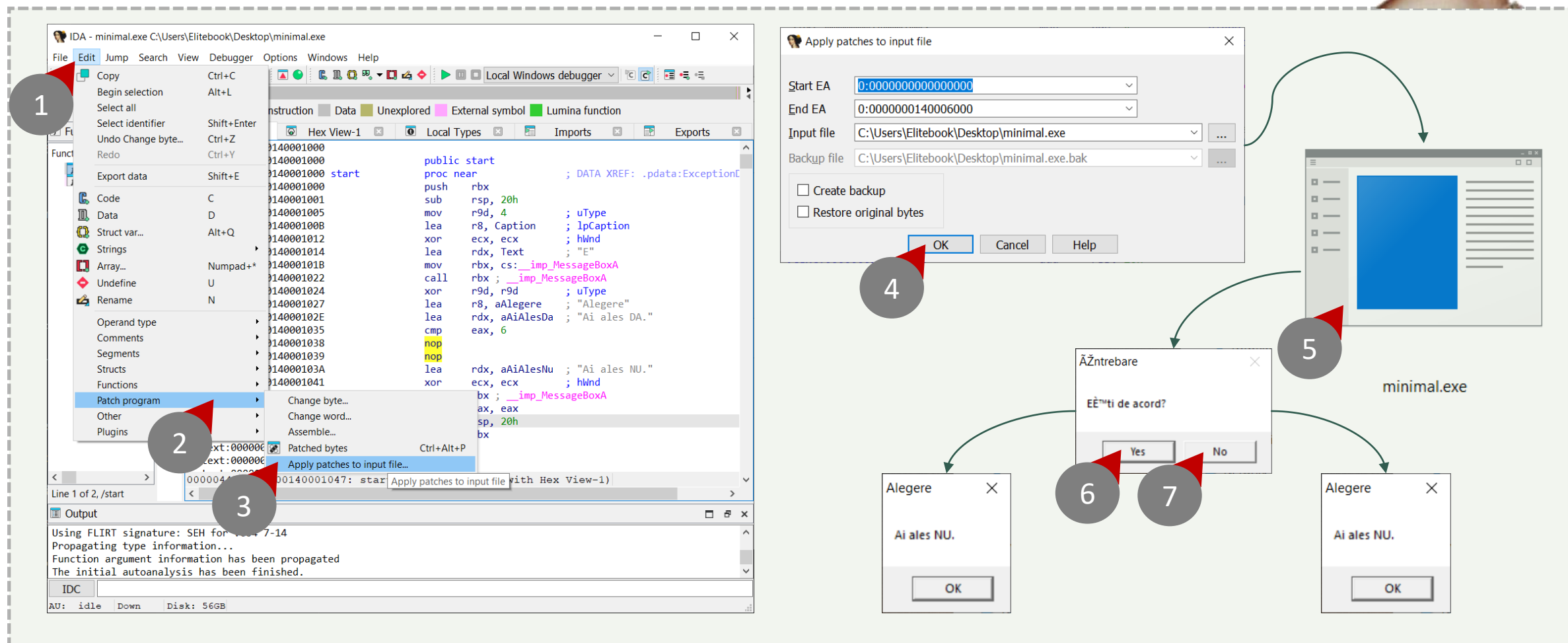


- Codul operațional **EB** urmat de un singur byte de fapt reprezintă o instrucțiune de săritură scurtă (short jump) în limbaj de asamblare x86 și x86-64. Această instrucțiune sare la o etichetă din apropiere (de obicei în cadrul a  $\pm 128$  de bytes de la locul curent), și byte-ul care urmează după **EB** reprezintă distanța (offset-ul) săriturii.
- Pentru a înlocui această săritură scurtă cu **NOP**, aveți nevoie doar de a înlocui **EB** și următorul byte cu **90 90**. Acest lucru va face ca procesorul să execute două instrucțiuni **NOP** în loc de săritura scurtă, lăsând fluxul programului să continue lin la următoarea instrucțiune.
- Dacă avem un **jmp** long (care este de obicei reprezentat de codul operațional **E9** urmat de un offset de 4 bytes), atunci ar trebui să înlocuim toți cei 5 bytes (opcode-ul **E9** plus cei 4 bytes de offset) cu **NOP**.



# PETICIRE SI TESTARE

GLISEAZA LA INSTRUCȚIUNEA CARE SE AFLĂ SUB **JMP**



# EXPLICAȚII

## COD

Dacă 6 este codul pentru un răspuns “Da”, atunci codul care urmează să fie reintrodus ar fi ceva de genul **jne** (sări dacă nu este egal), care ar sări peste încărcarea și afișarea mesajului “Ai ales NU.” dacă utilizatorul a apăsăat “Da”.

**lea rdx, Text** - Încarcă adresa textului pentru primul **MessageBoxA**.

**call rbx ; \_\_imp\_MessageBoxA** - Afișează primul **MessageBoxA** care pune întrebarea utilizatorului.

**cmp eax, 6** - Compară valoarea returnată de primul **MessageBoxA** (presupunând că **eax** este folosit pentru valoarea returnată și **6** reprezintă **IDYES**).

**nop** - Fostul loc unde fusese o instrucțiune de salt condiționată care ar fi sărit peste afișarea mesajului “Ai ales NU.” dacă utilizatorul ar fi selectat “Da”.

**lea rdx, aAiAlesNu** - Încarcă adresa textului “Ai ales NU.” pentru al doilea **MessageBoxA**.

**call rbx ; \_\_imp\_MessageBoxA** - Afișează al doilea **MessageBoxA**, care în acest moment va afișa întotdeauna “Ai ales NU.”, indiferent de opțiunea utilizatorului.



# EXPLICAȚII

## COD

În secvența de cod, observăm că **rdx** este încărcat inițial cu adresa textului “Ai ales DA.” cu instrucțiunea **lea rdx, aAiAlesDa**. Aceasta pregătește argumentul pentru apelul **MessageBoxA**, care este destinat să afișeze mesajul “Ai ales DA.” dacă utilizatorul a apăsă butonul corespunzător în caseta de dialog precedentă.

Totuși, după instrucțiunea **cmp eax, 6**, în locul unde ar fi fost un salt condiționat (jz, jnz, je, jne etc.), sunt două **nop** care efectiv anulează orice săritură și permit execuția să continue secvențial. Astfel, următoarea instrucțiune executată este **lea rdx, aAiAlesNu**, care suprascrive **rdx** cu adresa textului “Ai ales NU.”.

De aceea, pe parcursul execuției, mesajul “Ai ales NU.” este afișat indiferent de răspunsul utilizatorului, deoarece nu există nicio bifurcare a fluxului de execuție pentru a afișa “Ai ales DA.” în funcție de rezultatul comparației **cmp eax, 6**.

Dacă vrei să restaurezi funcționalitatea astfel încât mesajul “Ai ales DA.” să fie afișat când utilizatorul alege o opțiune specifică, va trebui să înlocuiți **nop**-urile cu o instrucțiune de salt care să sară peste încărcarea și afișarea mesajului “Ai ales NU.” dacă condiția este îndeplinită (de exemplu, dacă **eax** este egal cu **6**, presupunând că **6** este valoarea pentru “Da”).

# C.7.7 DETECTAREA UNEI FUNCȚII DE CRIPTARE



# CE ESTE OPERAȚIA XOR ȘI DE CE SE FOLOSEȘTE?

Criptează:  $A \text{ XOR } K \rightarrow C$

Decriptează:  $C \text{ XOR } K \rightarrow A$

Metodă	Tip	Complexitate	Ușor de detectat	Ușor de spart
XOR	Ofuscare	Foarte mică	Da	Da
ROL	Ofuscare	Mică	Nu întotdeauna	Da
AES	Criptare	Foarte mare	Greu	Nu (fără cheie)

Aceasta este una dintre cele mai simple forme de criptare — dar și una folosită frecvent în malware pentru:

- XOR (sau „sau exclusiv”) este o operație logică între doi biți.
- Dacă doi biți sunt egali, rezultatul este 0. Dacă sunt diferiți, rezultatul este 1.
- Folosită în criptare, XOR are o proprietate specială:

Dacă aplicați XOR de două ori cu aceeași cheie, obțineți textul original.

- XOR**: Operație logică simplă (sau-exclusiv); folosită pentru criptare rapidă și ușor reversibilă.
- ROL**: Rotate Left – mută biții spre stânga; utilizat în ofuscarea codului pentru a evita detecția.
- AES**: Advanced Encryption Standard – algoritm criptografic avansat, imposibil de spart fără cheie; folosit în ransomware.
- RSA**: Algoritm criptografic cu cheie publică; utilizat pentru criptarea cheilor AES în atacuri sofisticate.

## Ce face un malware real?

- Textul, stringurile critice sau shellcode-ul sunt de obicei **stocate criptat pe disc**.
- La execuție, malware-ul aplică **decriptare în memorie** (XOR, ROL, AES etc.) și apoi execută sau folosește datele în clar doar temporar.



```

format PE GUI 4.0
entry start

include 'INCLUDE/win32a.inc'

section '.data' data readable writeable
    text    db 'SecretText', 0          ; Textul original
    len      = $ - text                 ; Lungimea șirului
    key      db 0x42                    ; Cheie de criptare XOR
    title    db 'Criptare XOR', 0
    msg      db 'Criptarea a fost efectuată.', 0

section '.code' code readable executable
start:
    mov esi, text                      ; pointer la începutul textului
    mov ecx, len                       ; număr de caractere
    mov al, [key]                      ; cheia de criptare

.encrypt_loop:
    xor [esi], al                      ; XOR pe fiecare caracter
    inc esi
    loop .encrypt_loop

    push 0
    push title
    push msg
    push 0
    call [MessageBoxA]

    push 0
    call [ExitProcess]

section '.idata' import data readable writeable
    library kernel32, 'KERNEL32.DLL', \
            user32, 'USER32.DLL'

    import kernel32, \
            ExitProcess, 'ExitProcess'

    import user32, \
            MessageBoxA, 'MessageBoxA'

```

```

C:\Users\Paul\Desktop\fasm>FASM cript.asm cript.exe
flat assembler version 1.73.32 (1048576 kilobytes memory)
3 passes, 0.2 seconds, 2048 bytes.

C:\Users\Paul\Desktop\fasm>cript.exe

C:\Users\Paul\Desktop\fasm>_

```

## Cum funcționează?

### 1. Declarații de date (.data)

- `text` = șirul de caractere care va fi criptat.
- `len` = lungimea șirului (`SecretText` are 10 caractere).
- `key` = valoarea fixă cu care se face criptarea (0x42).
- `title` și `msg` = textul care va fi afișat în caseta de mesaj.

### 2. Codul principal (.code)

- `mov esi, text` – `esi` va arăta spre primul caracter din `SecretText`.
- `mov ecx, len` – `ecx` este folosit ca un contor, indică de câte ori trebuie să repetăm operația.
- `mov al, [key]` – cheia de criptare este copiată în registrul `al` (folosit pentru XOR).

### 3. Bucla de criptare

- `.encrypt_loop`: este eticheta de început a buclei.
- `xor [esi], al` – se aplică XOR între caracterul curent și cheia (0x42). Rezultatul este o versiune criptată a caracterului.
- `inc esi` – se trece la următorul caracter.
- `loop .encrypt_loop` – se repetă pașii de mai sus până când `ecx` ajunge la 0.

### 4. Afișarea rezultatului

- Se apelează `MessageBoxA` pentru a afișa o fereastră cu mesajul „Criptarea a fost efectuată.” și titlul „Criptare XOR”.
- După închidere, se apelează `ExitProcess` pentru a încheia programul.

# DE CE ESTE IMPORTANT ACEST EXEMPLU?

## Ce face exemplul ?

- Textul (SecretText) este stocat **în clar** în secțiunea .data a executabilului.
- La rulare, programul aplică **criptare în memorie**, modificând acest text.
- Este un exemplu construit **pentru a demonstra criptarea XOR în IDA**, într-un mod simplu și vizibil.

Aceasta este una dintre cele mai simple forme de criptare, dar și una folosită frecvent în malware pentru:

- a ascunde stringuri de analiză statică;
- a cripta shellcode injectat;
- a ocoli semnăturile antivirus.

## Observație didactică

**Acest comportament este similar cu cel folosit de unele aplicații malware**, care păstrează stringurile sensibile în clar în fișier, dar le criptează imediat după lansare, pentru a evita detectarea statică de către antivirus sau YARA.

**Textul este salvat în clar pe disc, în secțiunea .data a executabilului.**

La rulare, programul aplică o criptare de tip XOR în memorie, modificând conținutul inițial al șirului text.

Astfel, **după criptare**, șirul original nu mai este vizibil în clar în memorie, ci apare ca o succesiune de caractere ilizibile.



# EXEMPLU FASM

## DECOMPILARE

În acest exemplu, o buclă scurtă efectuează o criptare de tip **XOR** asupra unui șir de caractere stocat în secțiunea **.data**. Instrucțiunea **XOR [ESI], AL** este aplicată fiecărui caracter, folosind cheia **0X42**, iar pointerul **esi** este incrementat pe parcursul buclei. Valoarea **ecx** controlează numărul de caractere procesate.

Bucula este ușor de identificat în *Graph View*, unde fluxul de execuție este clar delimitat: codul de criptare este izolat într-un bloc repetitiv, iar după terminarea sa, programul afișează o casetă de mesaj cu *MessageBoxA*.

Astfel de bucle de criptare apar frecvent în fișiere malware, fiind utilizate pentru:

- ascunderea stringurilor sensibile,
- ofuscarea shellcode-ului,
- evitarea detecției statice.

IDA permite recunoașterea rapidă a acestor modele prin analiza structurii control-flow și a secvențelor caracteristice de instrucțiuni precum **XOR**, **LOOP**, **MOV**, **INC**.



# CE SE ÎNTÂMPLĂ DACĂ TEXTUL ESTE MAI LUNG DECÂT 8 BYTES?

- În codul actual, lungimea textului este determinată automat de această linie:
- `len = $ - text`
- Aceasta înseamnă: `len = (adresă curentă) - (adresă text)`, adică numărul de octeți din șirul text.
- Dacă text conține mai mult de 8 caractere (de exemplu db 'SuperSecretText123', 0), valoarea lui len va fi automat actualizată, iar bucla loop va cripta toate caracterele, nu doar 8.
- Deci nu e o problemă, codul este scris corect ca să funcționeze cu orice lungime de text. Numărul 08h pe care l-ai văzut în IDA era generat automat pentru lungimea exactă a stringului inițial („SecretText” are 10 caractere, dar poate \0 nu a fost inclus sau s-a tăiat la 8 din alte motive în exemplu).

Ce se întâmplă dacă `al = 0`?

Dacă registrul `al` conține valoarea 0, atunci operația:

- `xor [esi], al`
- va deveni echivalentă cu:
- `xor [esi], 0`

Iar orice valoare XOR 0 = valoarea originală. Așadar, în acest caz nu se produce nicio criptare: Textul rămâne nemodificat în memorie.

Dacă punem key db 0x00, deci `al` devine zero, și `xor [esi], al` devine inofensiv.”

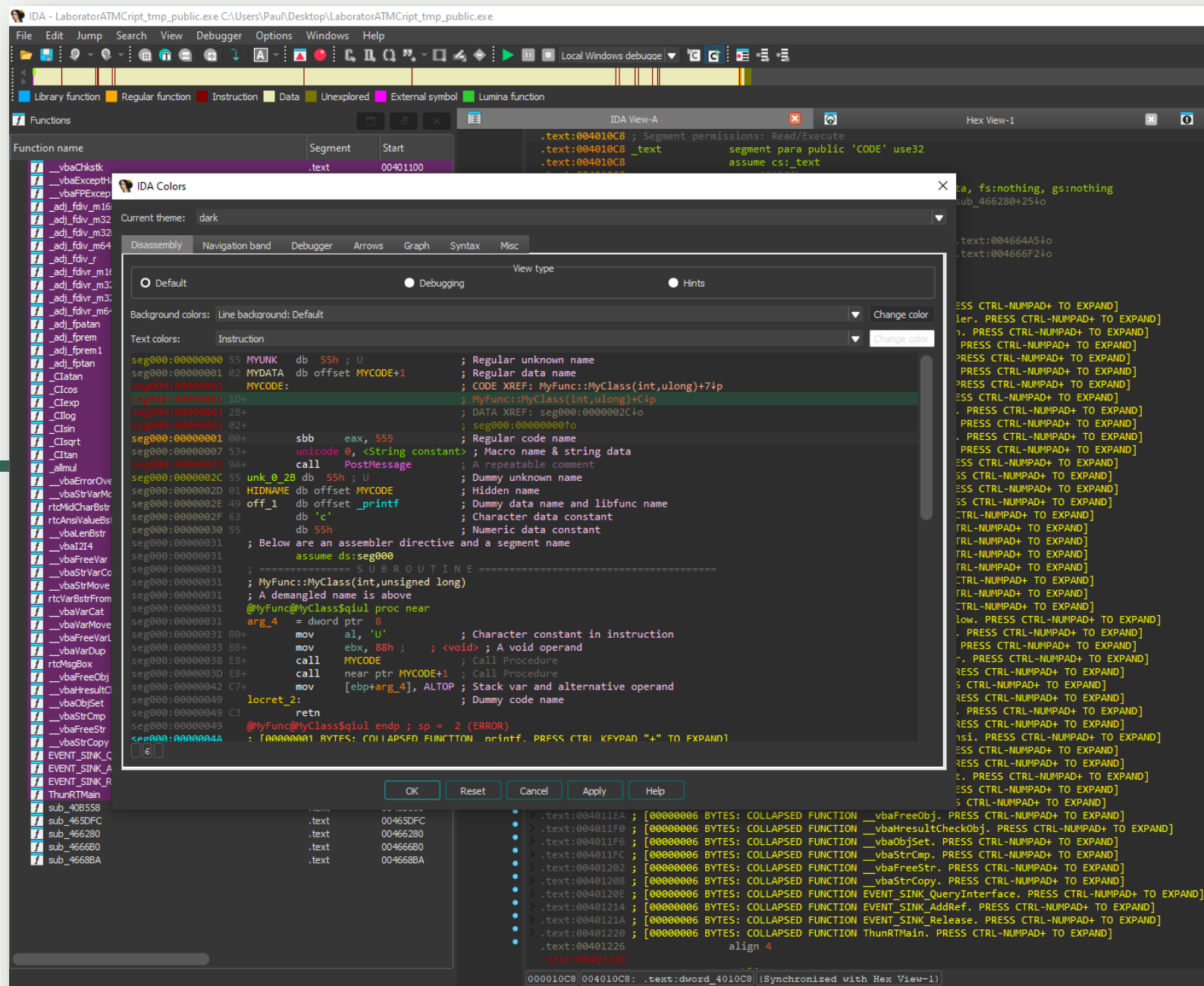
# C.7.8 IDA & INGINERIA INVERSĂ DE COD MALWARE

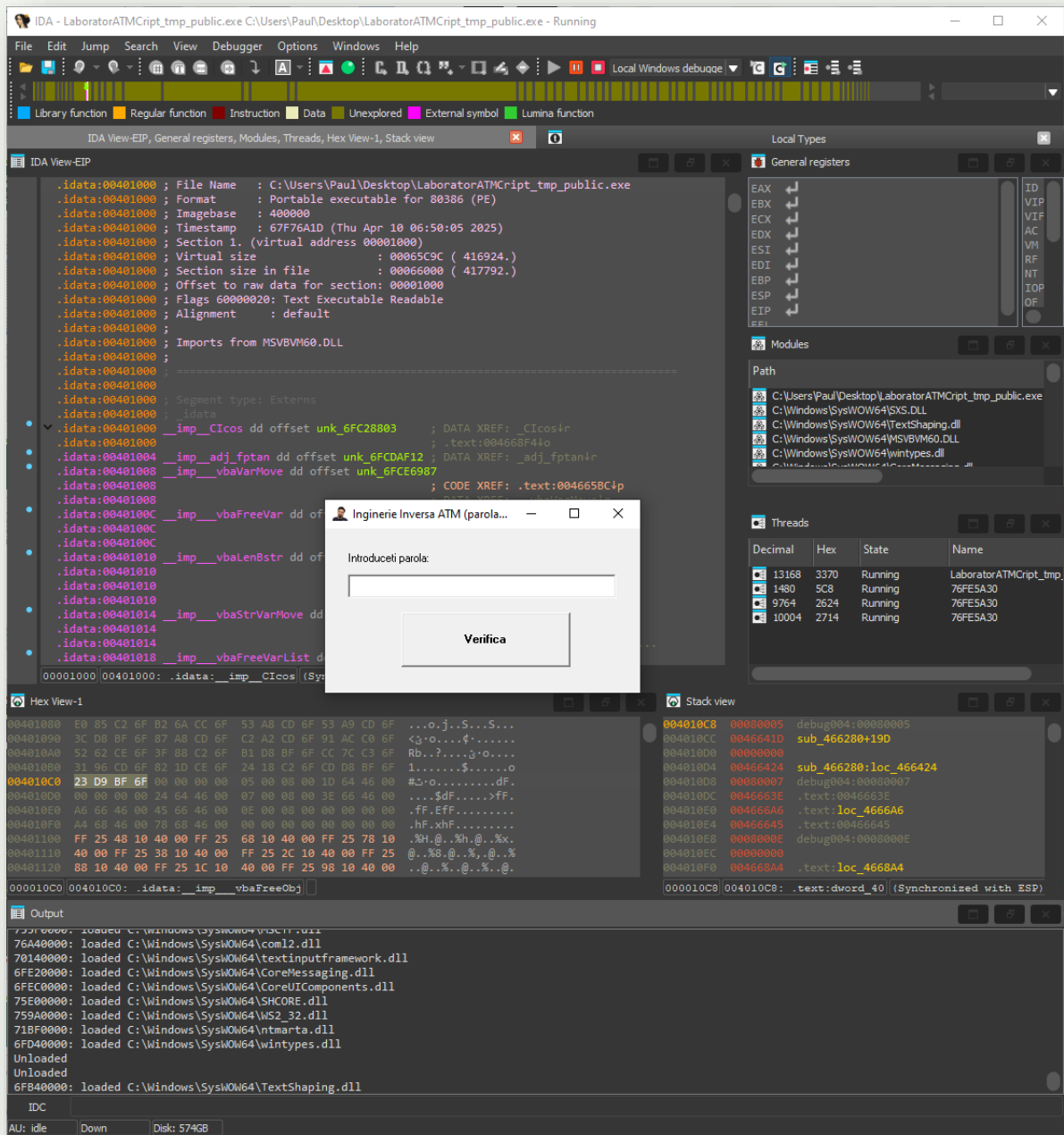


Pentru a îmbunătăți lizibilitatea și a reduce oboseala vizuală, IDA oferă posibilitatea de a personaliza culorile interfeței. Următorii pași permit comutarea la o temă întunecată:

### Pași pentru activarea modului Dark

1. Mergem în bara de sus la: **Options** → **Colors...**
2. Se deschide fereastra „IDA Colors”, unde putem selecta:
  - **Theme**: alege Dark background
3. Pentru stiluri personalizate, se poate ajusta manual fiecare categorie de culoare:
  - Instrucțiuni, comentarii, constante, simboluri, etc.
4. Apăsăm **Apply**, apoi **OK** pentru a salva și ieși.





În cazul executabilelor VB6, logica programului nu este prezentă ca funcții x86 clasice. Ea este distribuită între runtime-ul VB (msvbvm60.dll) și codul de apel, iar analiza trebuie să combine pași statici (IDA) și dinamici (debugging în timp real).

### 1. Bara de unelte (sus)

- Conține butoane pentru:
  - rulare (F9),
  - pas cu pas (F7, F8),
  - comutare între IDA View, Hex View, Graph View, și setare de breakpoint (Ctrl+B).
- Indicatori colorați arată starea segmentelor (cod, date, simboluri externe etc.).

### 3. Hex View-1 (jos stânga)

- Vizualizare în format hexazecimal a memoriei, sincronizată cu instrucțiunile ASM.
- Permite identificarea exactă a valorilor binare și a stringurilor stocate în secțiuni.

### 2. IDA View-EIP (stânga sus)

- Afișează conținutul segmentului .idata, cu focus pe importurile din MSVBVM60.DLL (runtime-ul VB6).
- Funcții precum \_\_vbaVarMove, \_\_vbaStrVarMove sunt esențiale pentru execuția codului VB6.

### 4. Stack View (dreapta jos)

- Afișează stiva în momentul execuției, cu funcțiile apelate și parametrii lor.
- Util pentru urmărirea apelurilor în lanț (call stack) și identificarea locului în care este procesată parola.

### 5. Modules și Threads (dreapta mijloc)

- **Modules** listează toate DLL-urile încărcate în proces, inclusiv MSVBVM60.DLL, user32.dll, kernel32.dll etc.
- **Threads** arată thread-urile active ale aplicației, important în cazul aplicațiilor multi-threaded.



## 1. Meniul „Open subviews” (View → Open subviews)

• Acesta este centrul de comandă pentru toate subferestrele de analiză:

- Functions, este lista tuturor funcțiilor recunoscute.
- Imports / Exports – vizualizarea dependențelor și a punctelor de intrare.
- Threads, Segments, Structures, Strings, Local types – acces la componente interne ale fișierului executabil.
- Cross references (xrefs) – permite să vezi unde și cum este folosită o variabilă, un string sau o funcție.

## 2. Hex View-1 (jos stânga)

- Afișează conținutul binar brut al programului.
- Săritura este sincronizată cu zona de cod: când navighezi în ASM, vezi și codul sursă hexazecimal.

## 3. Stack View (jos dreapta)

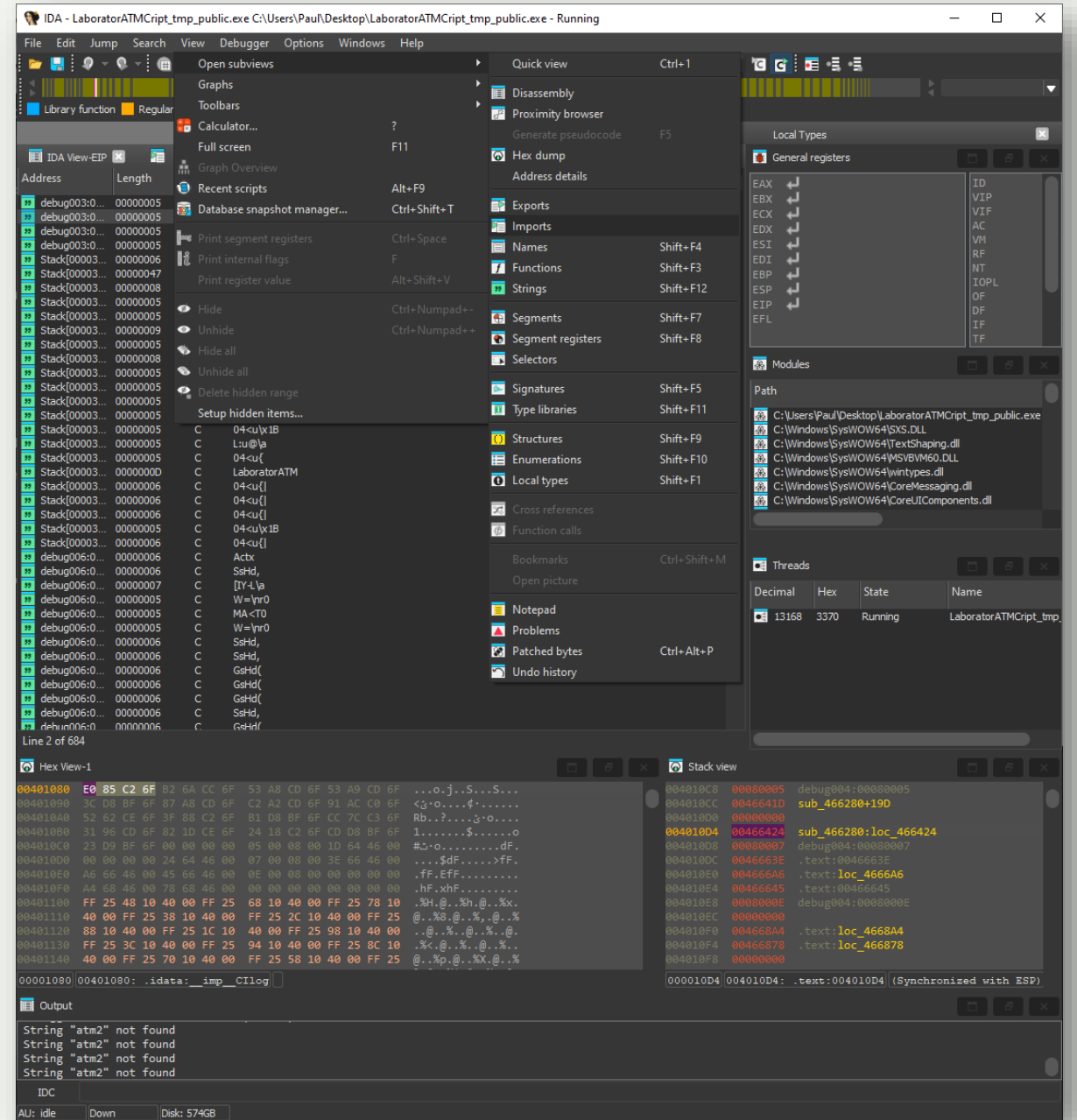
- Vizualizarea în timp real a stivei în timpul execuției.
- Permite identificarea pașilor de apel (call stack) și înțelegerea fluxului de execuție.

## 4. Fereastra „Output” (jos de tot)

- Afișează rezultate ale comenzilor IDC, căutări, erori și alerte.
- În această captură se observă eșecul căutării stringului „atm2”.

## 5. General Registers & Threads / Modules (dreapta sus)

- Registrele procesorului sunt actualizate în timp real când se rulează programul.
- Modules: toate DLL-urile încărcate, inclusiv MSVBVM60.DLL necesar în executabilele VB6.
- Threads: starea curentă a thread-urilor din aplicație (important pentru analiză concurentă sau GUI blocking).



# CONCLUZIE



- IDA este în primul rând un dezasamblor static, dar are și funcționalități de debugging dinamic, astfel că poate fi folosit atât pentru analiză statică, cât și pentru analiză dinamică, dar cu unele limitări, mai ales în versiunea gratuită.

## 1. Dezasamblor (Static Analysis) – partea principală din IDA

- la un fișier binar (ex: .exe, .dll, .bin) și îl convertește în:
  - instrucțiuni de asamblare (mov, cmp, jmp, etc.),
  - grafuri de control (Graph View),
  - structuri (Imports, Strings, Segments, etc.).
- **Nu rulează programul** ci doar îl analizează ca text.
- Putem face:
  - reconstrucție de flux logic,
  - redenumire de funcții și variabile,
  - analiză a secțiunilor, semnăturilor, constante etc.

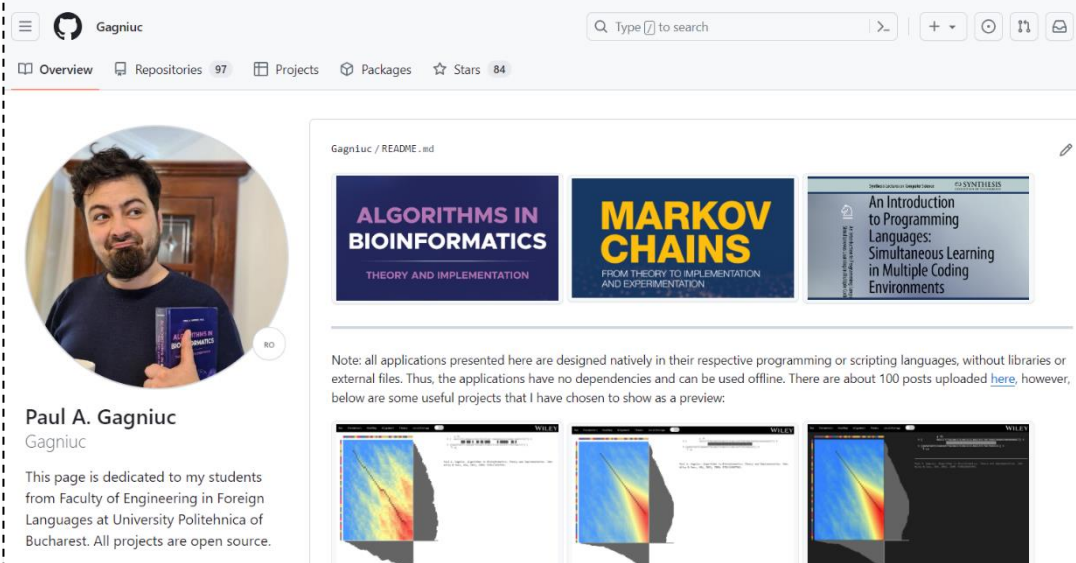
## 2. Debugger (Dynamic Analysis) – opțional în IDA

- Rulează executabilul într-un **mediu controlat** și îți arată:
  - **registre** în timp real (EAX, ECX...),
  - **stivă** (ESP, call stack),
  - **memorie modificată**,
  - **executarea pas cu pas** (F7, F8),
  - **breakpoint-uri** (pe instrucții, funcții, imports).
- Este compatibil cu:
  - Windows executables (x86/x64),
  - GDB pentru Linux,
  - Android, iOS (în versiunea Pro).
- **Limitări:**
  - Versiunea Free are funcționalități dinamice limitate.
  - Debugging avansat (ex: kernel-mode, trace-uri automate) e disponibil doar în IDA Pro + driver.

# BIBLIOGRAFIE / RESURSE

- Paul A. Gagniuc. *Antivirus Engines: From Methods to Innovations, Design, and Applications*. Cambridge, MA: Elsevier Syngress, 2024. pp. 1-656.
- Paul A. Gagniuc. *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments. Synthesis Lectures on Computer Science*. Springer International Publishing, 2023, pp. 1-280.
- Paul A. Gagniuc. *Coding Examples from Simple to Complex - Applications in MATLAB*, Springer, 2024, pp. 1-255.
- Paul A. Gagniuc. *Coding Examples from Simple to Complex - Applications in Python*, Springer, 2024, pp. 1-245.
- Paul A. Gagniuc. *Coding Examples from Simple to Complex - Applications in Javascript*, Springer, 2024, pp. 1-240.
- Paul A. Gagniuc. *Markov chains: from theory to implementation and experimentation*. Hoboken, NJ, John Wiley & Sons, USA, 2017, ISBN: 978-1-119-38755-8.

<https://github.com/gagniuc>



The screenshot shows the GitHub profile of Gagniuc. The header includes the username 'Gagniuc', a search bar, and navigation links for Overview, Repositories (97), Projects, Packages, Stars (84), and a notification bell. The profile picture shows a man with a beard holding a book. Below the profile picture, the name 'Paul A. Gagniuc' and the bio 'This page is dedicated to my students from Faculty of Engineering in Foreign Languages at University Politehnica of Bucharest. All projects are open source.' are visible. The main content area displays a 'README.md' file with three book covers: 'ALGORITHMS IN BIOINFORMATICS', 'MARKOV CHAINS', and 'An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments'. A note below the covers states: 'Note: all applications presented here are designed natively in their respective programming or scripting languages, without libraries or external files. Thus, the applications have no dependencies and can be used offline. There are about 100 posts uploaded [here](#), however, below are some useful projects that I have chosen to show as a preview.' At the bottom, three small images of heatmaps or data visualizations are shown.