C.8 DEZASAMBLARE ȘI PATCHING CU X96DBG

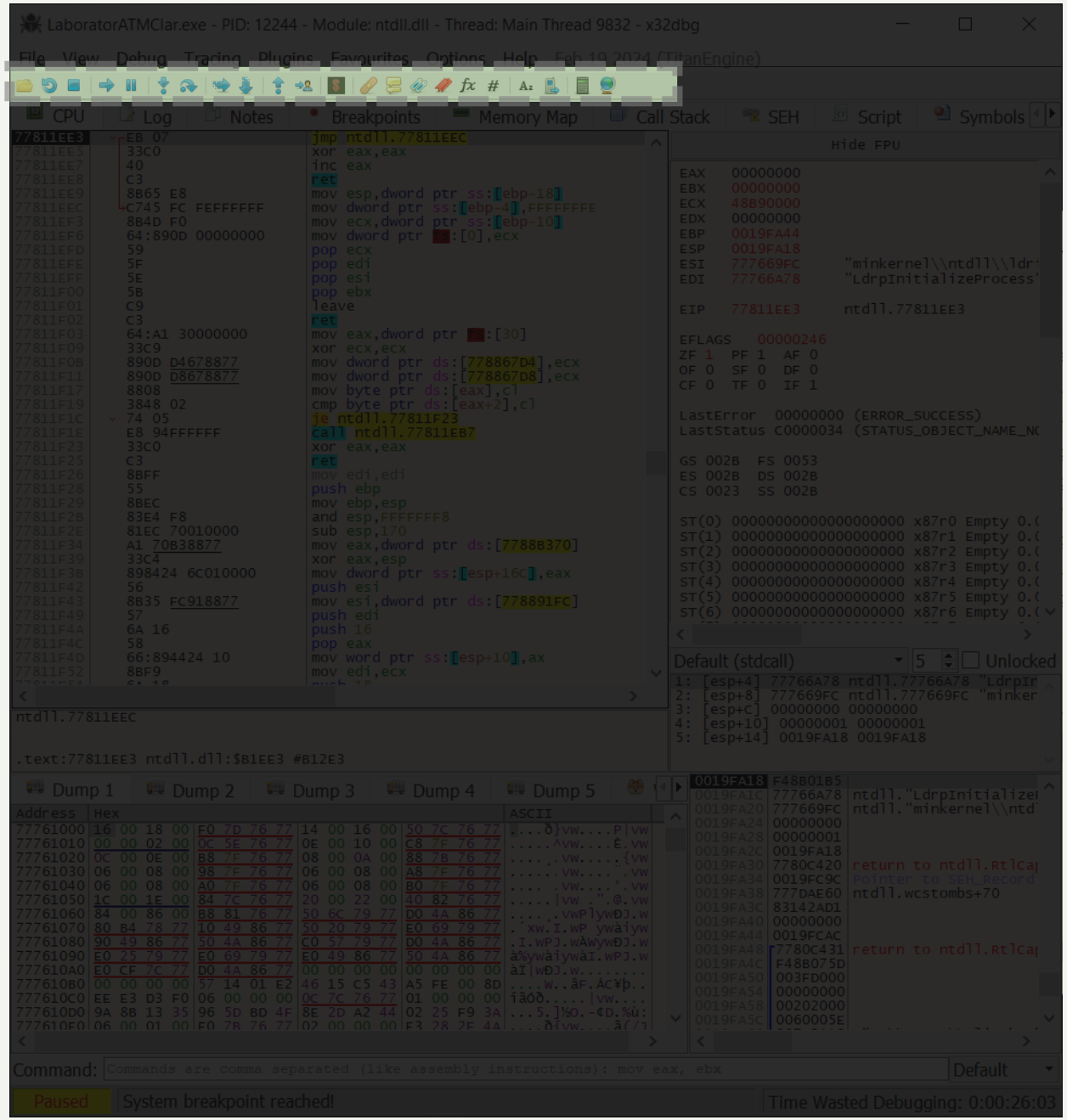
PAUL A. GAGNIUC

Academia Tehnică Militară „Ferdinand I”

PRINCIPALELE PĂRȚI ALE PREZENTĂRII

C.8 Dezasamblare și Patching cu **X32dbg / X64dbg**:

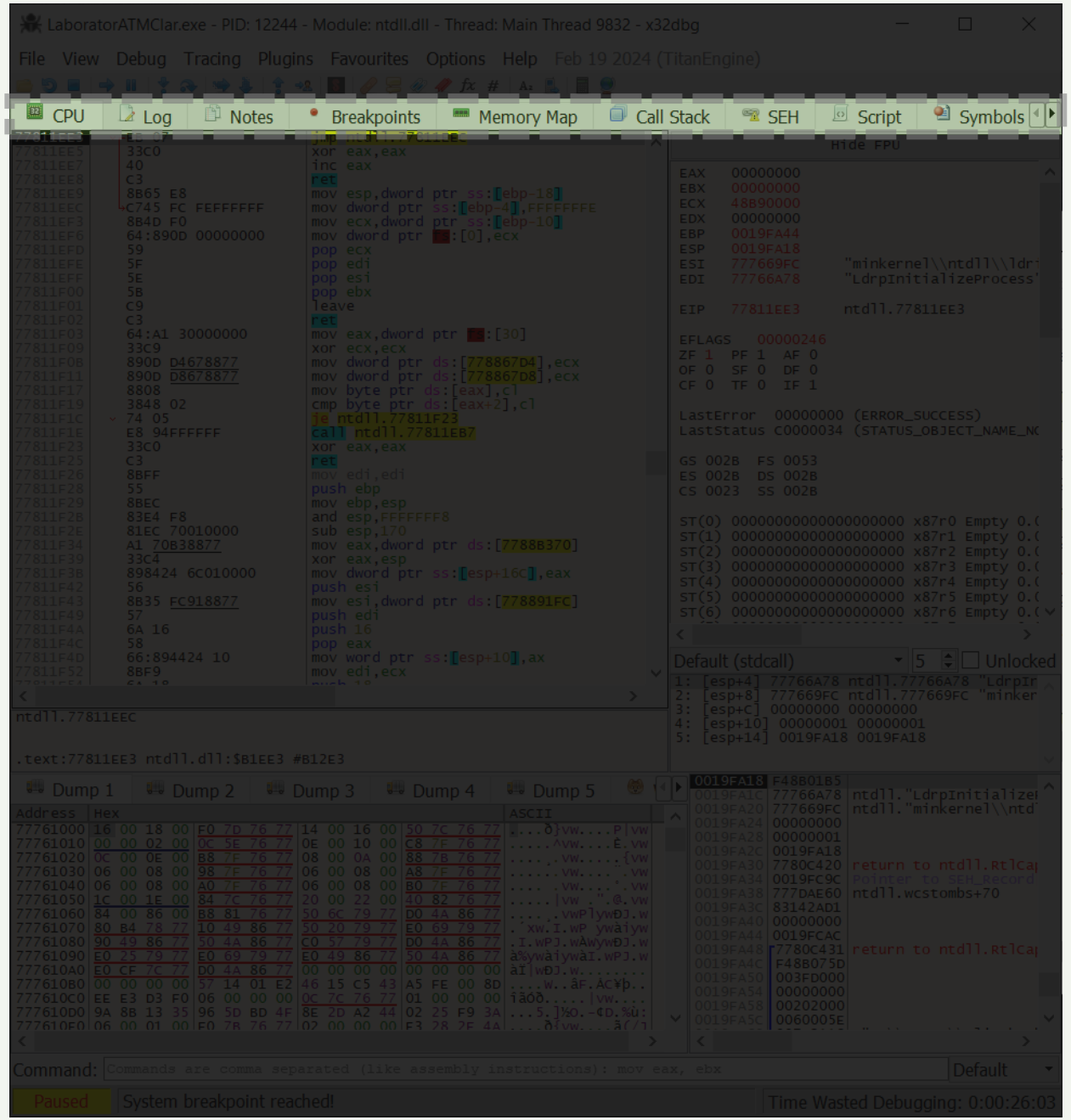
- C.8.1 IDENTIFICAREA PAROLELOR ASCUNSE IN EXECUTABILE
- C.8.2 ADĂUGAREA DE COD MAȘINĂ
- C.8.3 MODIFICARE ENTRY POINT ȘI INSERȚIE DE COD MAȘINĂ
- C.8.4 ADAUGAREA DE STRUCTURI DE DATE IN EXECUTABILE
- C.8.5 MODIFICĂRI DE ADRESE CĂTRE ALTE ARGUMENTE



X64dbg User Interface

Play Controls

Bara de meniu oferă acces la opțiuni avansate de configurare, controlul execuției, încărcarea sau salvarea de sesiuni, integrarea de pluginuri și personalizarea mediului de depanare. Rolul ei este de a centraliza și structura toate comenzile disponibile, servind drept punct de plecare pentru orice acțiune pe care utilizatorul dorește să o efectueze asupra executabilului analizat.

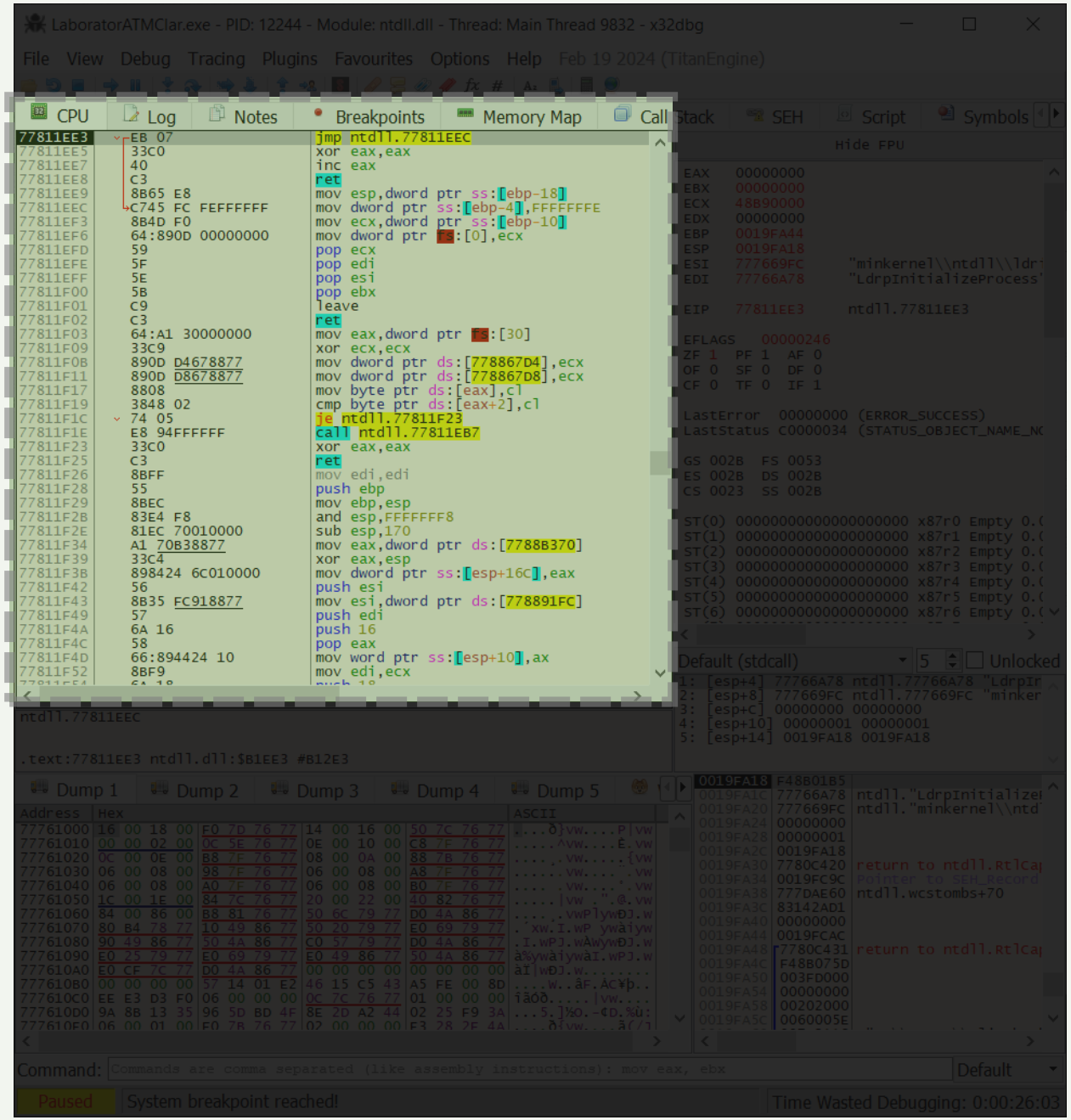


X64dbg User Interface

View Tabs

Taburile ofera acces rapid la funcțiile esențiale ale depanatorului, precum vizualizarea CPU, log-urile de execuție, notițele, gestionarea breakpoints, harta de memorie, stiva de apeluri (Call Stack), gestionarea excepțiilor (SEH), rularea scripturilor și afișarea simbolurilor.

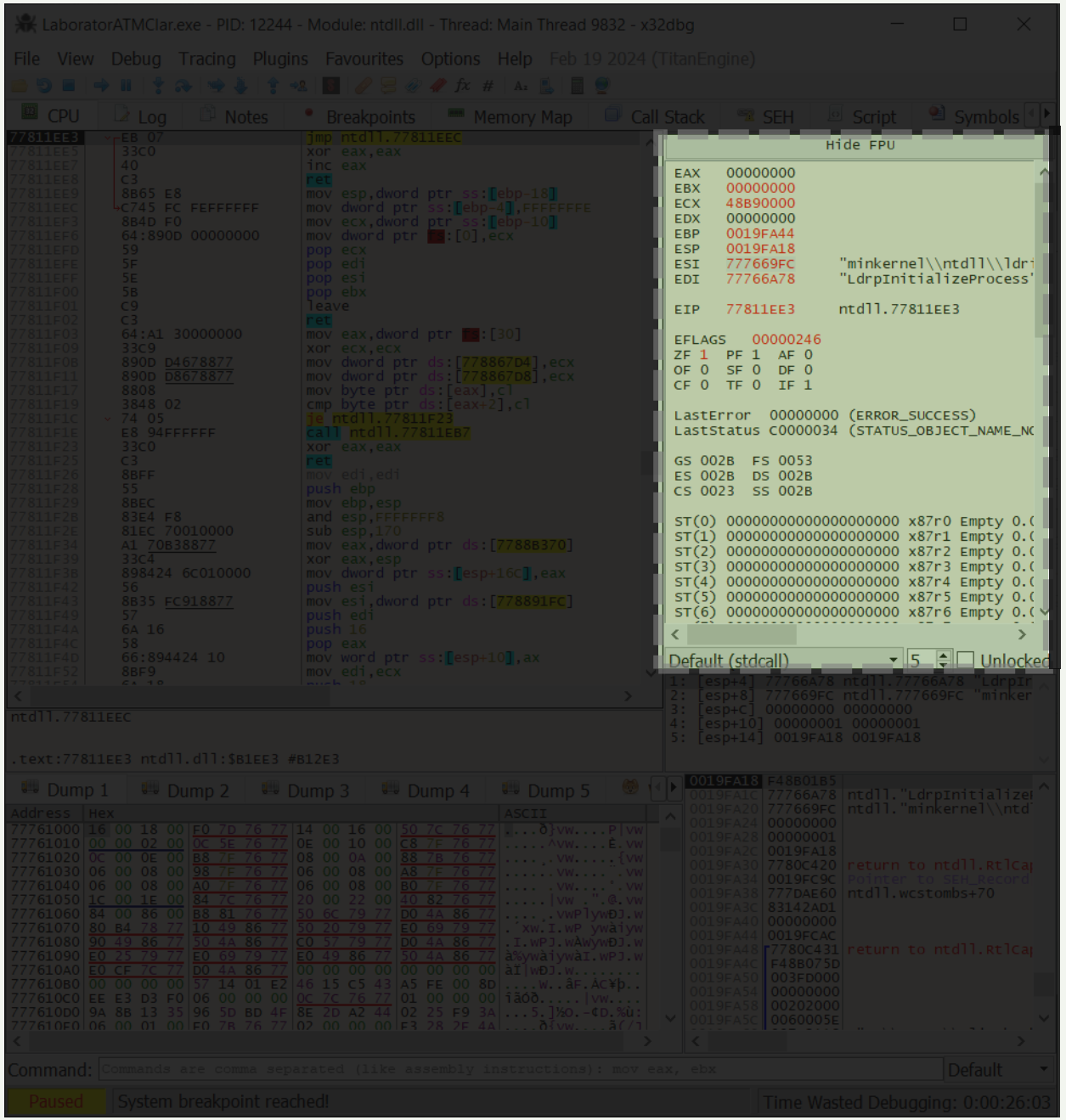
Practic, aceste taburi controlează ce tip de informații sunt afișate în fereastra de lucru, iar utilizatorul poate comuta între ele pentru a analiza diferite aspecte ale executabilului. Bara de instrumente are rolul de a centraliza principalele unelte de analiză, oferind o navigare rapidă și structurată prin componentele procesului aflat în execuție.



X64dbg User Interface

CPU Assembly

Aici se afla fereastra principală de disassembly. Rolul ei este să arate instrucțiunile pe care procesorul urmează să le execute, traduse din cod mașină în limbaj de asamblare. Utilizatorul poate urmări pas cu pas cum se execută programul, poate seta breakpoints, poate vedea fluxul de execuție și apelurile de funcții. Practic, această fereastră este „inima” analizării unui executabil, deoarece afișează direct logica internă a programului așa cum este interpretată de procesor.



X64dbg User Interface

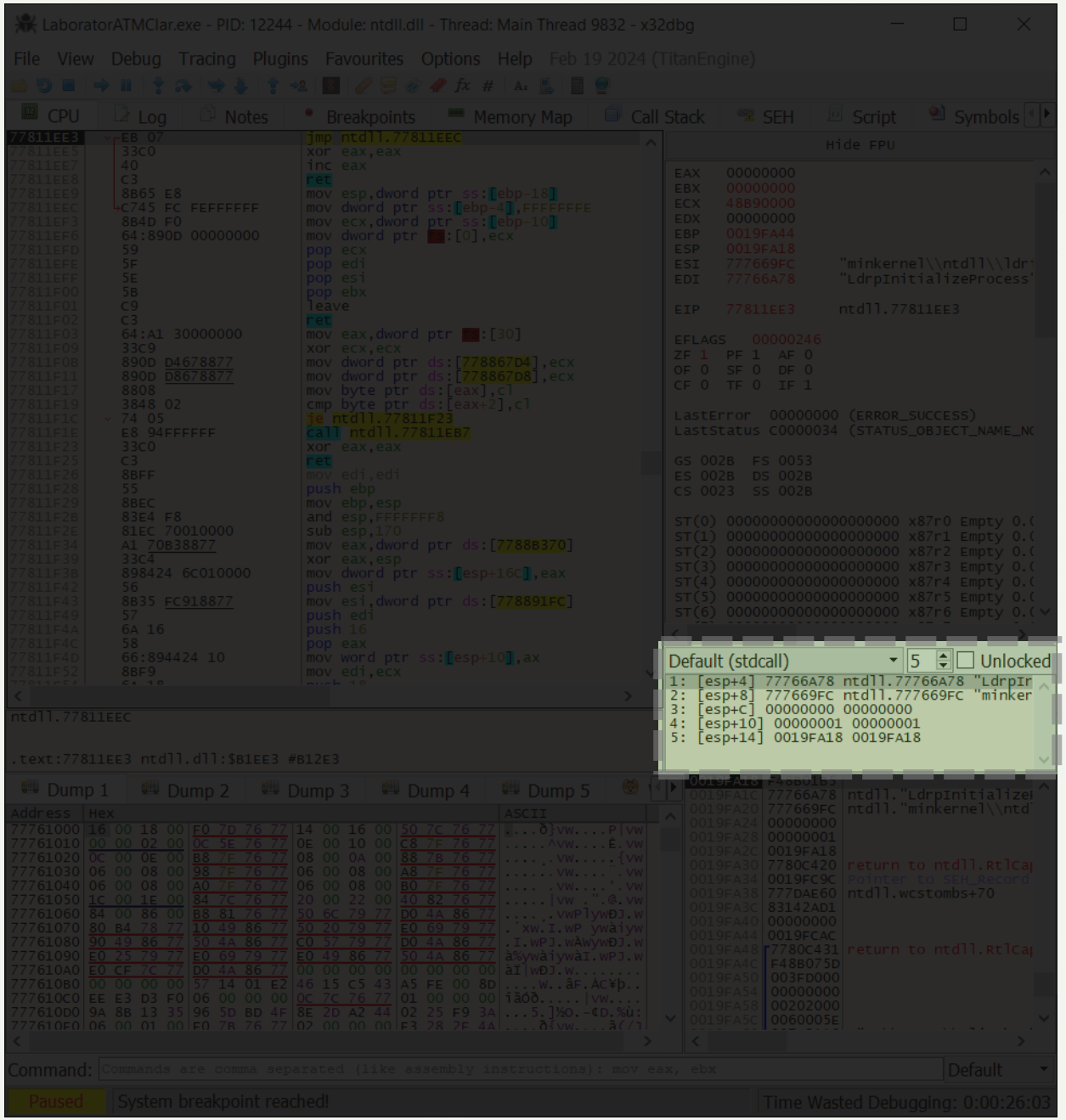
Registers

Rolul acestei zone este să ofere o imagine completă asupra execuției programului la nivel de procesor. Aici se văd valorile regiștrilor și ale flag-urilor procesorului, care se actualizează după fiecare instrucțiune executată. În partea de jos sunt afișate informații despre stivă, inclusiv adresele de retur și parametrii funcțiilor. Practic, aceasta zonă este nucleul analizei: combină instrucțiunile, starea regiștrilor și conținutul stivei pentru a oferi utilizatorului control și vizibilitate totală asupra execuției unui fișier executabil.

Legenda:
- **EAX:** Math and Return Values
- **EBX:** Base index (for use with arrays)
- **ECX:** Counter
- **EDX:** Math
- **EBP:** Base Pointer
- **ESP:** Stack Pointer
- **ESI/EDI:** Memory Transfer

Valorile precum **GS, ES, FS, DS, CS, SS** sunt **registre de segment** din arhitectura x86/x86-64.
- **CS (Code Segment)** indică segmentul unde se află codul executabil.
- **DS (Data Segment)** segmentul implicit pentru date.
- **SS (Stack Segment)** segmentul folosit pentru stivă.
- **ES (Extra Segment)** segment suplimentar pentru operații pe date.
- **FS / GS** registrii de segment speciali, folositi de sistemul de operare pentru structuri interne (în Windows, FS și GS sunt utilizate pentru Thread Environment Block (TEB) și alte date legate de thread/proces).

ZF (Zero Flag) setat la 1 dacă rezultatul unei operații este zero.
PF (Parity Flag) indică dacă rezultatul are un număr par de biți setați la 1.
AF (Adjust Flag) folosit pentru operații aritmetice pe 4 biți (ex. BCD).
OF (Overflow Flag) semnalează depășirea limitei la operații pe numere semnate.
SF (Sign Flag) reflectă semnul rezultatului (1 = negativ).
DF (Direction Flag) determină direcția operațiilor pe șiruri (0 = înainte, 1 = înapoi).
CF (Carry Flag) semnalează un transport sau un împrumut la operații aritmetice.
TF (Trap Flag) dacă e setat, activează execuția pas-cu-pas (debugging).
IF (Interrupt Flag) controlează dacă întreruperile hardware sunt permise (1 = activat).

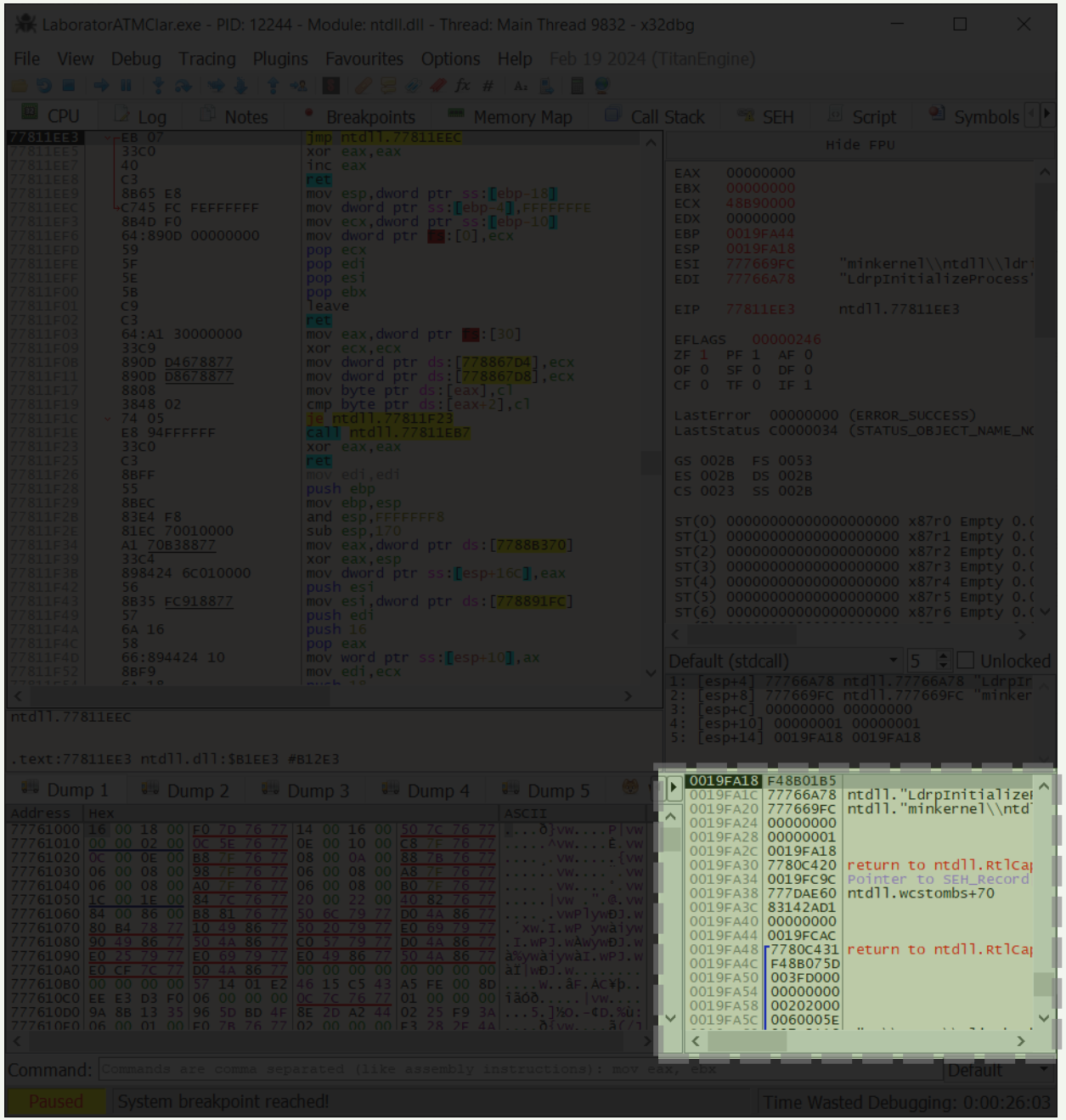


X64dbg User Interface

Stack Window

Conținut brut al stivei (argumente, valori locale)

Această fereastră afișează conținutul stivei procesului, adică zona de memorie unde sunt plasate adrese de retur, variabile locale și parametrii funcțiilor apelate. Fiecare linie corespunde unei adrese din stivă, iar valorile afișate permit urmărirea succesiunii apelurilor și a modului în care programul gestionează execuția funcțiilor. În analiza unui executabil, fereastra Stack este esențială pentru a înțelege cum se transmit parametrii, cum se realizează salturile în cod și pentru a identifica eventuale suprascrieri de memorie sau exploatări de tip buffer overflow.

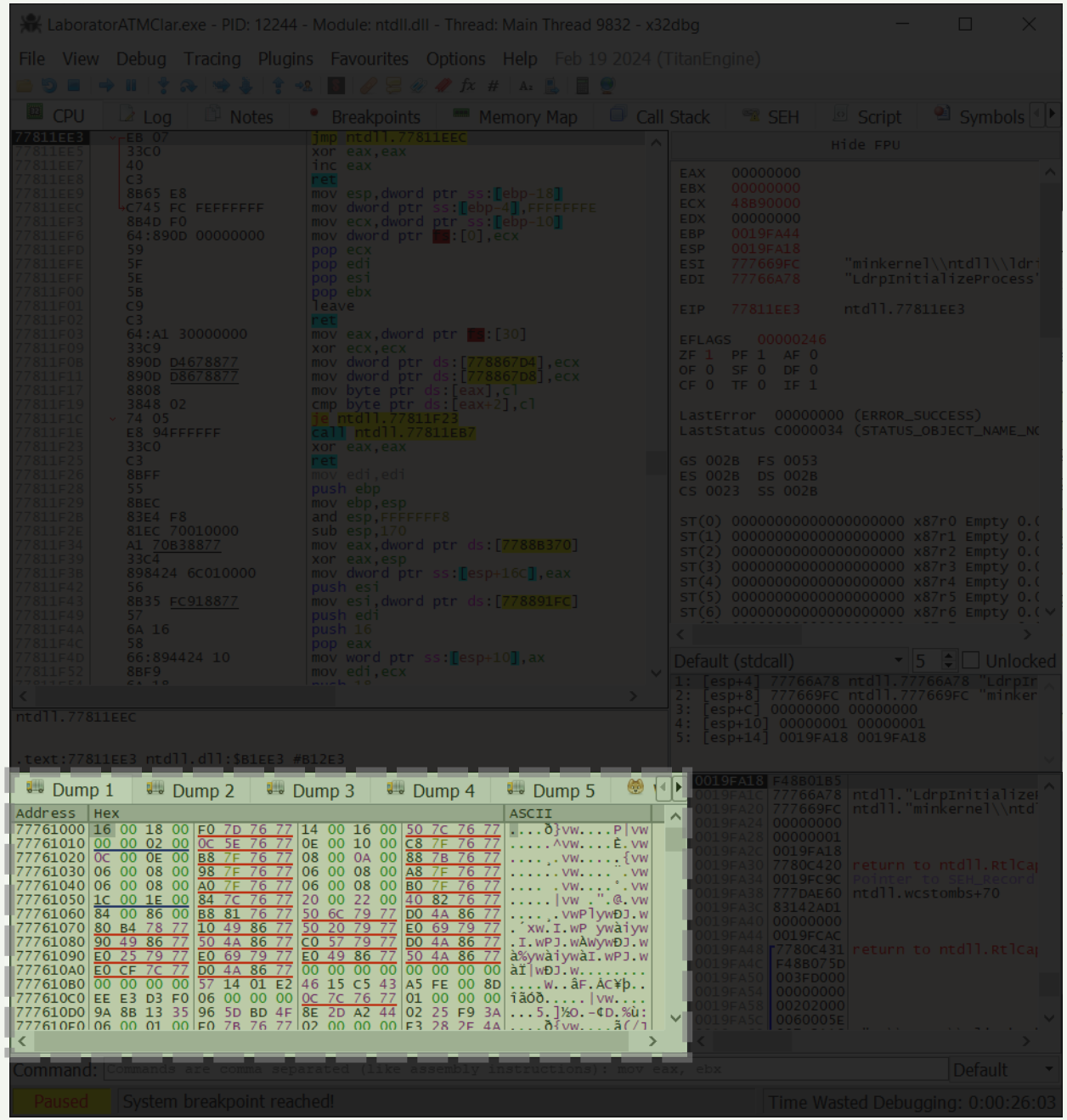


X64dbg User Interface

Call Stack

Lista apelurilor de funcții și adresele de return.

Această zonă arată succesiunea apelurilor de funcții efectuate până la punctul curent de execuție. Fiecare linie indică o adresă de retur și, atunci când este posibil, numele funcției sau modulului asociat. Call Stack-ul este util pentru a înțelege contextul în care se află programul, ordinea apelurilor și pentru a reconstrui traseul execuției. În analiza executabilelor, această vizualizare este esențială pentru depistarea funcțiilor critice, urmărirea fluxului logic și identificarea eventualelor vulnerabilități ce țin de gestionarea apelurilor și a memoriei.



X64dbg User Interface

Dumps (Heap)

Zona de Dump, este cea mai importantă zonă pentru inginerie inversă! Aici se afișează conținutul memoriei procesului sub două forme: în stânga valorile hexazecimale, iar în dreapta echivalentul ASCII acolo unde este posibil. Fereastra permite utilizatorului să vizualizeze și să inspecteze datele încărcate în anumite adrese, cum ar fi șiruri de caractere, pointeri, structuri sau instrucțiuni codificate. În analiza unui executabil, Dump-ul este esențial pentru a observa direct cum sunt stocate și manipulate datele în memorie, fiind un instrument indispensabil în ingineria inversă și detectarea tehnicilor de obfuscare sau criptare.

C.8.1

IDENTIFICAREA PAROLELOR ASCUNSE IN EXECUTABILE

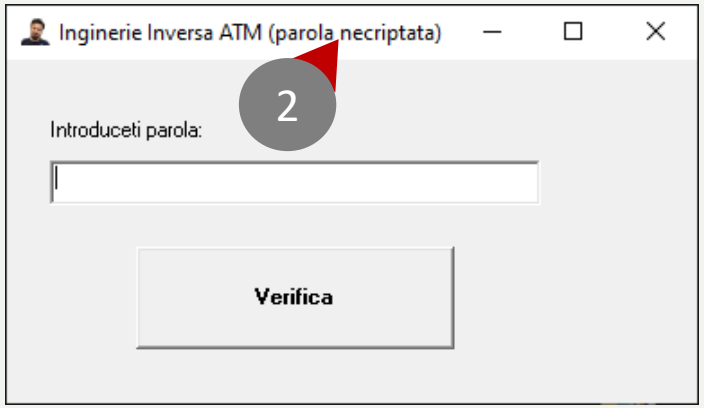
APLICAȚIE PROTEJATĂ PRIN PAROLĂ

CE VEDEM CÂND EXECUTĂM FIȘIERUL ȘI CUM FUNCȚIONEAZĂ?

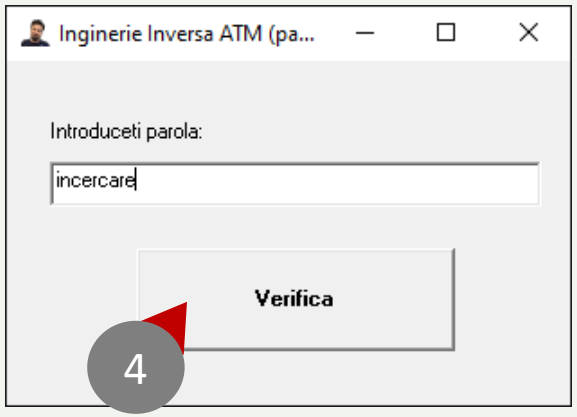
1

Visual Basic 6.0 este RAD, adică Rapid Application Development)

3

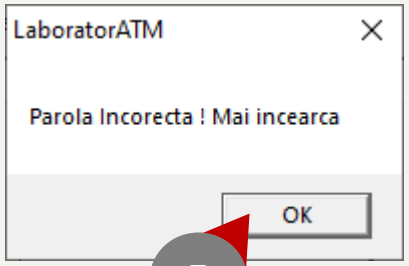


2



4

Ce se află în spatele aplicației compilate?



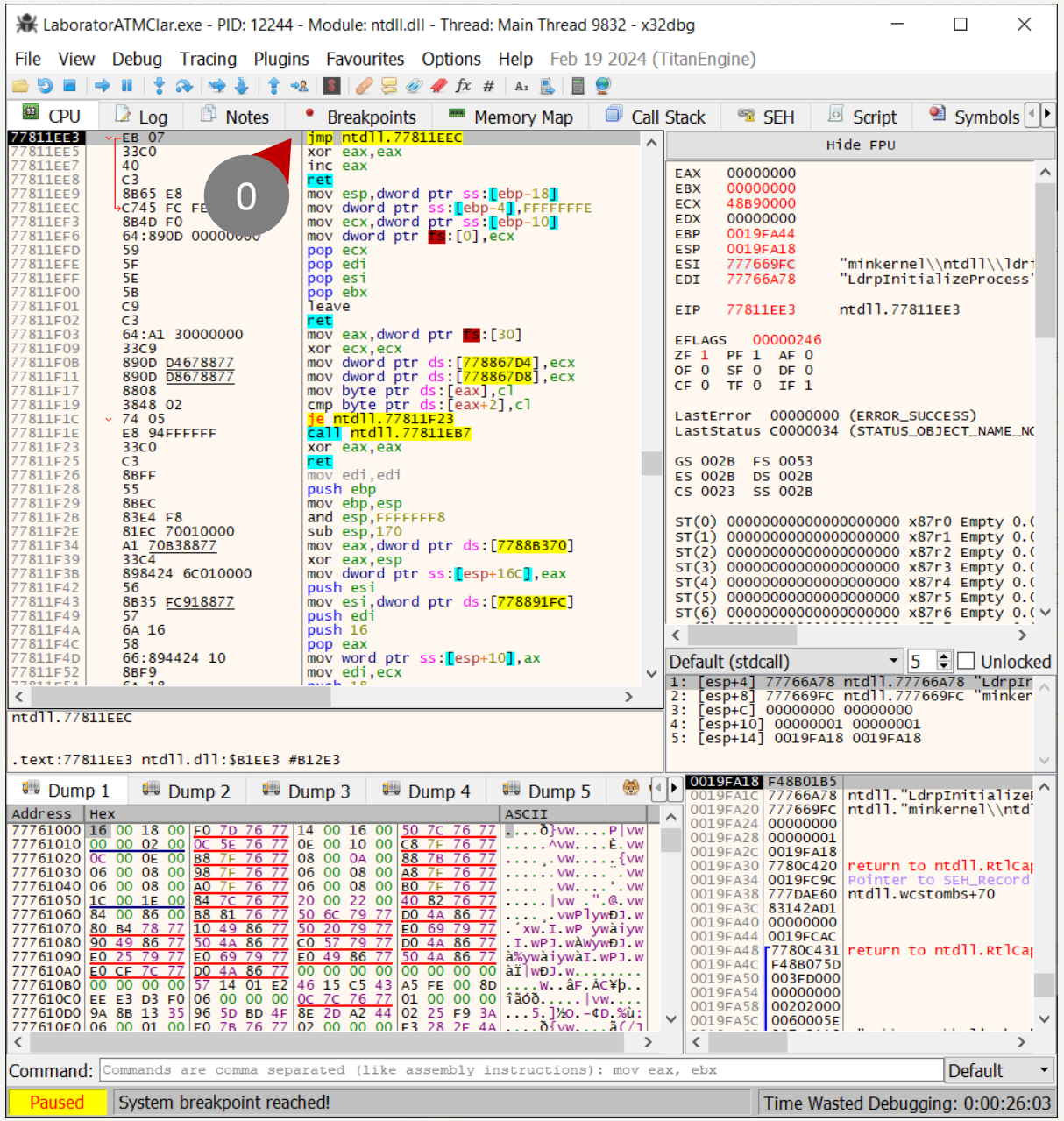
5

Sursa executabilului:

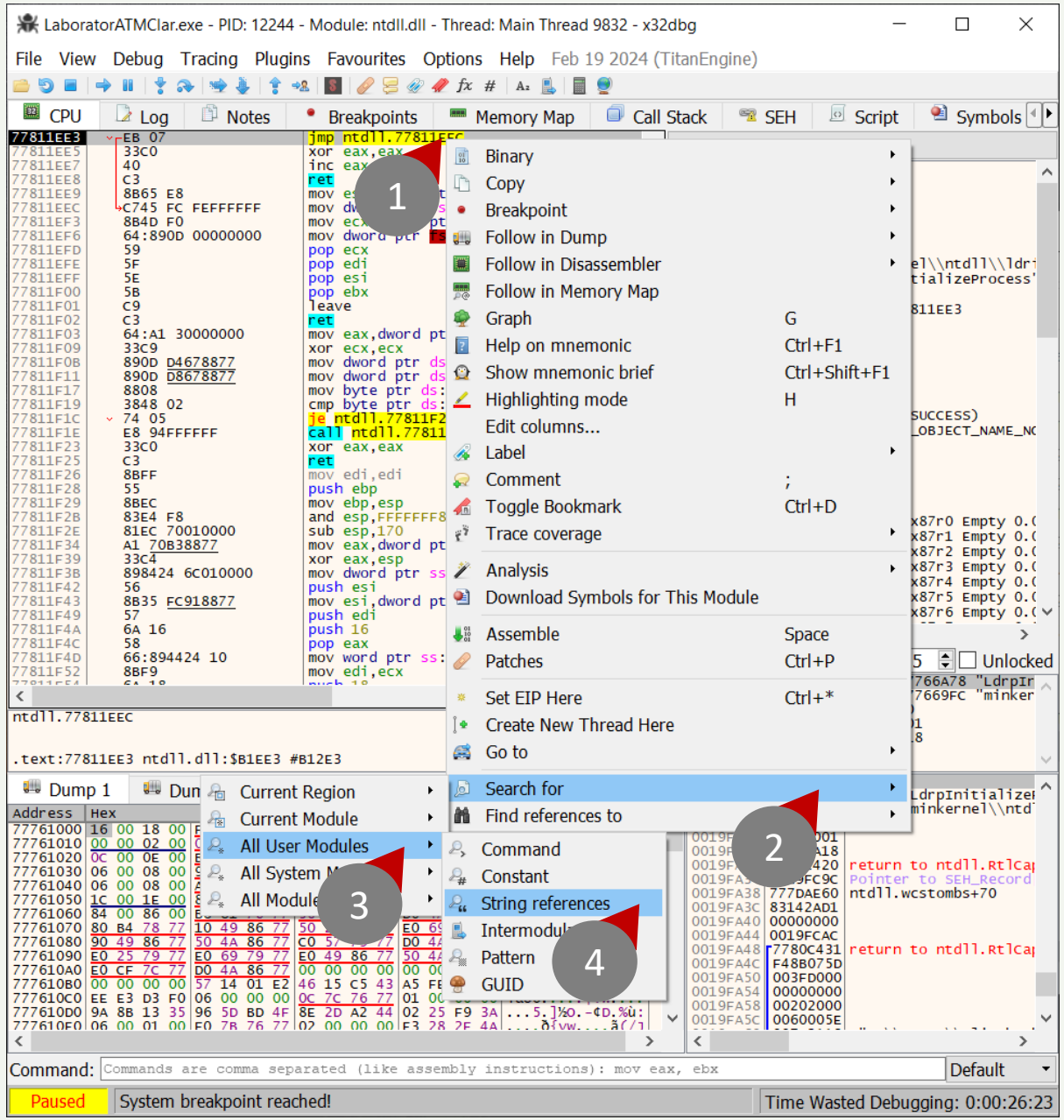
```
Private secret As String

Private Sub Buton_Click()

    If (InputParola.Text = secret) Then
        MsgBox "Parola corecta !"
    Else
        MsgBox "Parola Incorecta ! Mai incearca"
    End If

End Sub

Private Sub Form_Load()
    secret = "atm2024"
End Sub
```

6

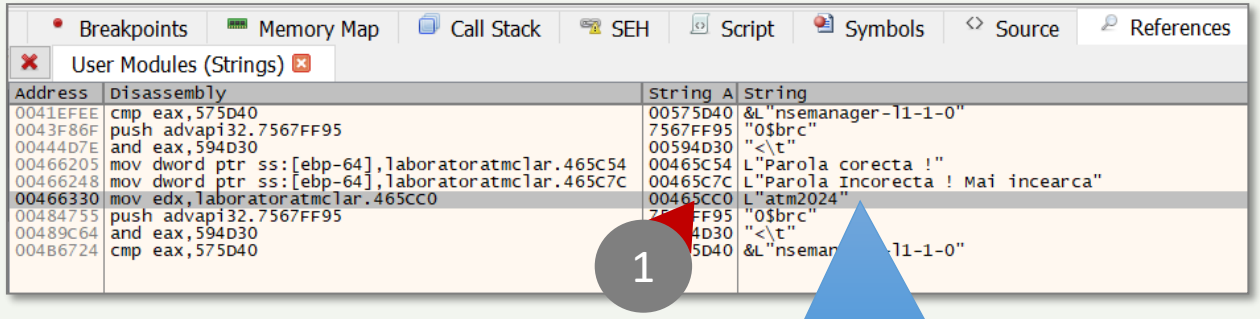
X32dbg ne setează punctul de intrare inițial (nu cel din **.text**):



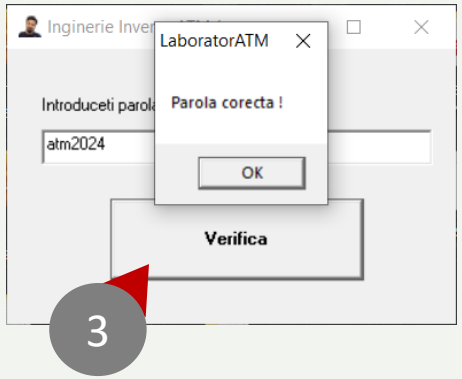
Căutăm șiruri în executabil:

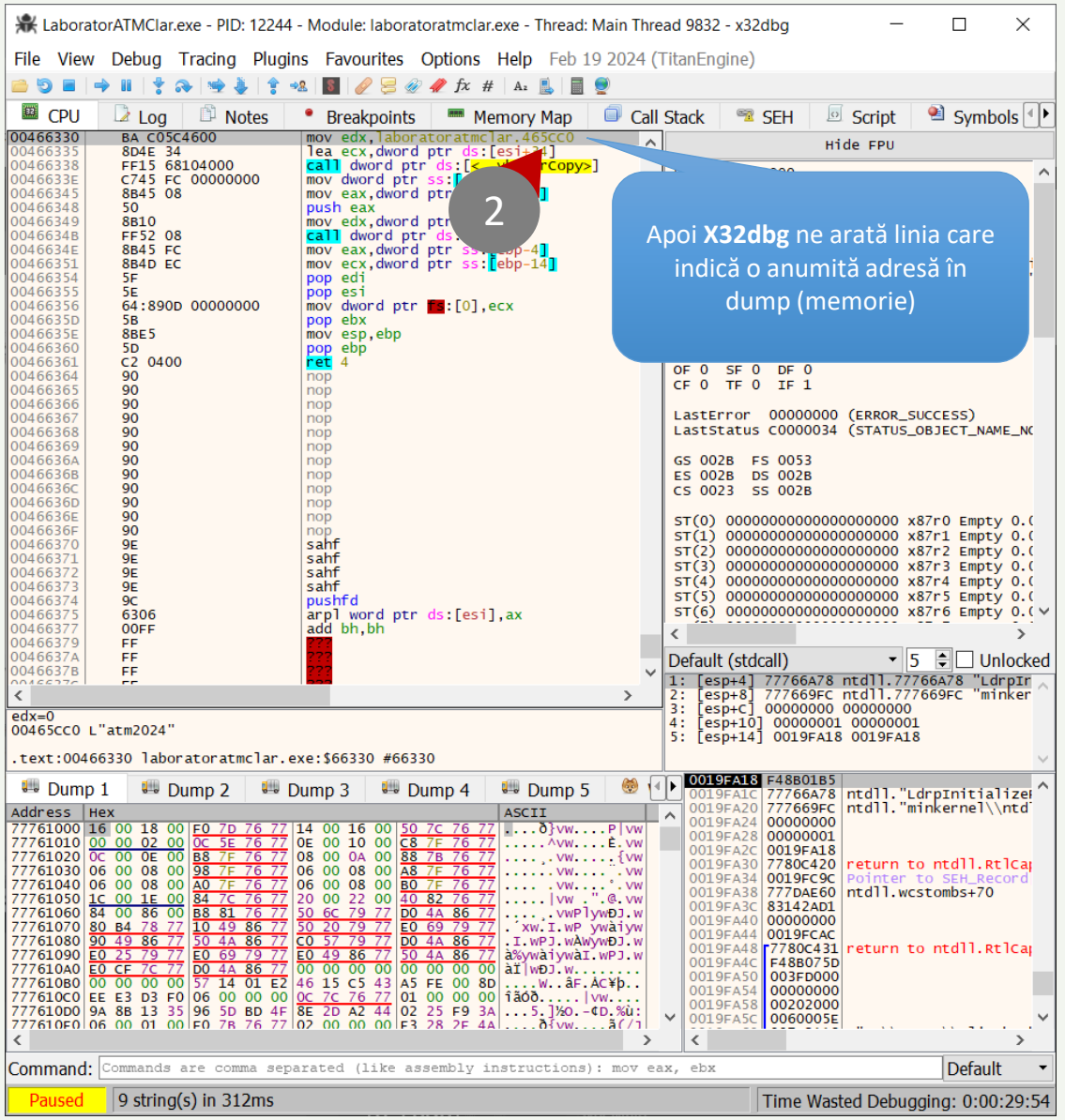


Prin forța brută putem testa toate șirurile, dar unele sunt evidente:



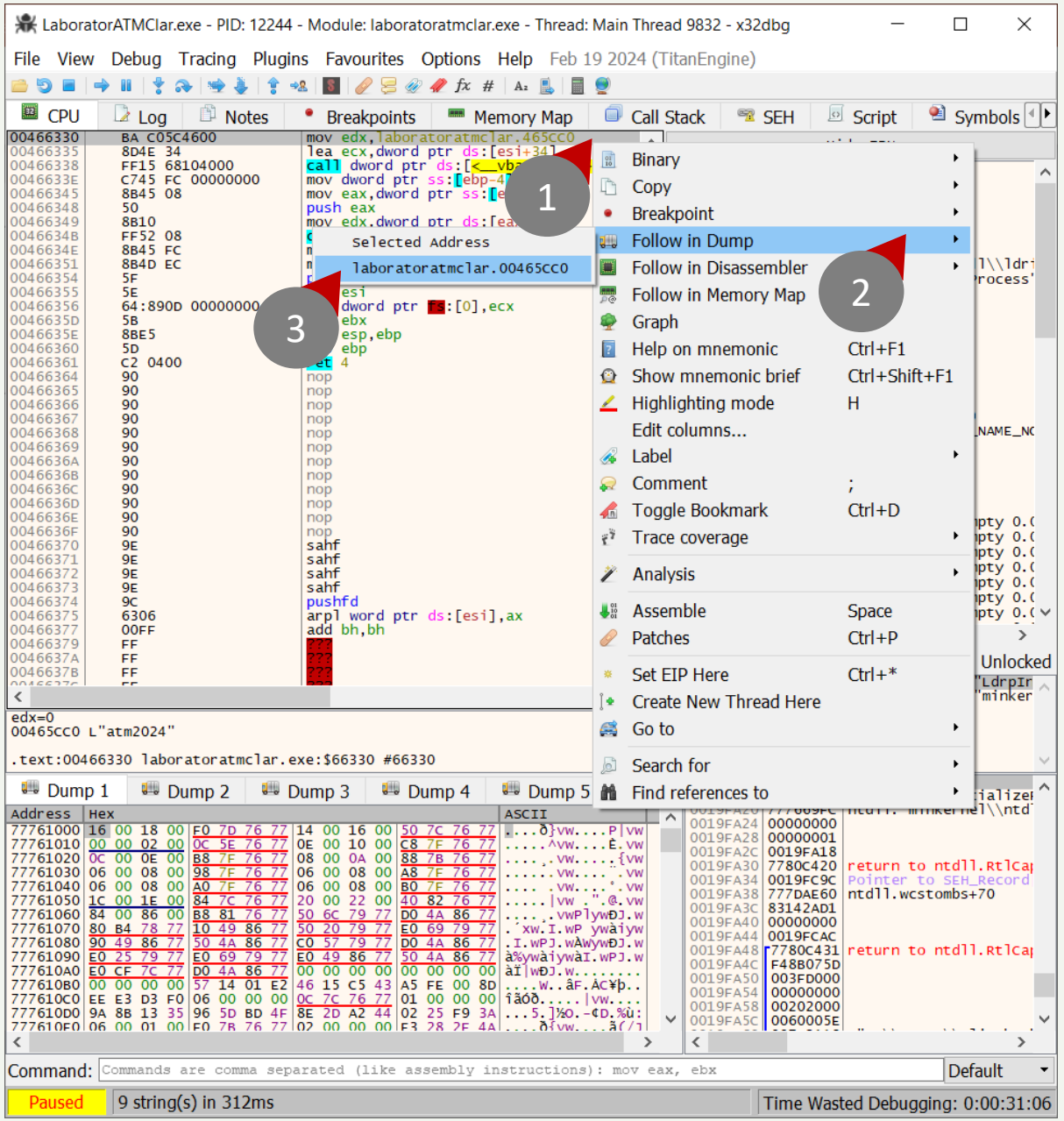
Facem dublu clic pe șirul de interes care ne-a permis accesul la program.



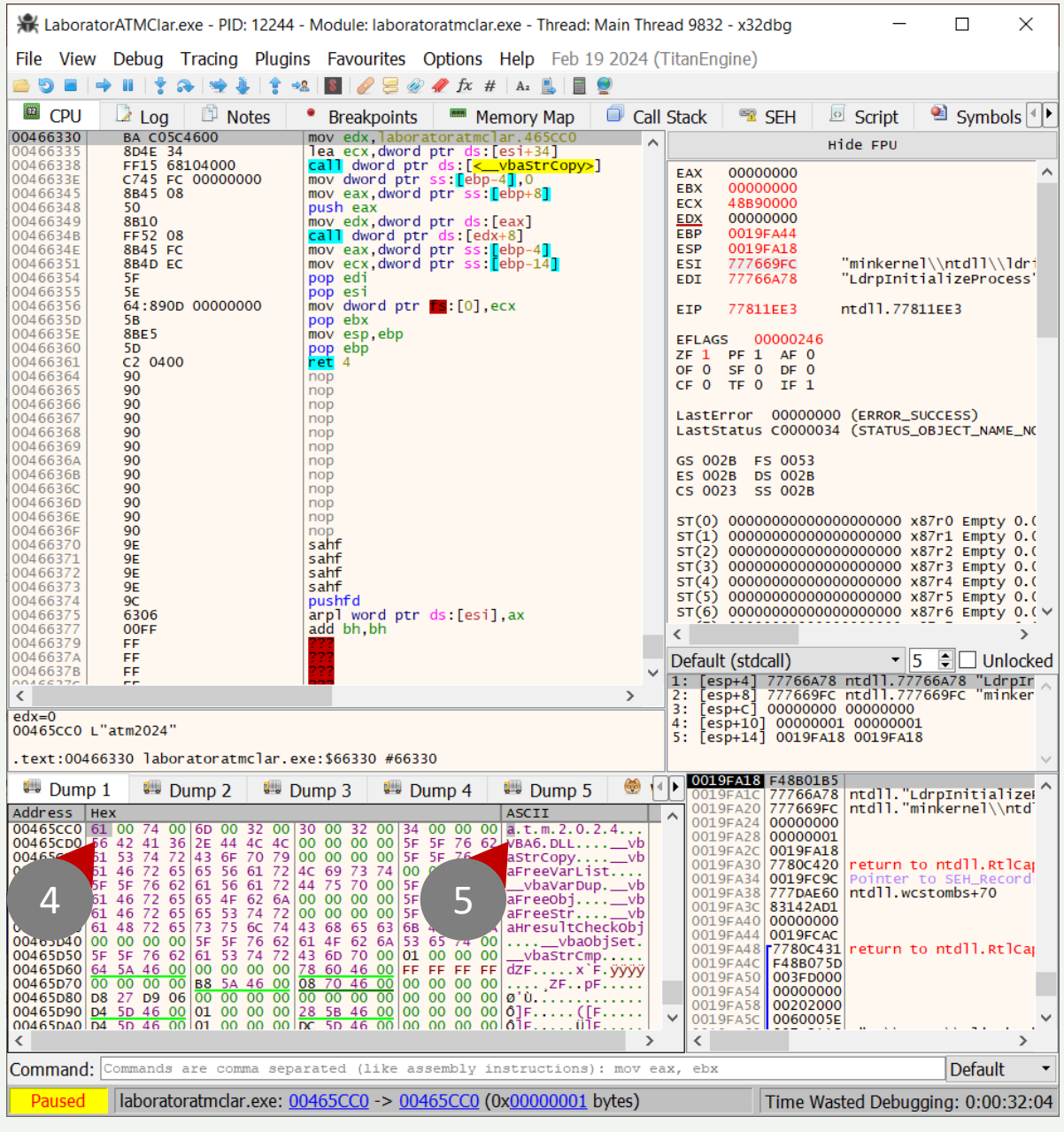


Putem verifica direct string-urile pe care le găsim. Parola este ușor de găsit atâta timp cât este în text clar. Întrebarea este: Cum putem schimba parola din fisierul executabil?

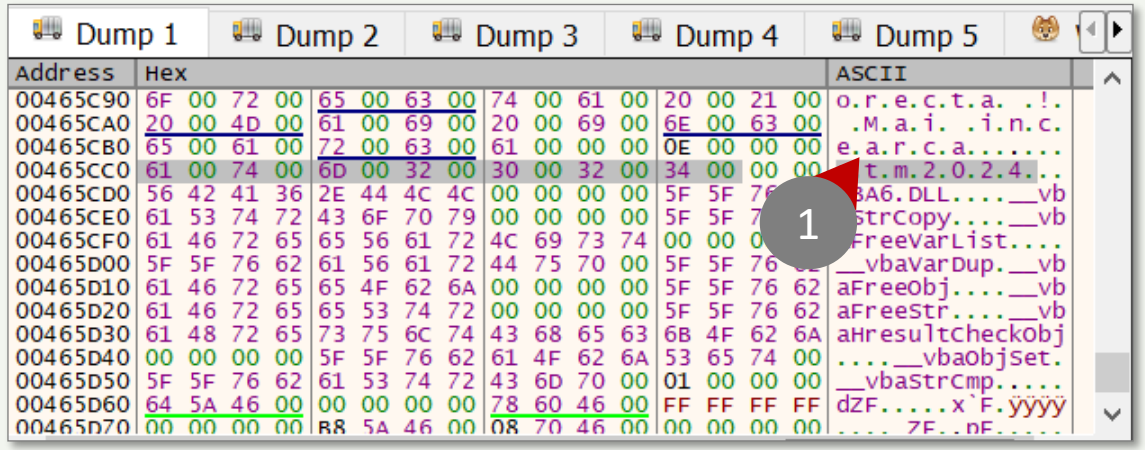
Accesăm adresa *00465CC0* din memoria *dump*:

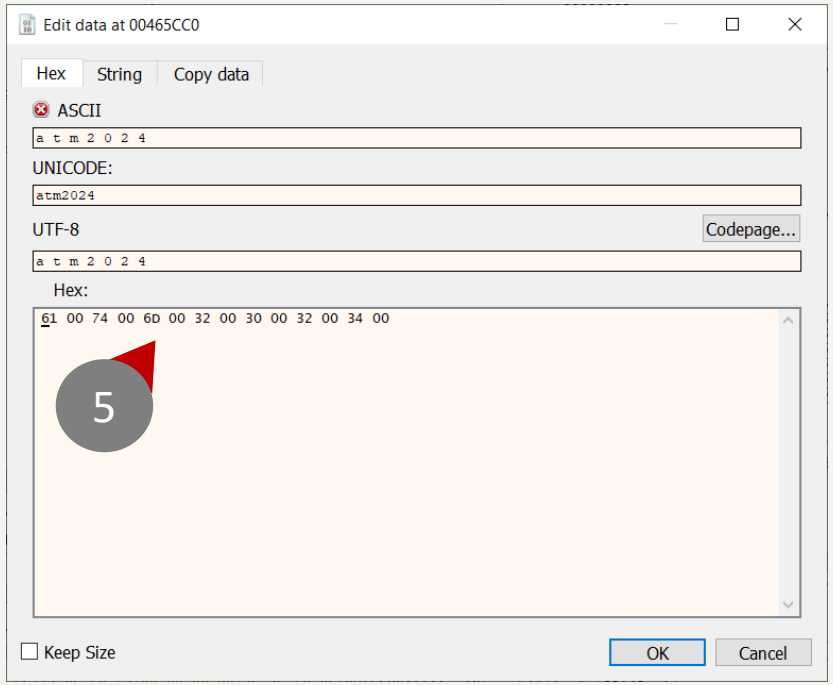


În memoria *dump* putem vedea clar parola:

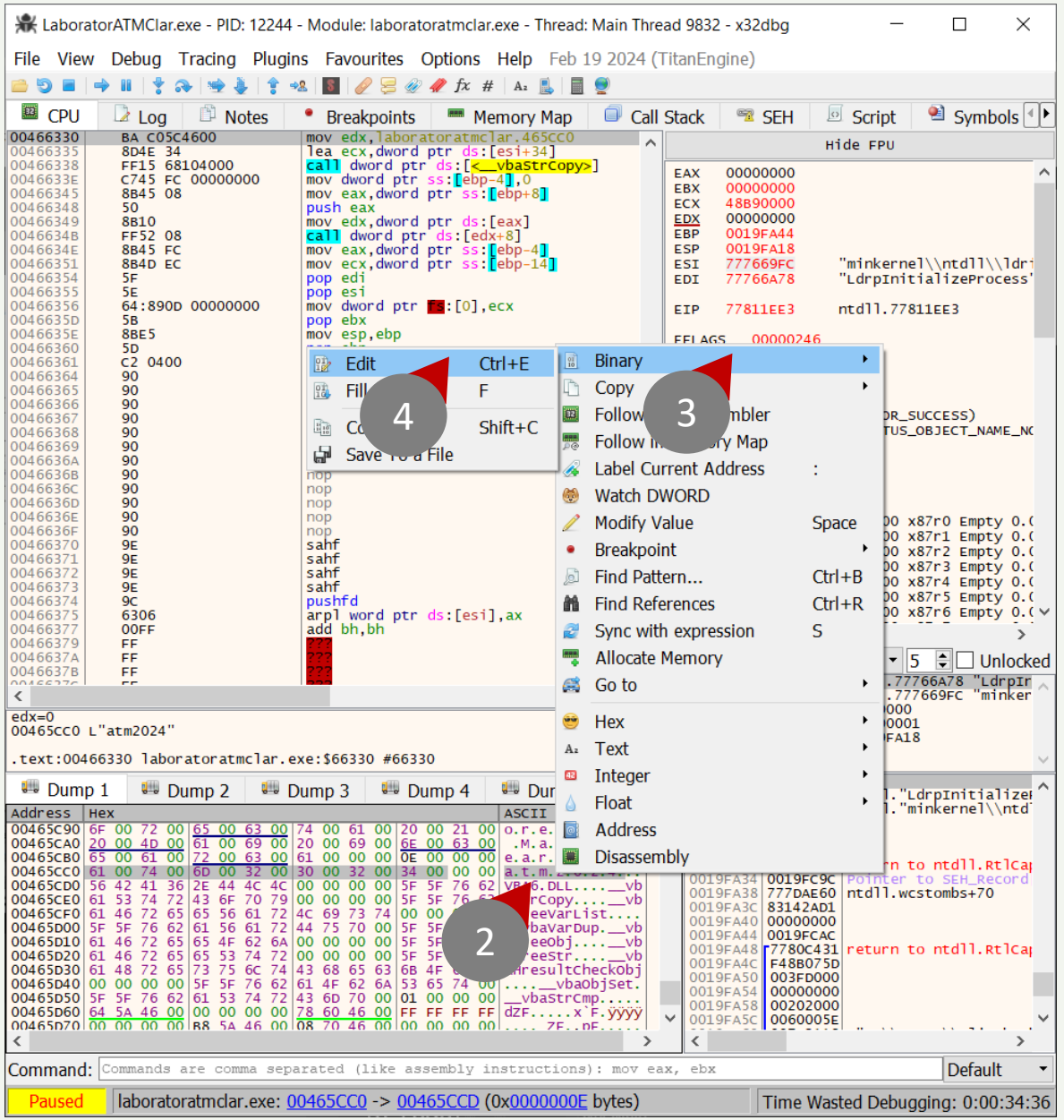


Selectăm parola în *dump* pentru editare:





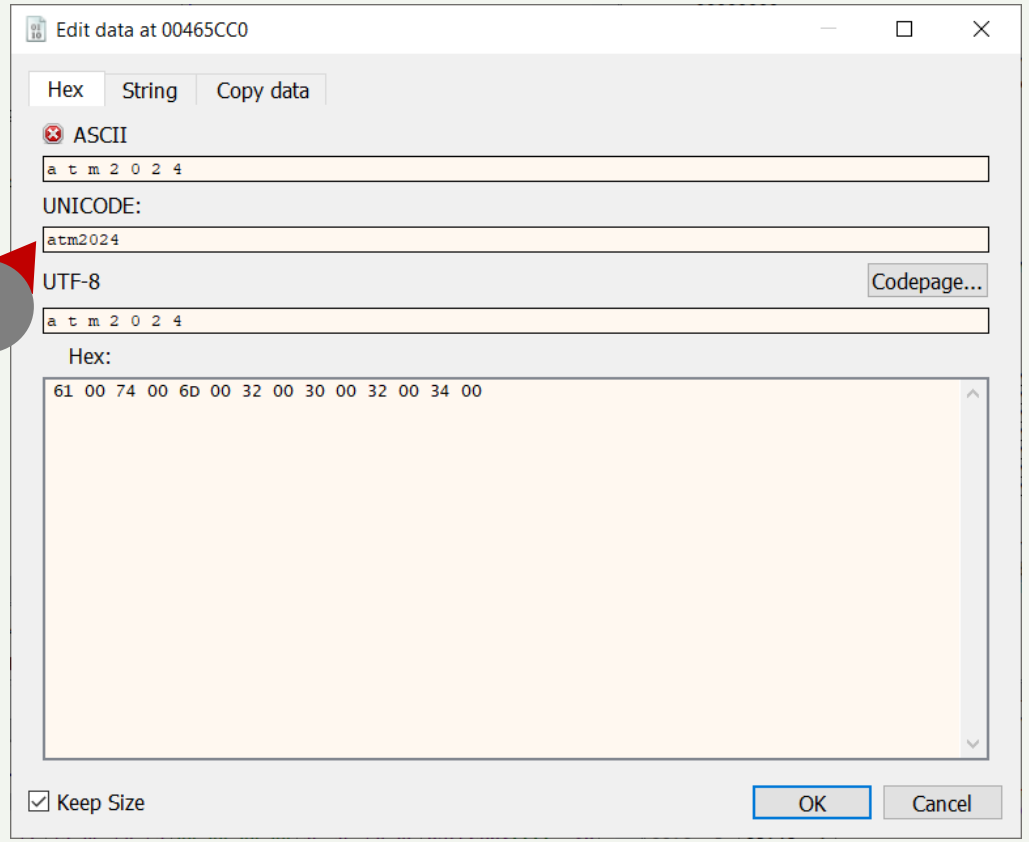
Editățm secțiunea selectată în dump în modul binar:

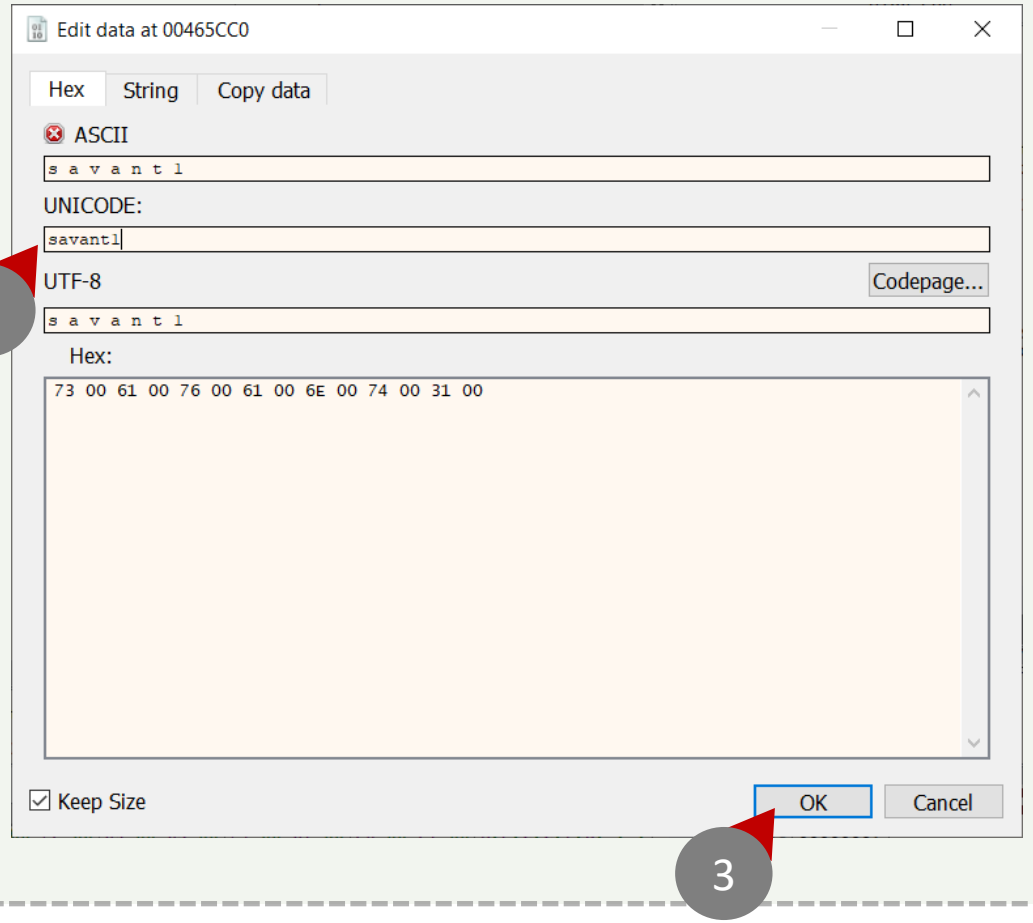


UTF-16 codifică fiecare caracter pe cel puțin 2 octeți (bytes), chiar și atunci când caracterul este ASCII pur (“a.t.m.2.0.2.4.”).

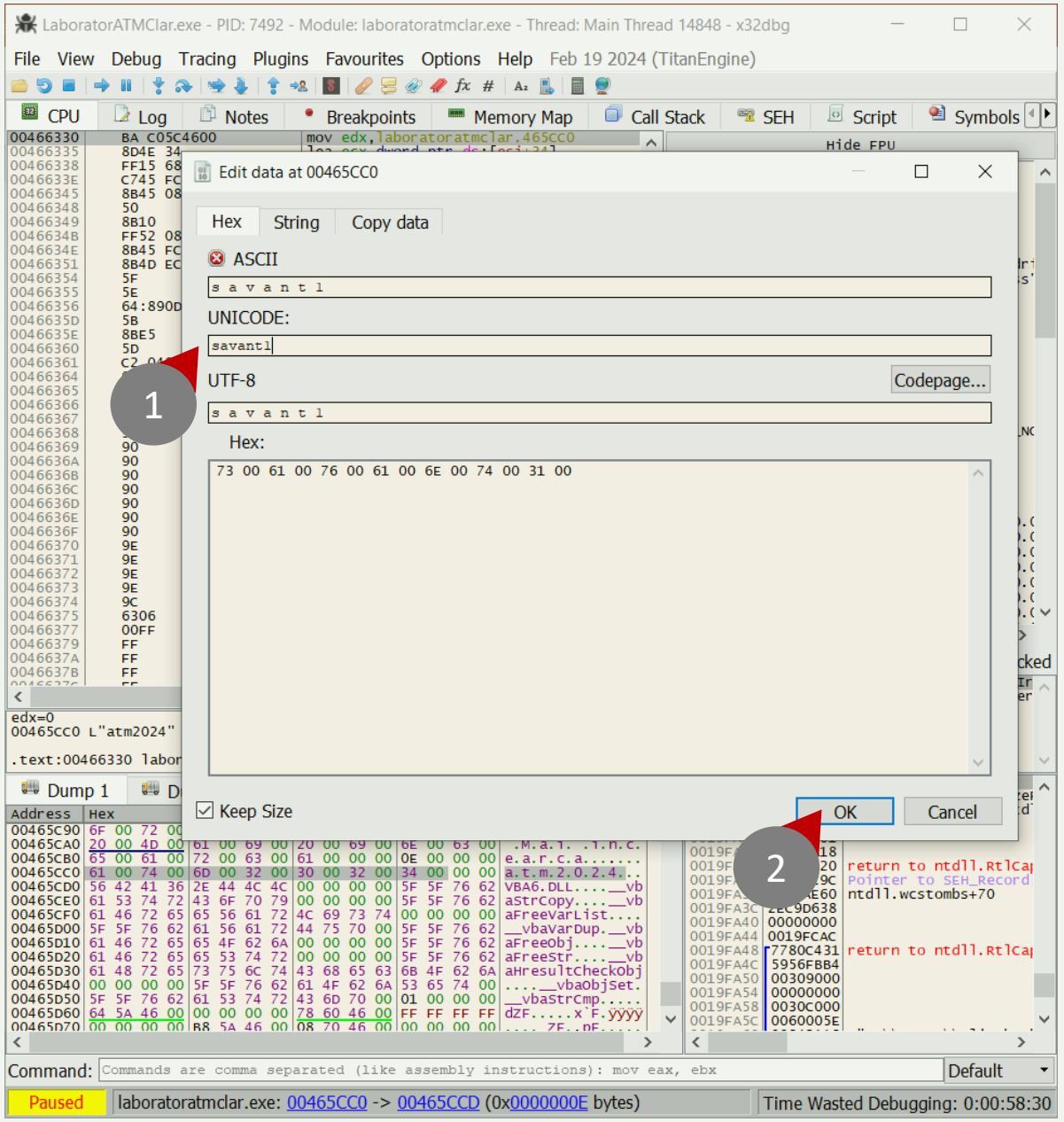
EDITARE PAROLĂ

MENȚINEREA DIMENSIUNII ESTE IMPORTANTĂ

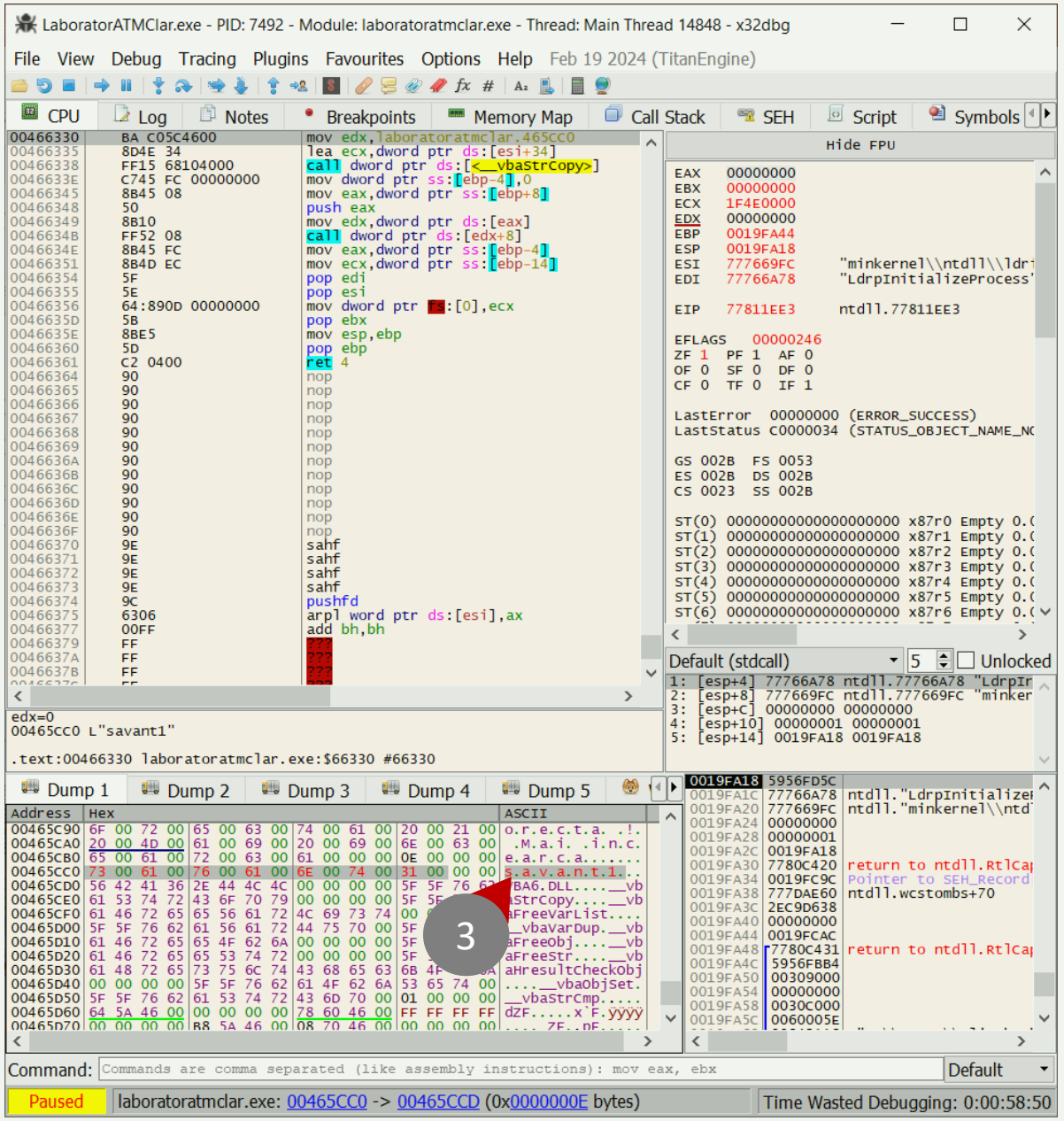




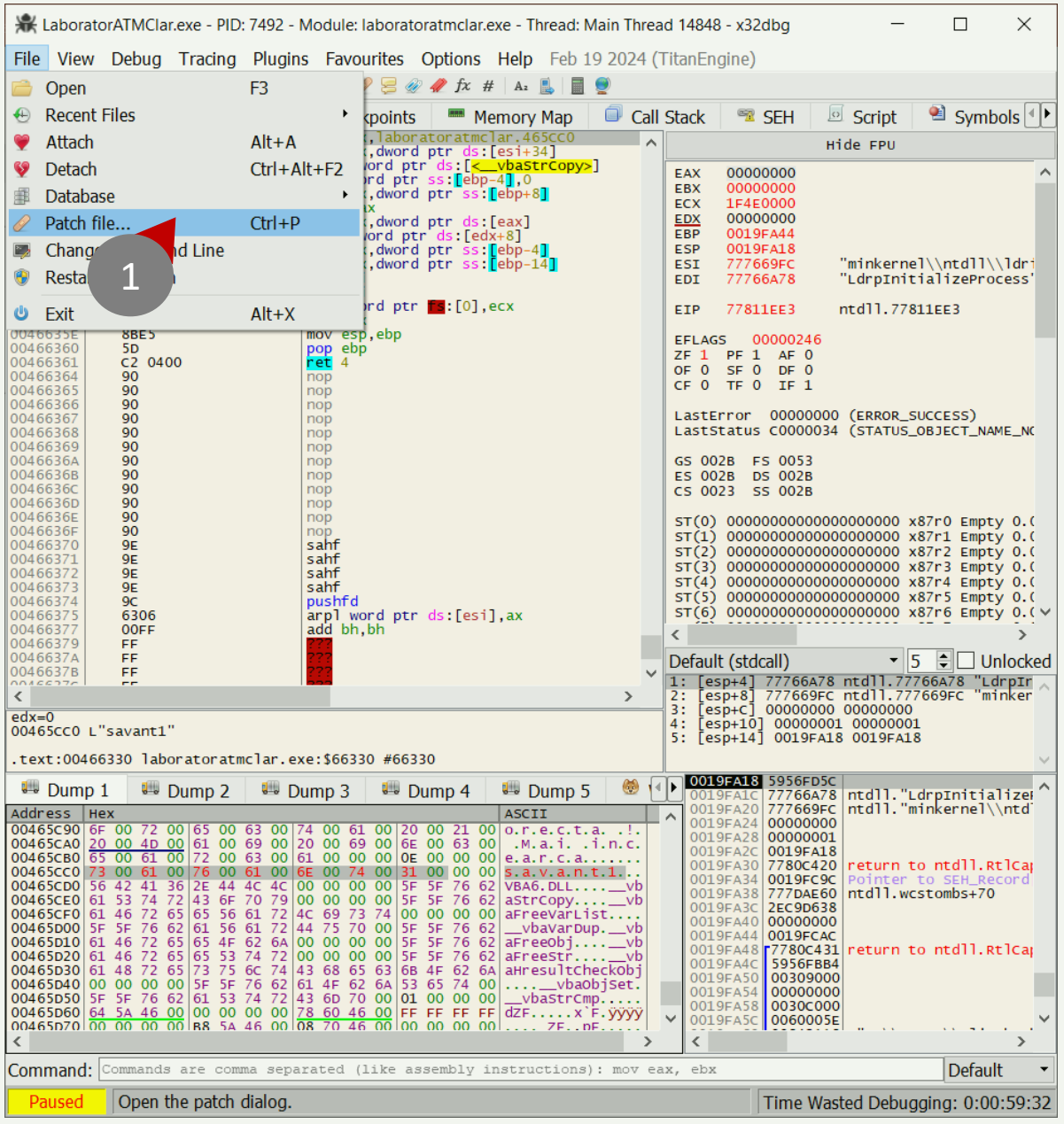
Schimbăm parola:

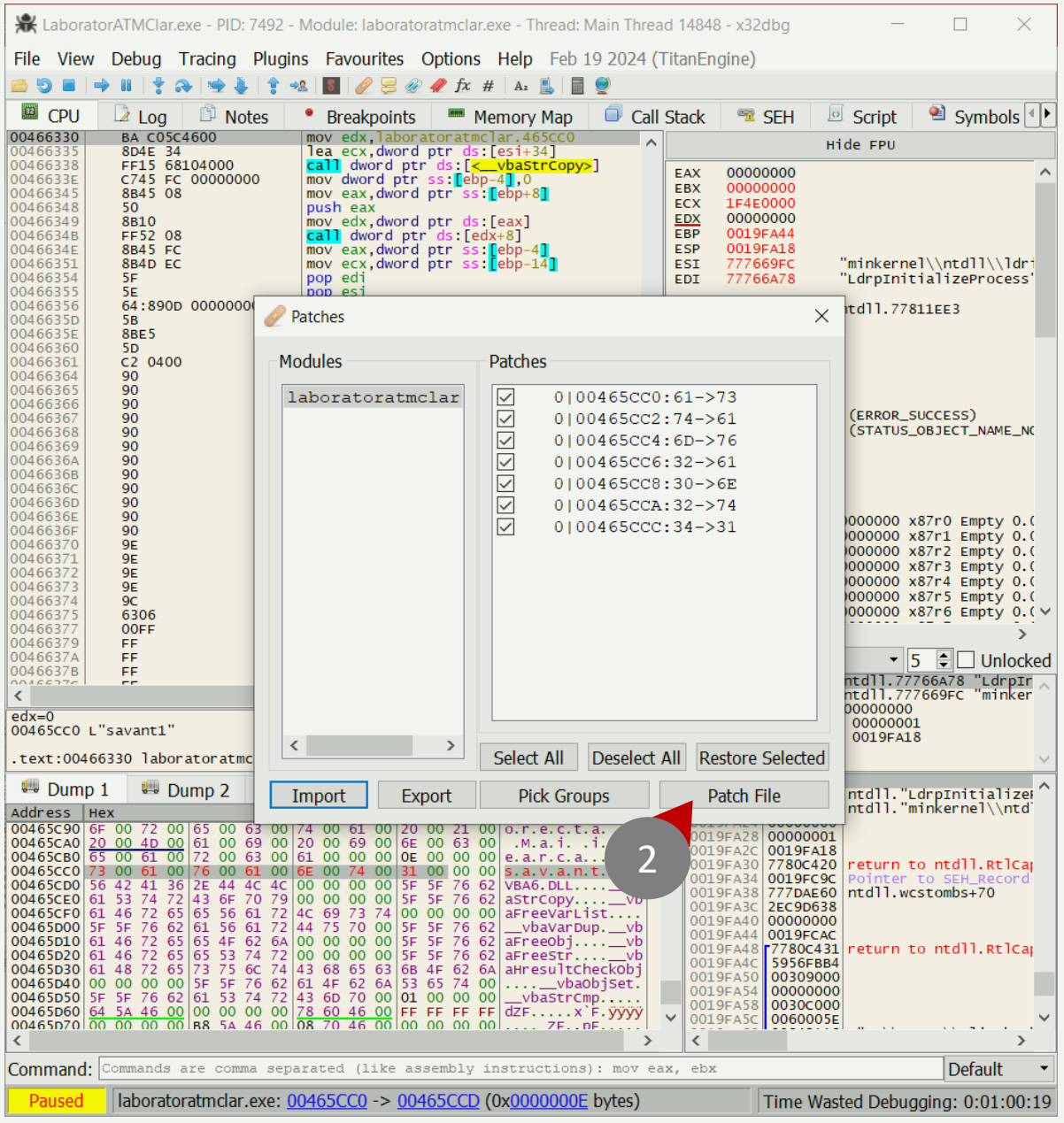


Observăm modificările salvate în *dump*:



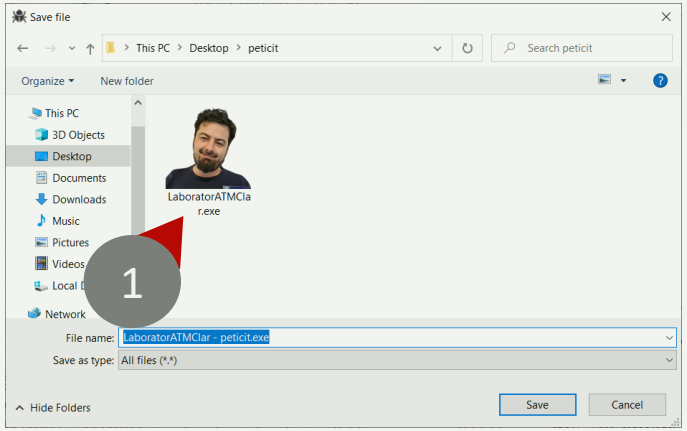
Scriem noua versiune a executabilului:

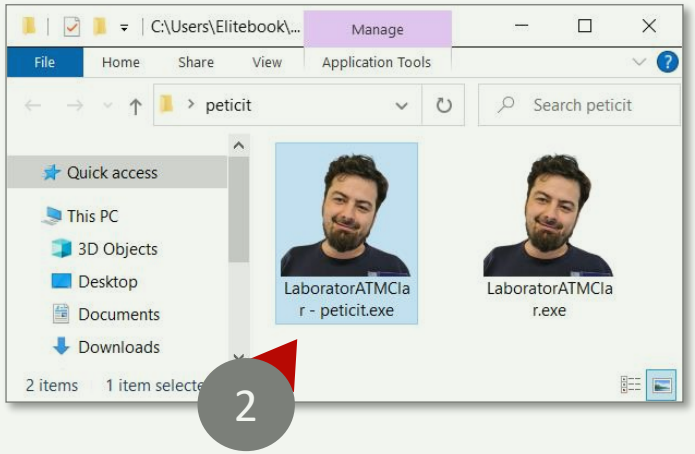




PAROLĂ SCHIMBATĂ

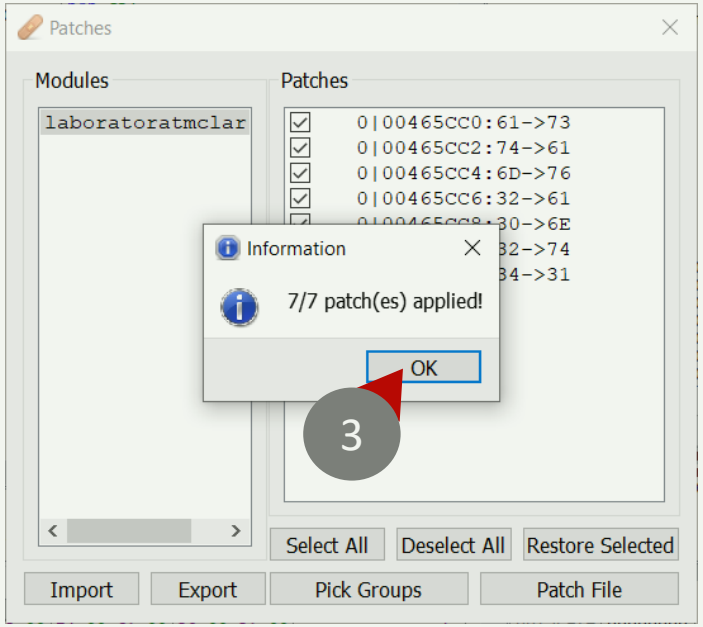
TESTĂM NOUL EXECUTABIL





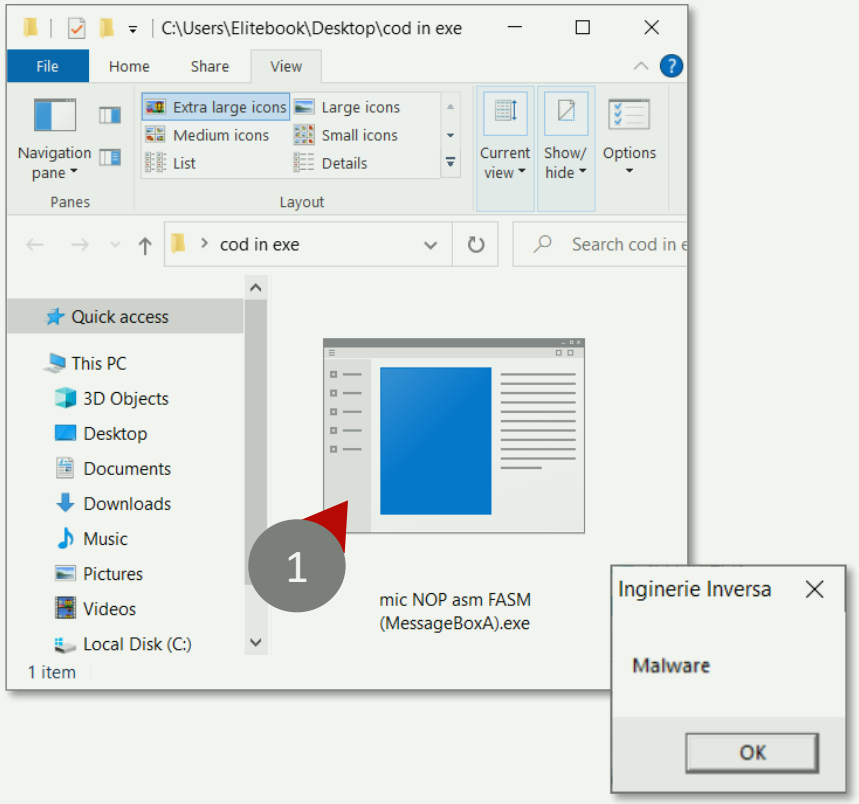
Noua parola functioneaza !

Cate modificari?



C.8.2 ADĂUGAREA DE COD MAȘINĂ

“PEȘTERA ÎN COD” (CODE CAVE)
SAU CUM SĂ ADĂUGAȚI COD MAȘINĂ IN EXECUTABILE



Cum să adăugați cod mașină în fișiere executabile PE?

Pentru a adăuga cod mașină:

- Trebuie să găsim o zonă din executabil care nu este folosită!
- Trebuie să eliminăm o parte din codul original fără a afecta „prea mult” funcționalitatea!

Compilarea unui executabilul în FASM (din nou)

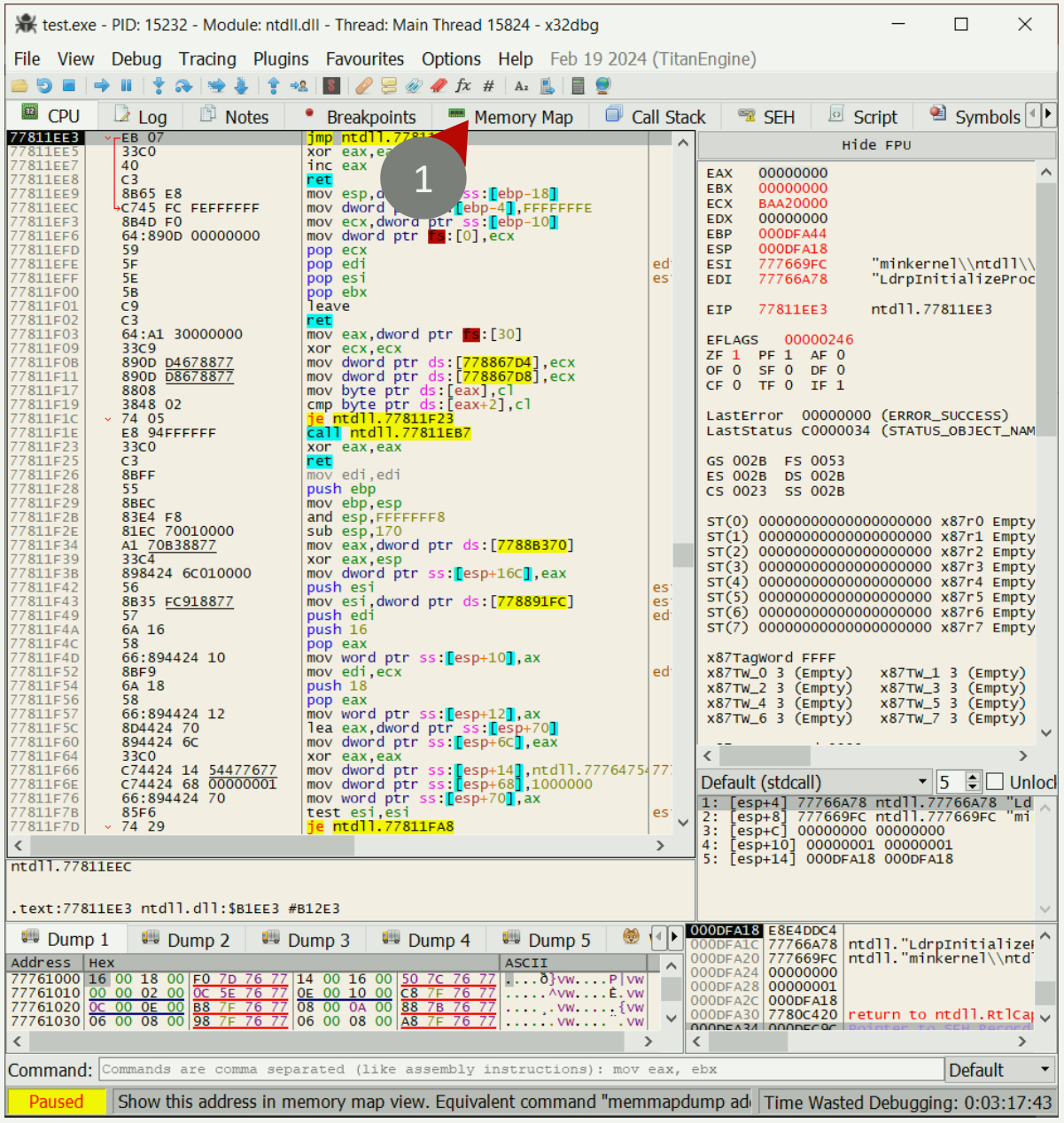
.data

.text

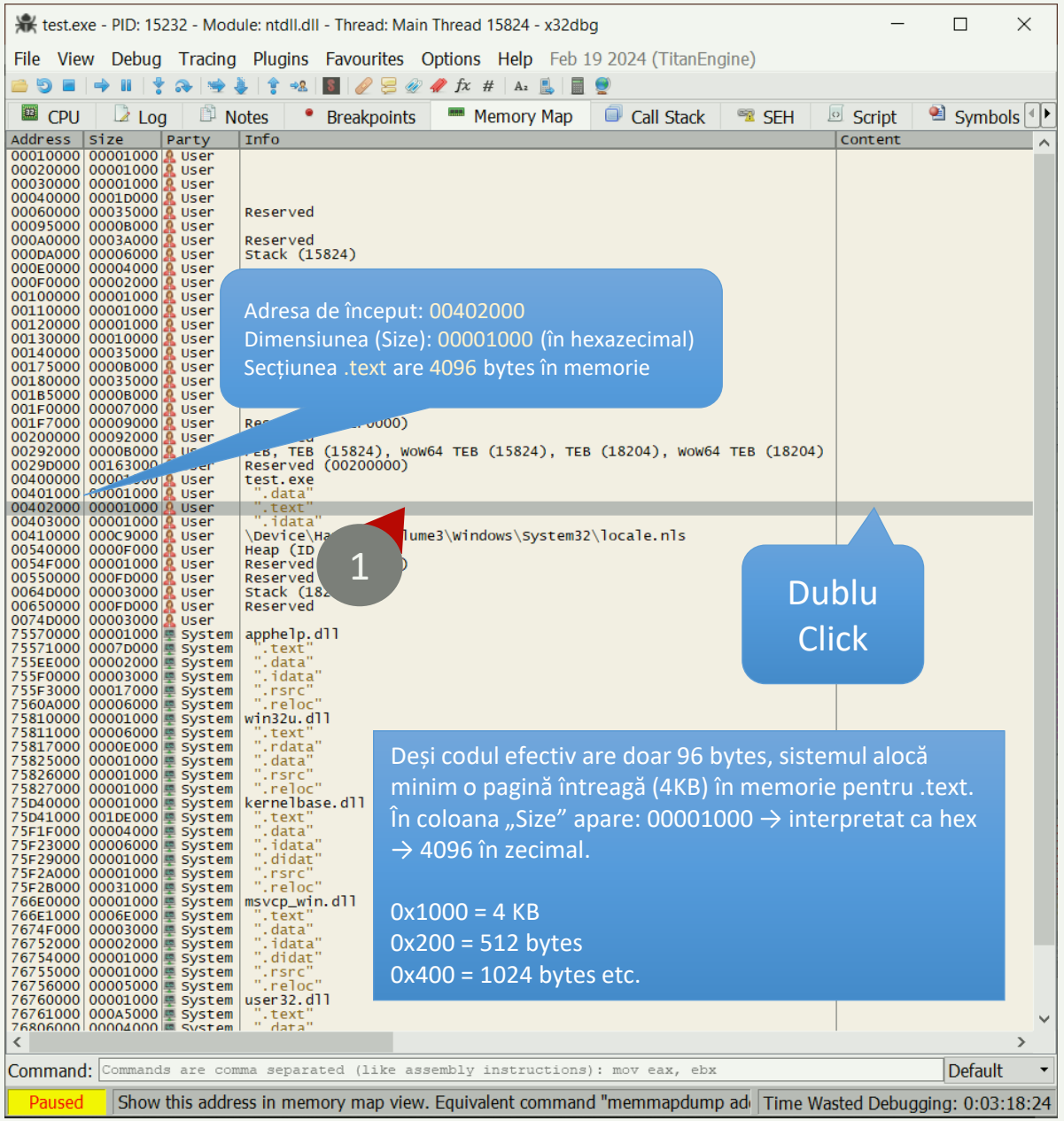
.idata

```
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

include 'INCLUDE/win32a.inc' ; include fișierul de antet 'win32a.inc' care conține macro-uri și definiții pentru interfața cu API-ul Windows

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)

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
    NOP                 ; introducem instructiuni NOP (No Operation), atatea cate dorim pentru exemplificare
    NOP
    NOP
    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

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
```

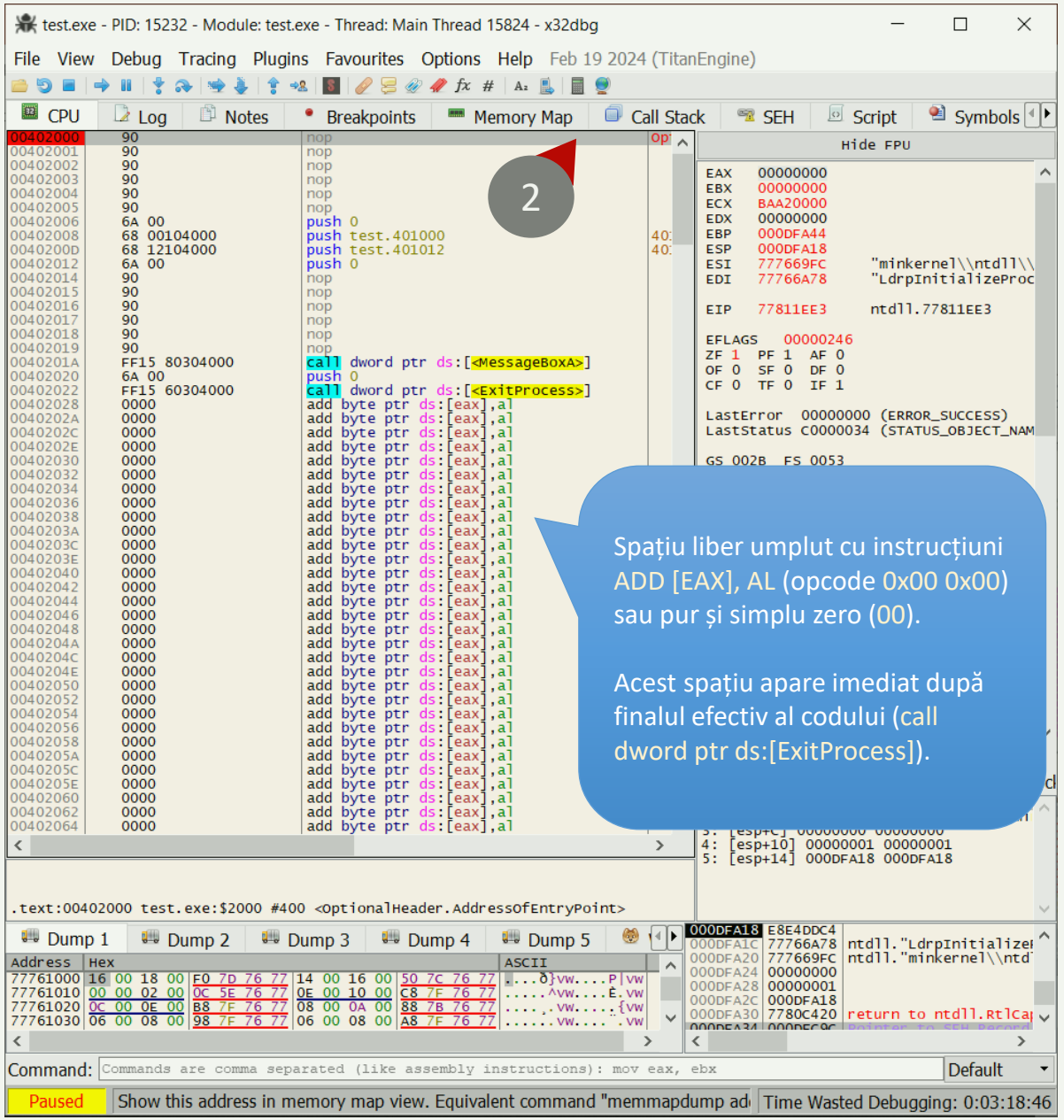
Dezasamblam executabilul:



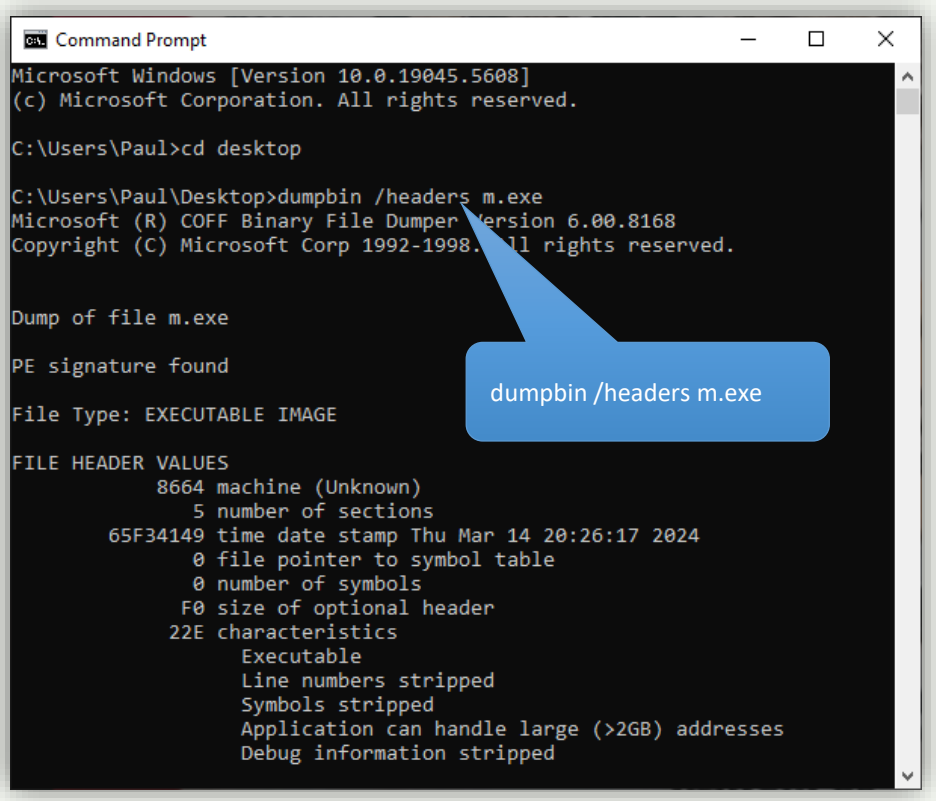
Selectați „Memory map” și căutați secțiunea **.text** a executabilului:



Suntem îndrumați la adresa punctului de intrare (real):



0x1000 (hexazecimal) = 4096 (zecimal)



Mai sus se prezintă utilizarea comenzii dumpbin /headers aplicată asupra unui fișier executabil (m.exe), prin care sunt afișate informații detaliate despre structura internă a fișierului PE (Portable Executable). Rezultatul include antetul fișierului (File Header), antetul opțional (Optional Header) și anteturile secțiunilor (.text, .rdata, .pdata, .xdata, .idata). Aceste informații arată tipul fișierului, arhitectura țintă, dimensiuni ale codului și datelor, punctul de intrare, precum și caracteristicile fiecărei secțiuni, fiind utile pentru analiza statică a executabilelor și pentru înțelegerea modului în care sunt organizate datele și instrucțiunile în memorie.

Microsoft Windows [Version 10.0.19045.5608]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Paul>cd desktop

C:\Users\Paul\Desktop>dumpbin /headers m.exe
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file m.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
8664 machine (Unknown)
5 number of sections
65F34149 time date stamp Thu Mar 14 20:26:17 2024
0 file pointer to symbol table
0 number of symbols
F0 size of optional header
22E characteristics
Executable
Line numbers stripped
Symbols stripped
Application can handle large (>2GB) addresses
Debug information stripped

OPTIONAL HEADER VALUES
20B magic #
2.36 linker version
200 size of code
800 size of initialized data
0 size of uninitialized data
1000 RVA of entry point
1000 base of code

SECTION HEADER #1
.text name
60 virtual size
1000 virtual address
200 size of raw data
400 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
60500020 flags
Code
RESERVED - UNKNOWN
RESERVED - UNKNOWN
Execute Read

SECTION HEADER #2
.rdata name
50 virtual size
2000 virtual address
200 size of raw data
600 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
40500040 flags
Initialized Data
RESERVED - UNKNOWN
RESERVED - UNKNOWN
Read Only

SECTION HEADER #3
.pdata name
C virtual size
3000 virtual address
200 size of raw data
800 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
40300040 flags
Initialized Data
RESERVED - UNKNOWN
RESERVED - UNKNOWN
Read Only

SECTION HEADER #4
.xdata name
8 virtual size
4000 virtual address
200 size of raw data
A00 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
40300040 flags
Initialized Data
RESERVED - UNKNOWN
RESERVED - UNKNOWN
Read Only

SECTION HEADER #5
.idata name
68 virtual size
5000 virtual address
200 size of raw data
C00 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
C0300040 flags
Initialized Data
RESERVED - UNKNOWN
RESERVED - UNKNOWN
Read Write

Summary

68 .idata
C .pdata
50 .rdata
60 .text
8 .xdata

De ce există spațiu liber după cod în .text?

- Formatul PE cere ca fiecare secțiune (.text, .data, .rdata etc.) să fie aliniată la o dimensiune multiplă de *SectionAlignment* (RAM - implicit 4096 bytes) și *FileAlignment* (DISC - implicit 512 bytes) din headerul PE.

- De exemplu, chiar dacă .text conține doar 0x1A bytes de cod, secțiunea poate fi alocată pe 0x200 sau 0x1000 bytes în fișierul PE sau în memorie.

Dacă .text conține 4097 bytes de cod, ce dimensiune va avea secțiunea .text în fișierul PE?

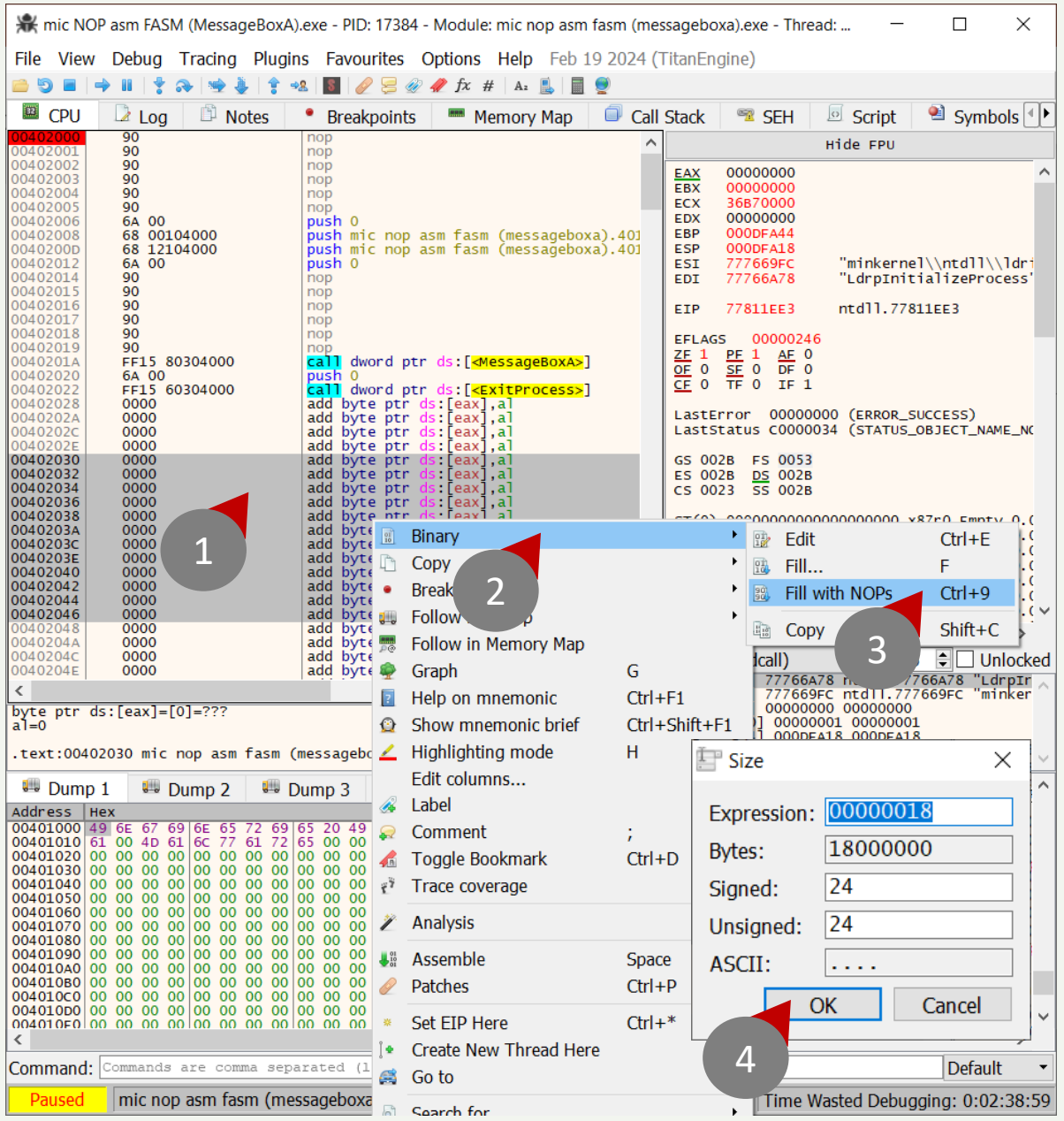
- **În fișierul PE (on disk)**, dimensiunea secțiunii este rotunjită la multiplu de **FileAlignment** (implicit 512 bytes). Deci 4097 este rotunjit la 4608 bytes (9 × 512).

- **În RAM (on load)**, dimensiunea secțiunii este rotunjită la multiplu de **SectionAlignment** (implicit 4096 bytes = 1 pagină de memorie). Deci 4097 este rotunjit la 8192 bytes (2 × 4096).

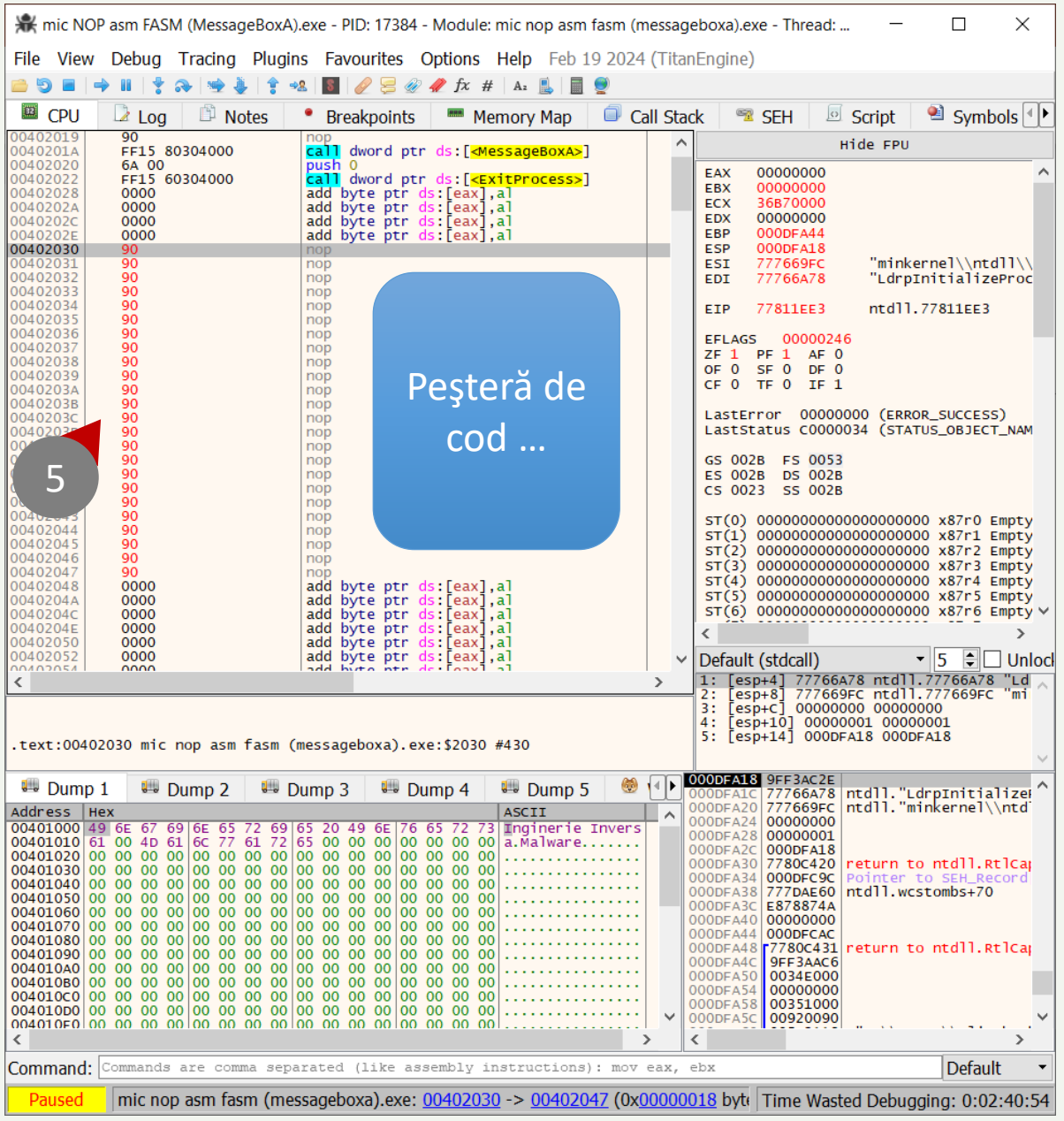
Prin urmare:

- În fișier: **4608 bytes**
- În RAM: **8192 bytes**

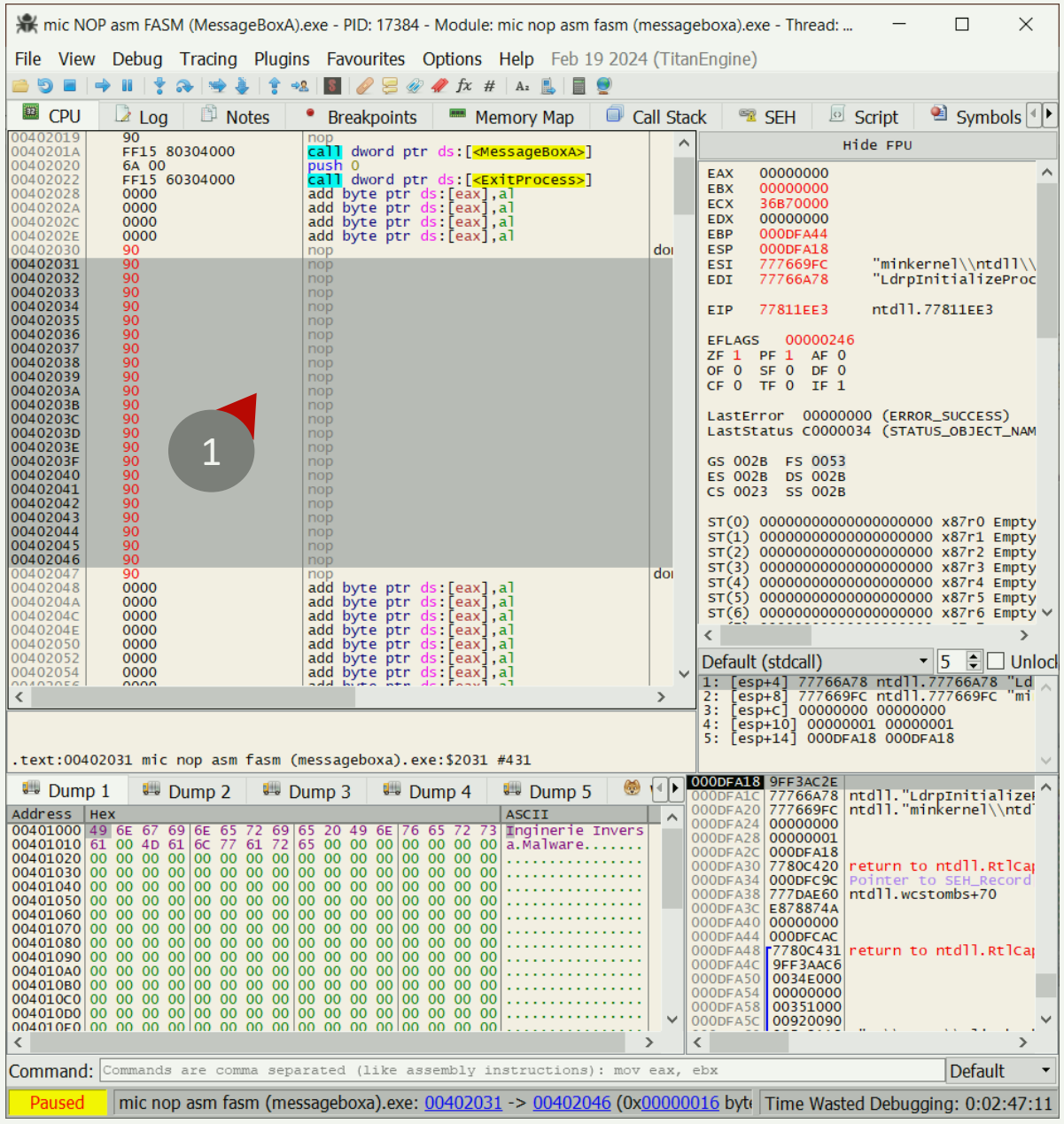
Facem un spațiu umplut cu instrucțiuni **nop**



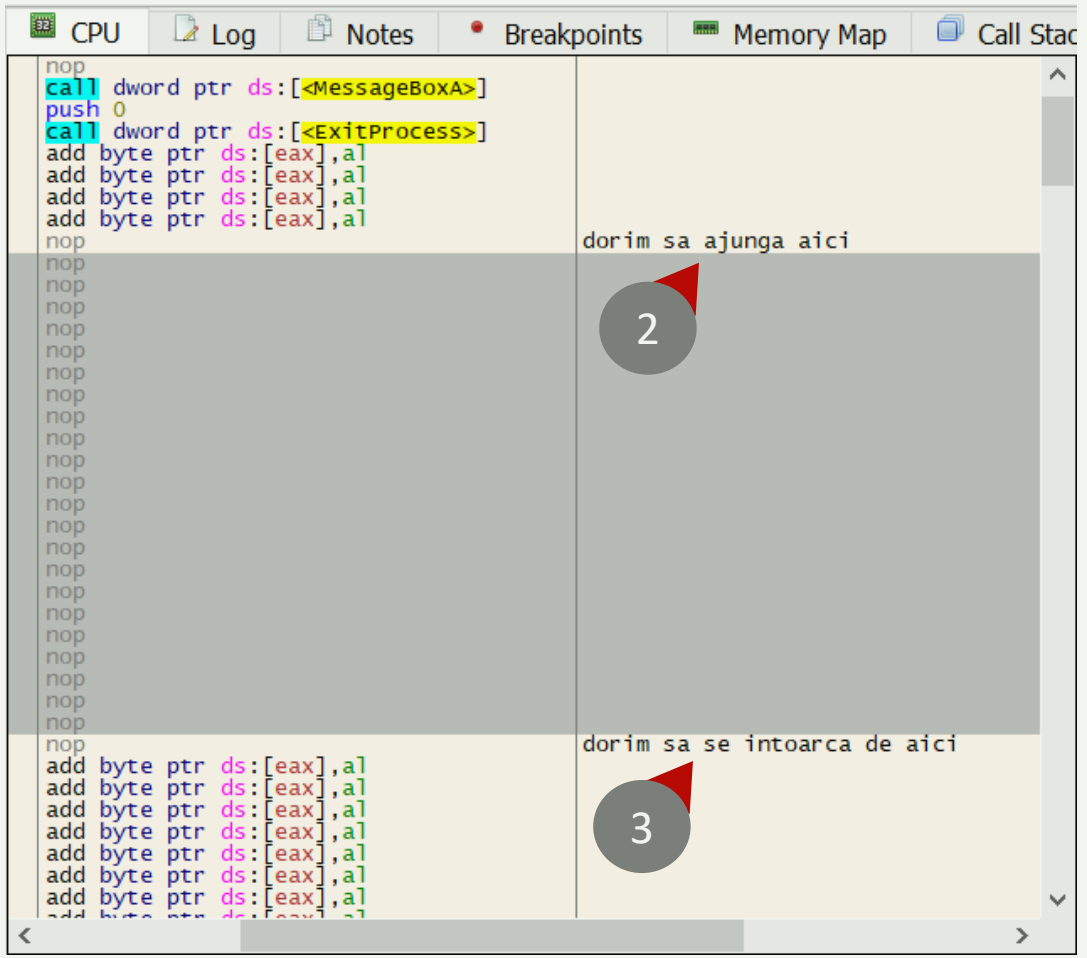
Acum am definit o regiune de interes în zona liberă a executabilului:



Selectați regiunea pentru o vizualizare mai bună:

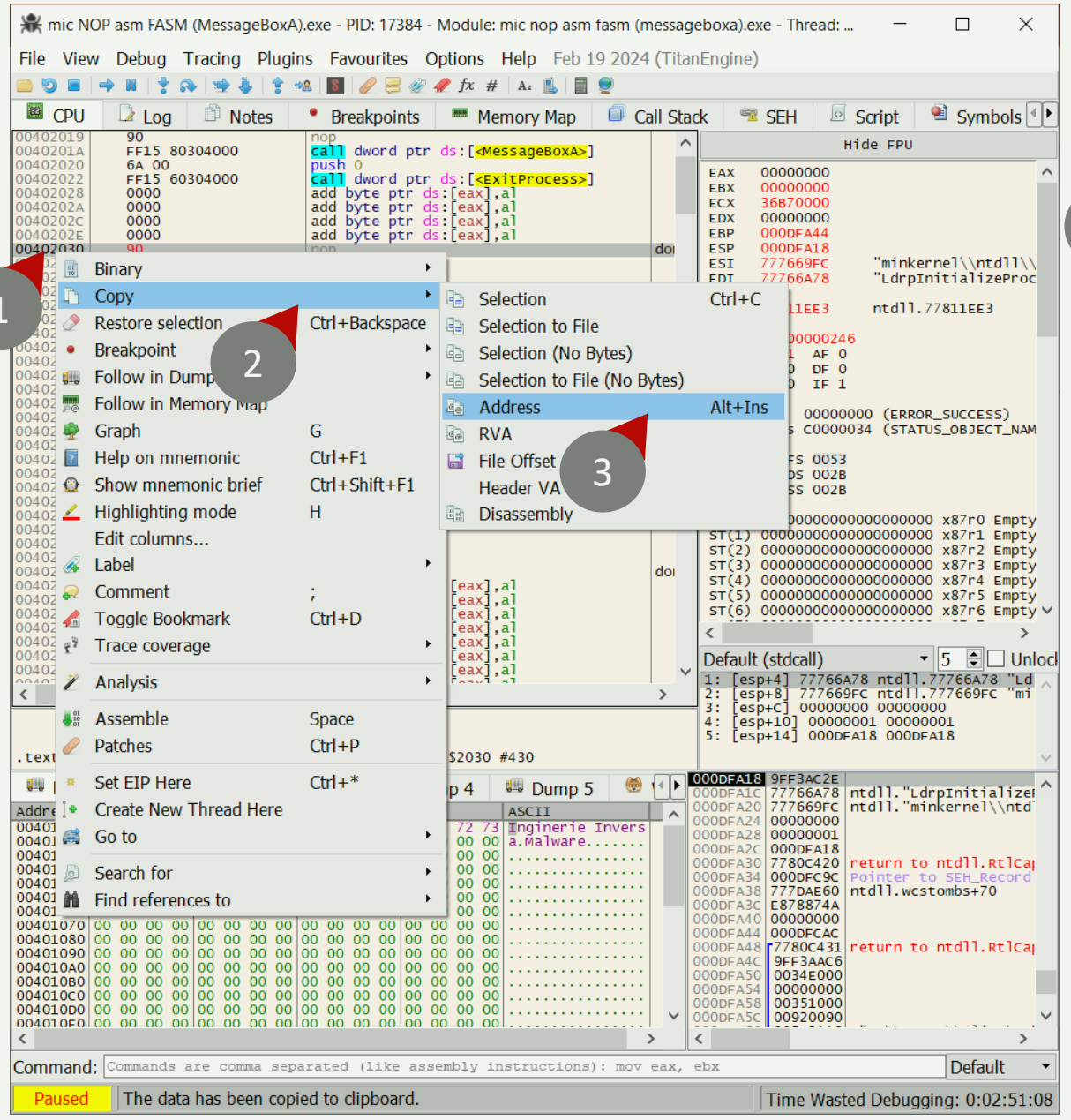


Comentăm punctele de intrare și ieșire din zona delimitată:

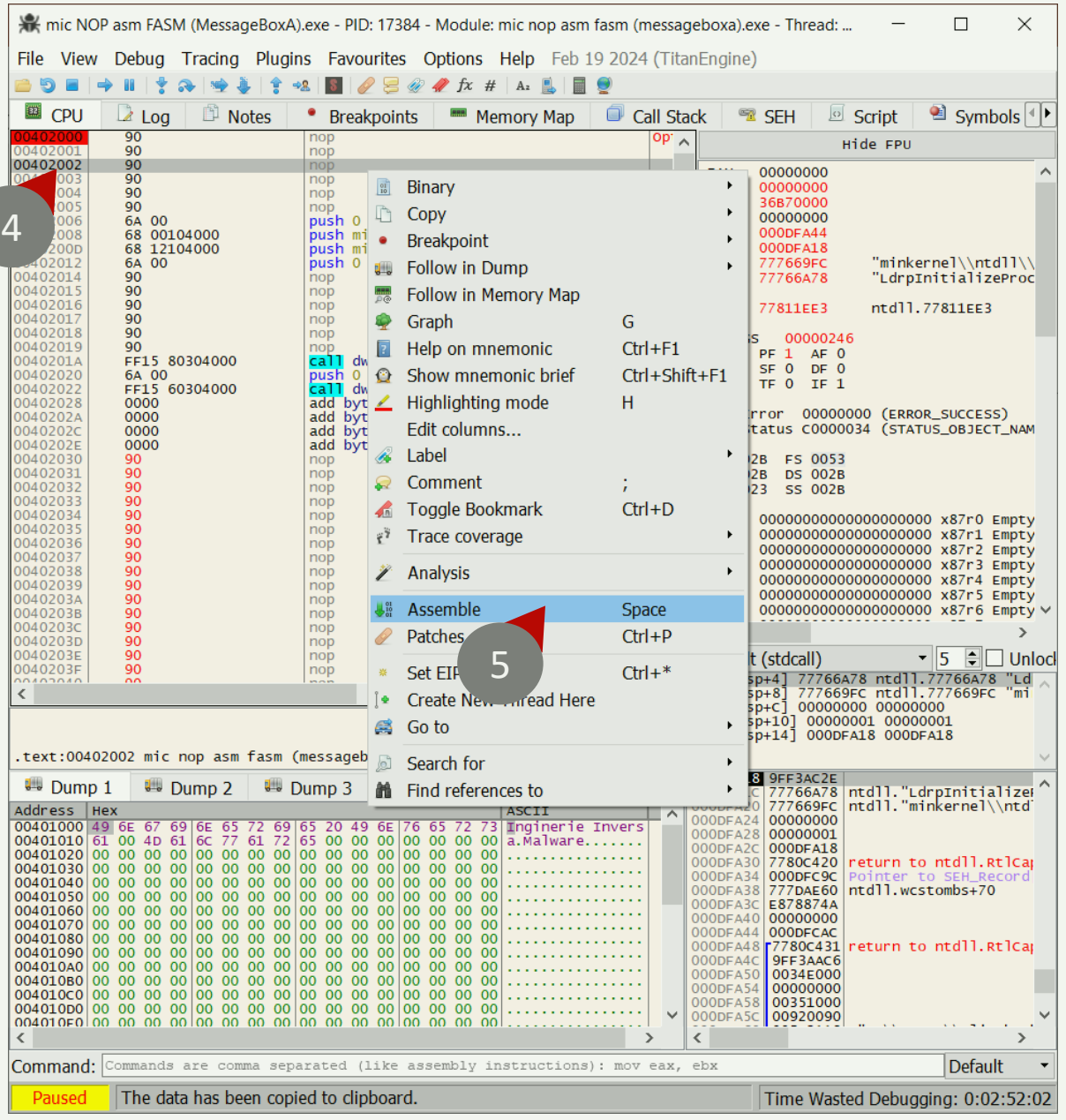


Dorim sa ajungem la adresa 00402030 și să revenim la adresa 00402046.

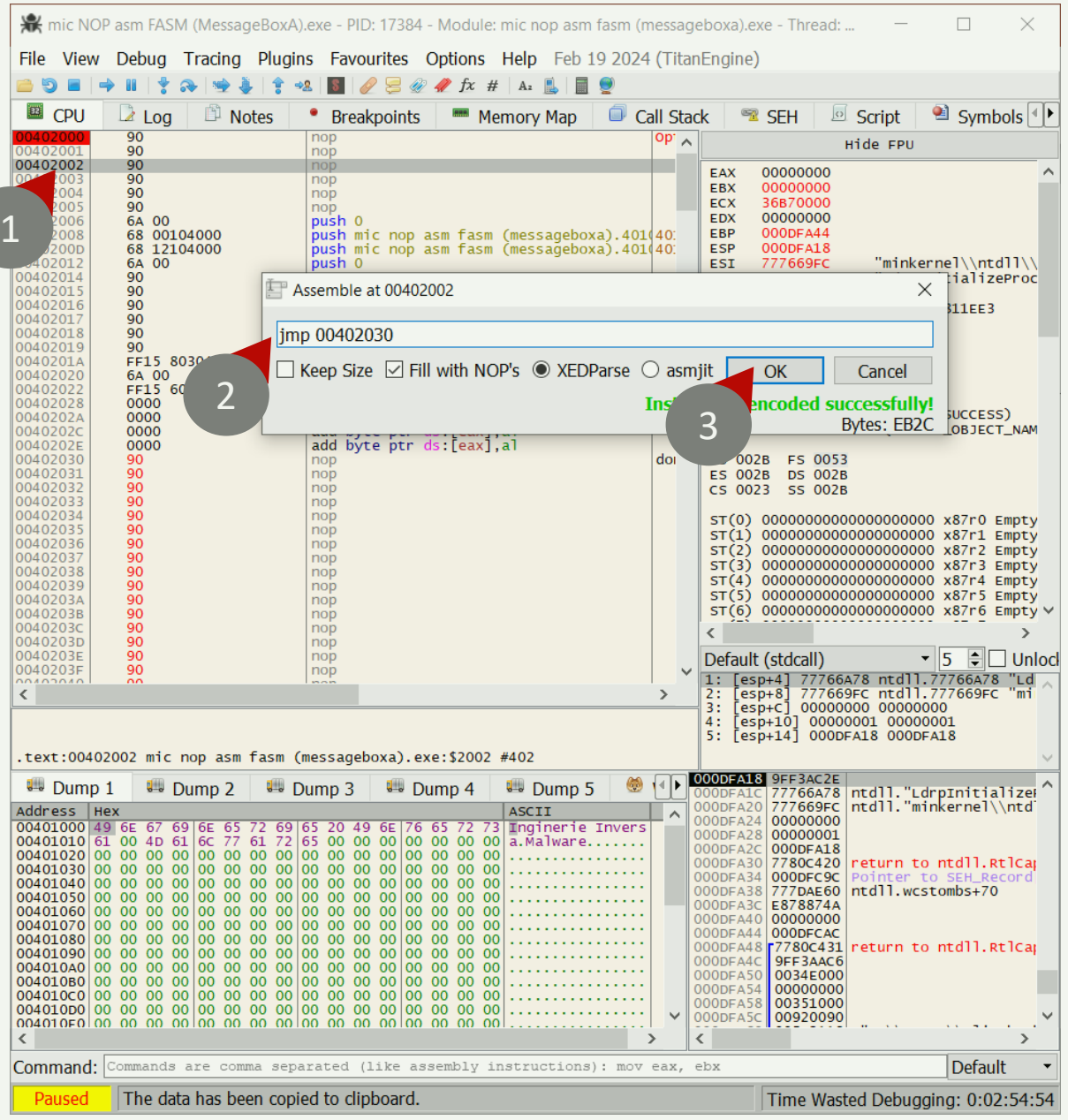
Este copiată adresa punctului de intrare în regiunea definită:



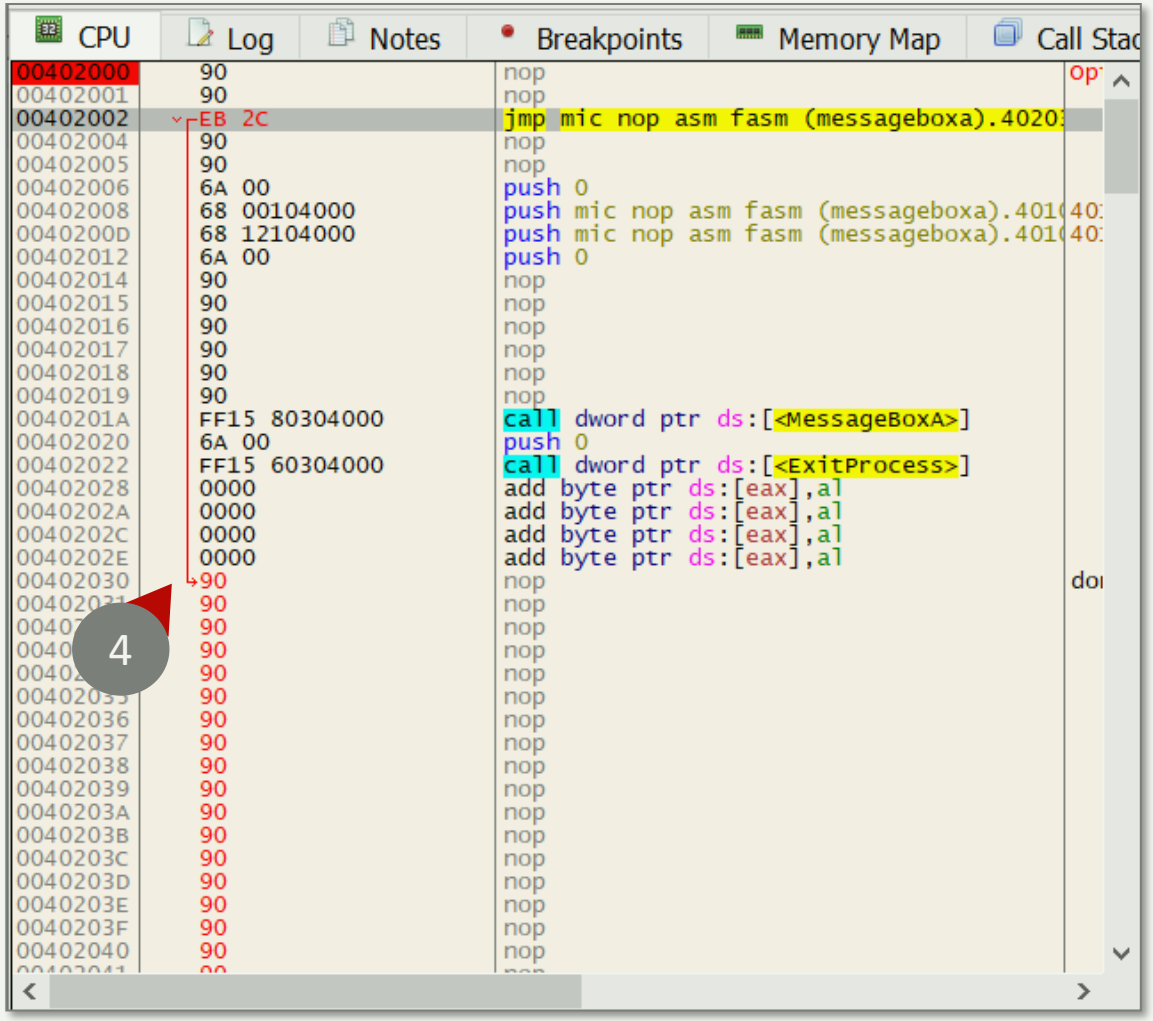
O instrucțiune JMP este inserată spre punctul de intrare în regiune:



Introduceți instrucțiunea JMP + adresa de unde începe regiunea:

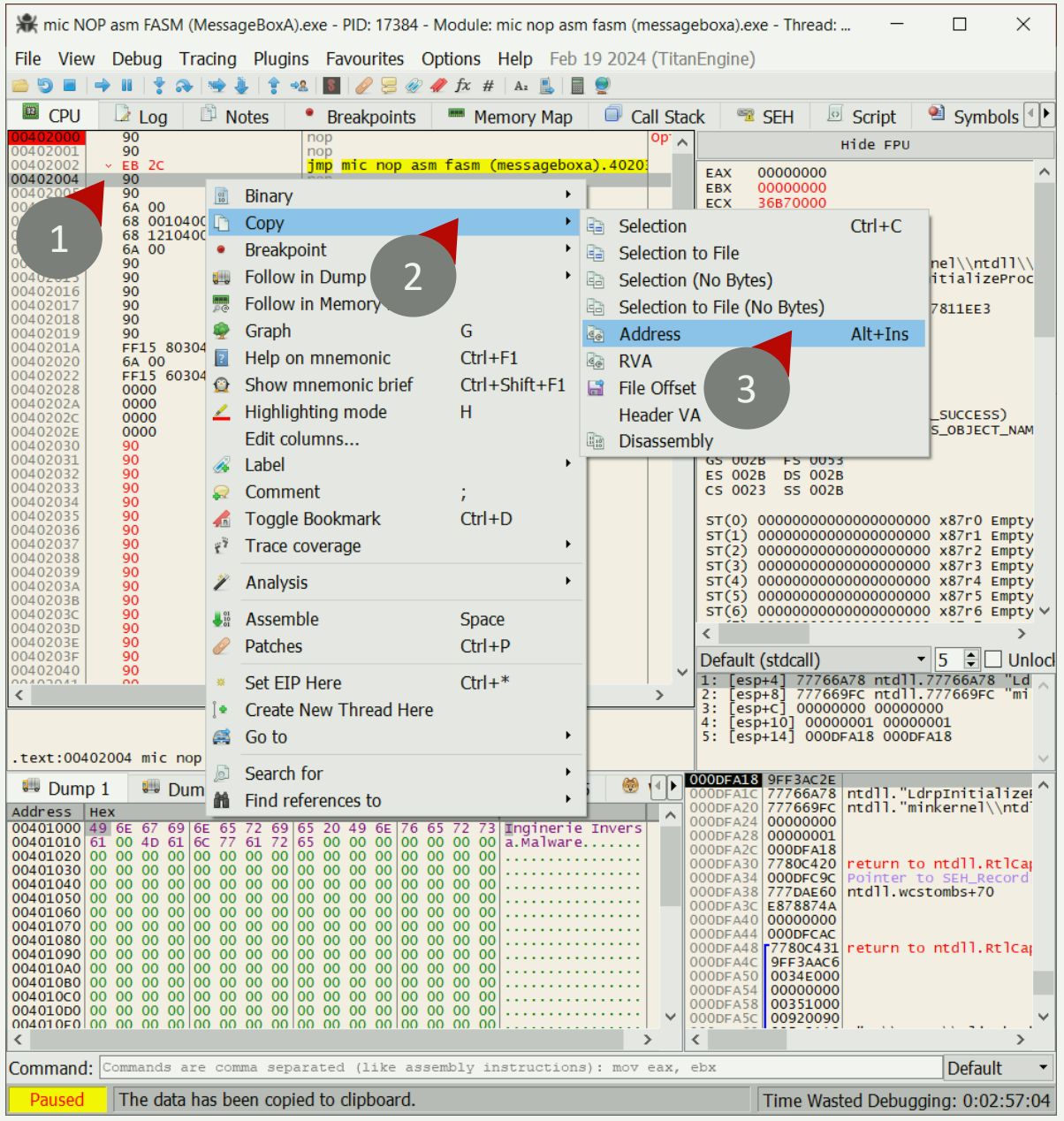


X32dbg permite vizualizarea salturilor cu ajutorul săgeților:

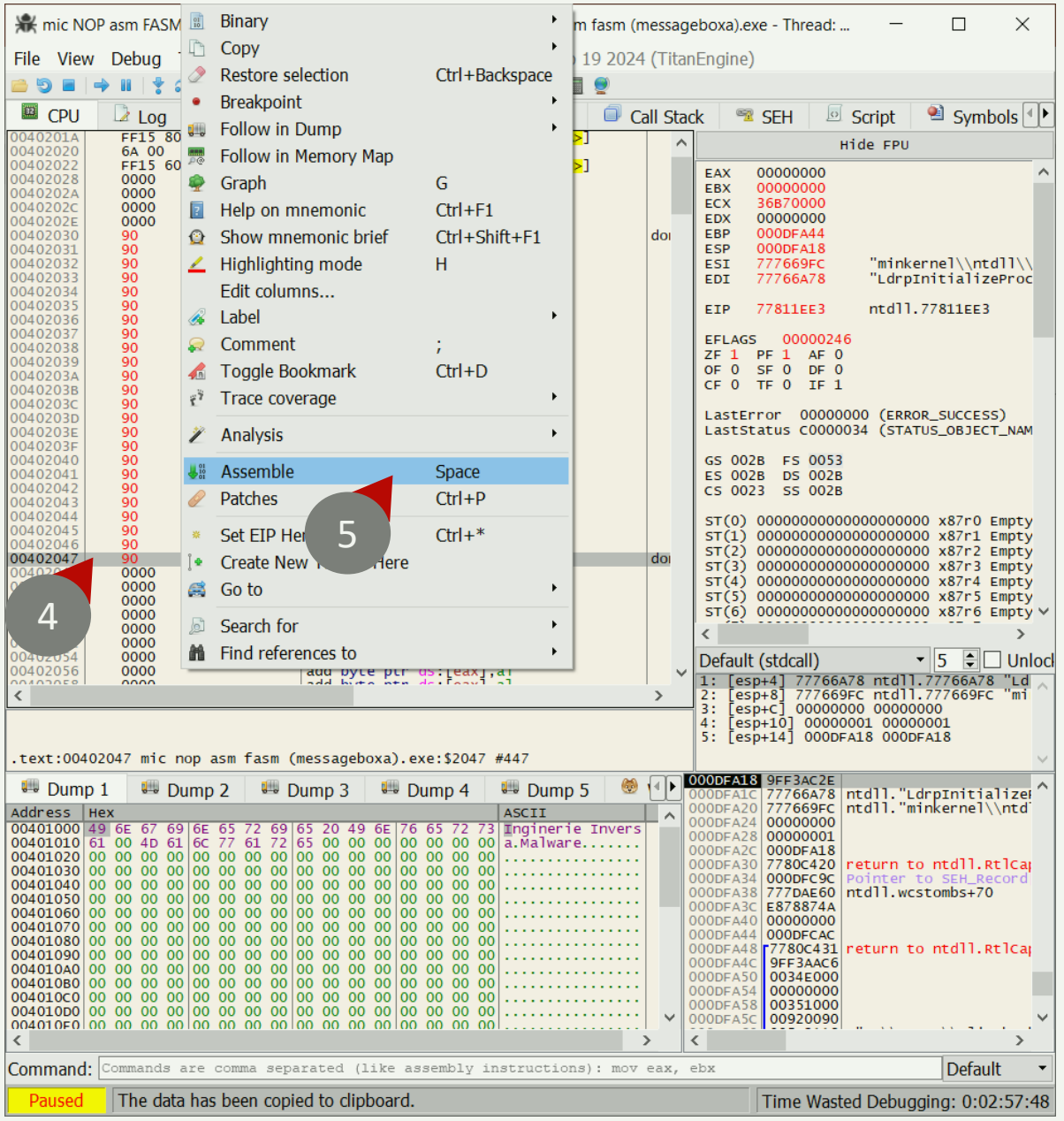


Saritura de la adresa 00402002, la zona imediat după adresa codului ! (00402030)

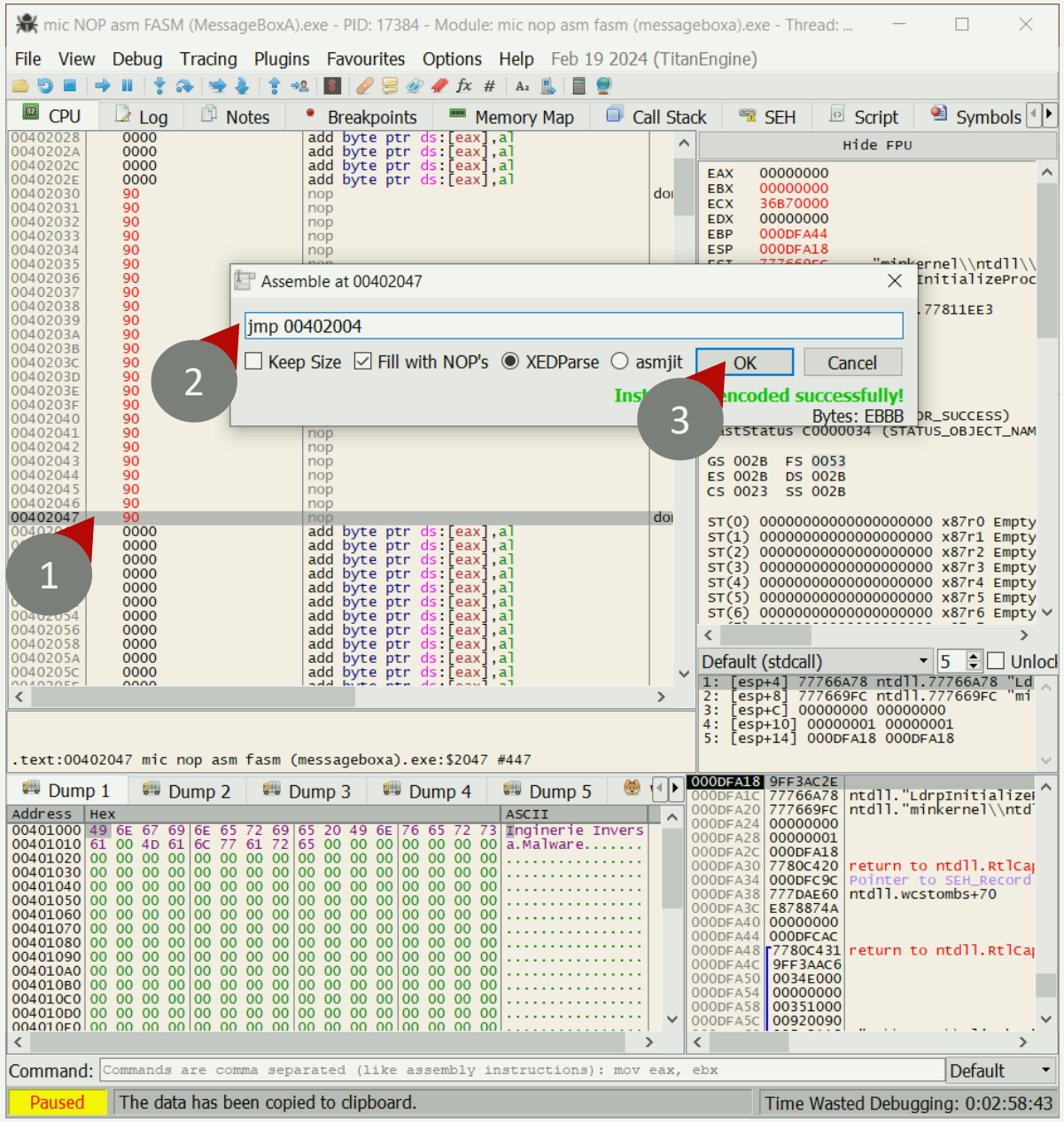
Adresa de după saltul inițial este copiată:



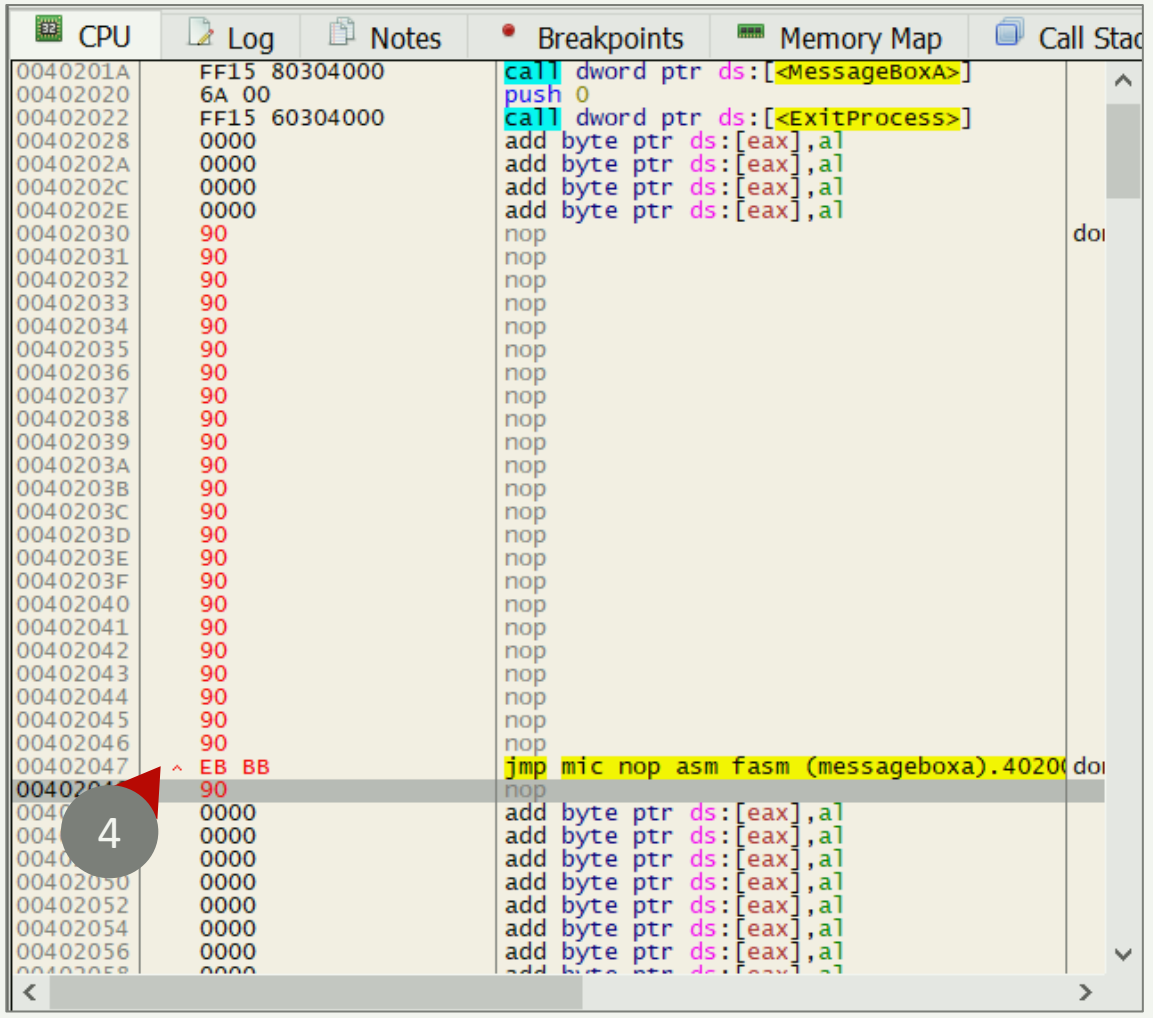
La punctul de ieșire al regiunii, se cere o inserție de instrucțiuni:



În punctul de ieșire din regiune, introduceți JMP + adresa copiată:



Instrucțiunea JMP + adresa copiată a fost scrisă!

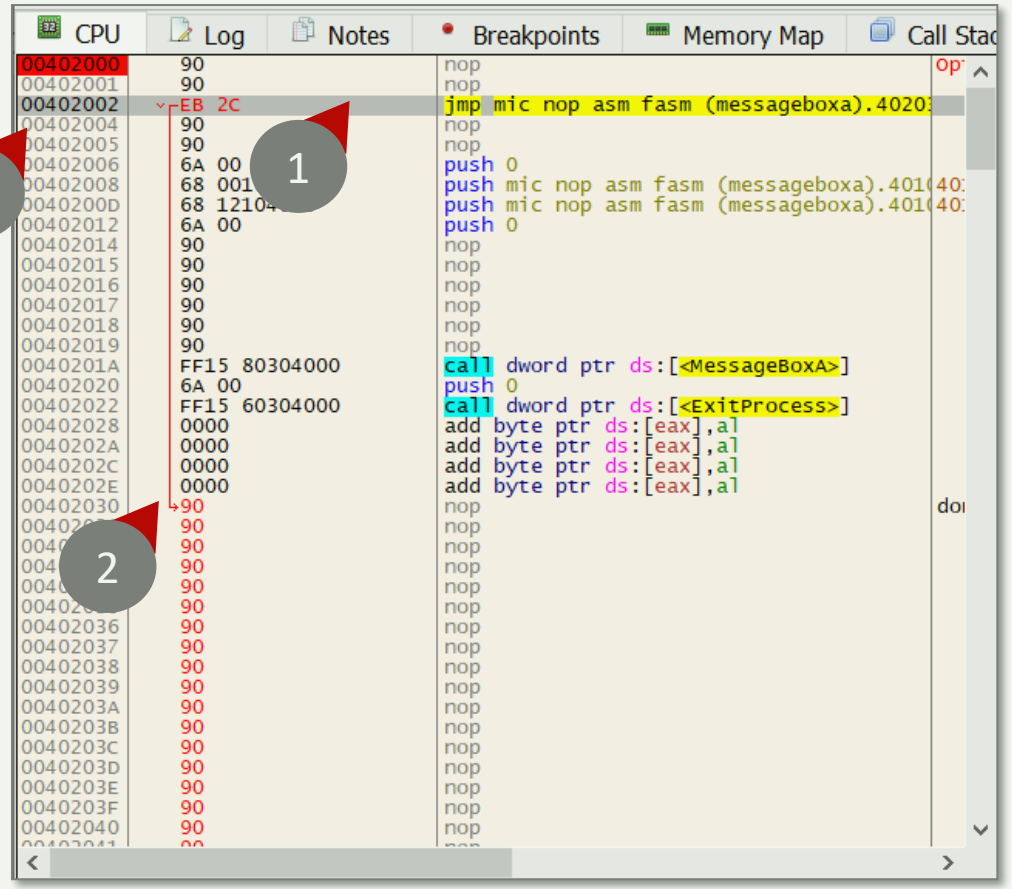


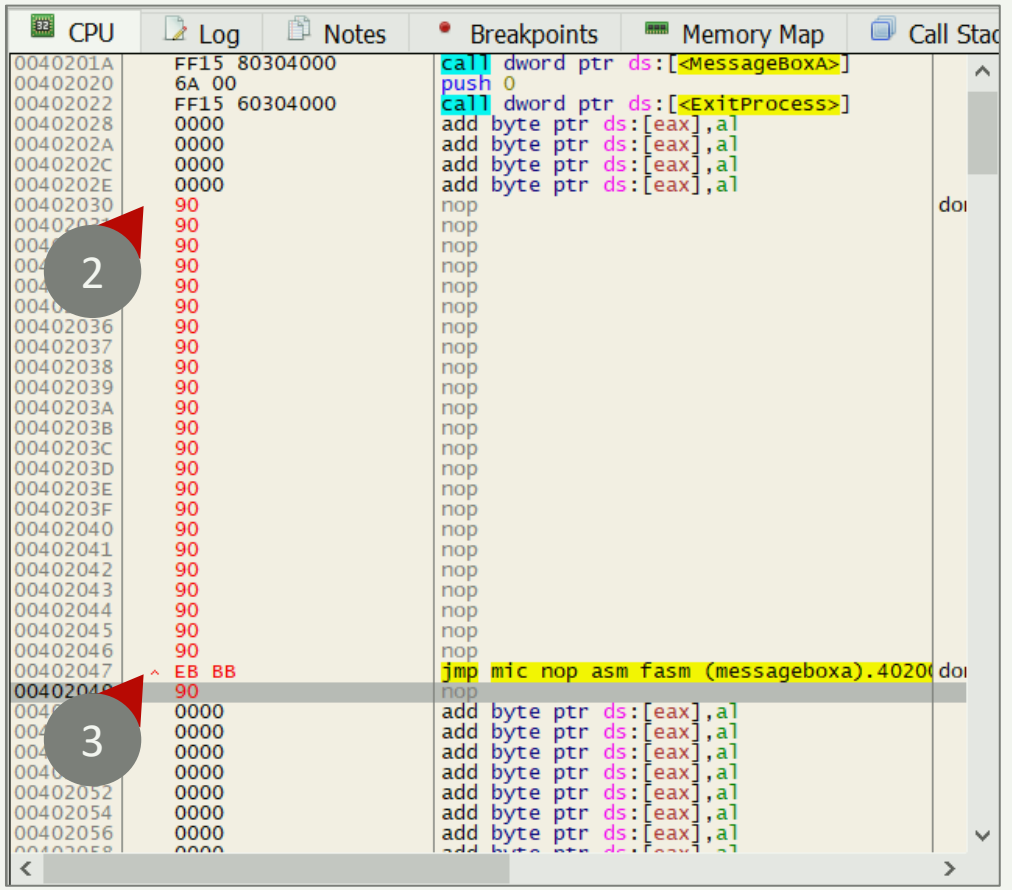
Saritura de la adresa 00402047, la zona imediat după adresa punctului de intrare! (00402004)

"PEȘTERA ÎN COD" (CODE CAVE)

ACUM, ÎN LOC DE INSTRUCȚIUNI NOP, PUTEM SCRIE COD

Avem un „buzunar" gol care poate fi folosit pentru a executa cod arbitrar (care se potrivește cu dimensiunea alocată de noi).



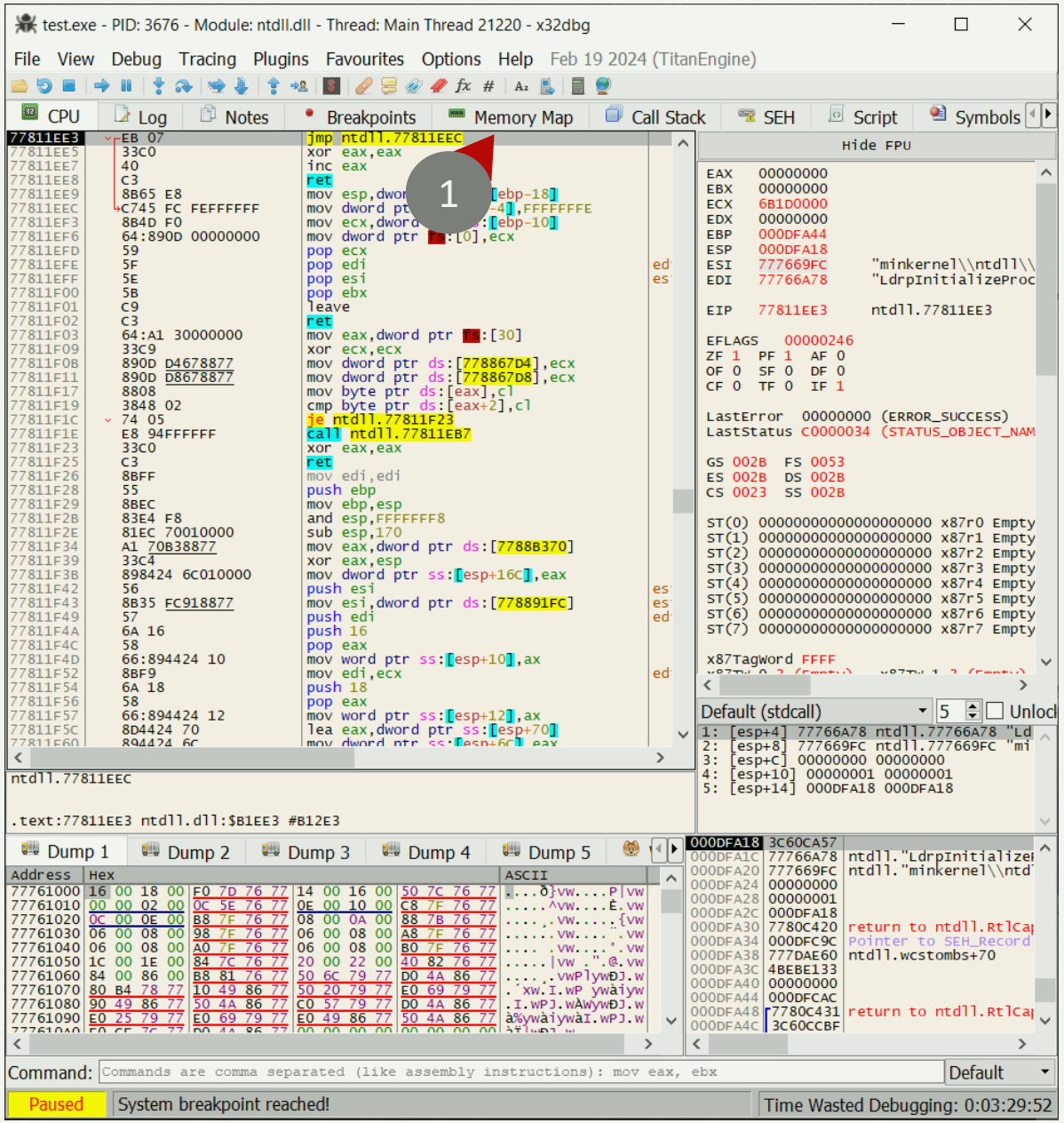


C.8.3
MODIFICARE ENTRY POINT ȘI INSERȚIE DE COD MAȘINĂ

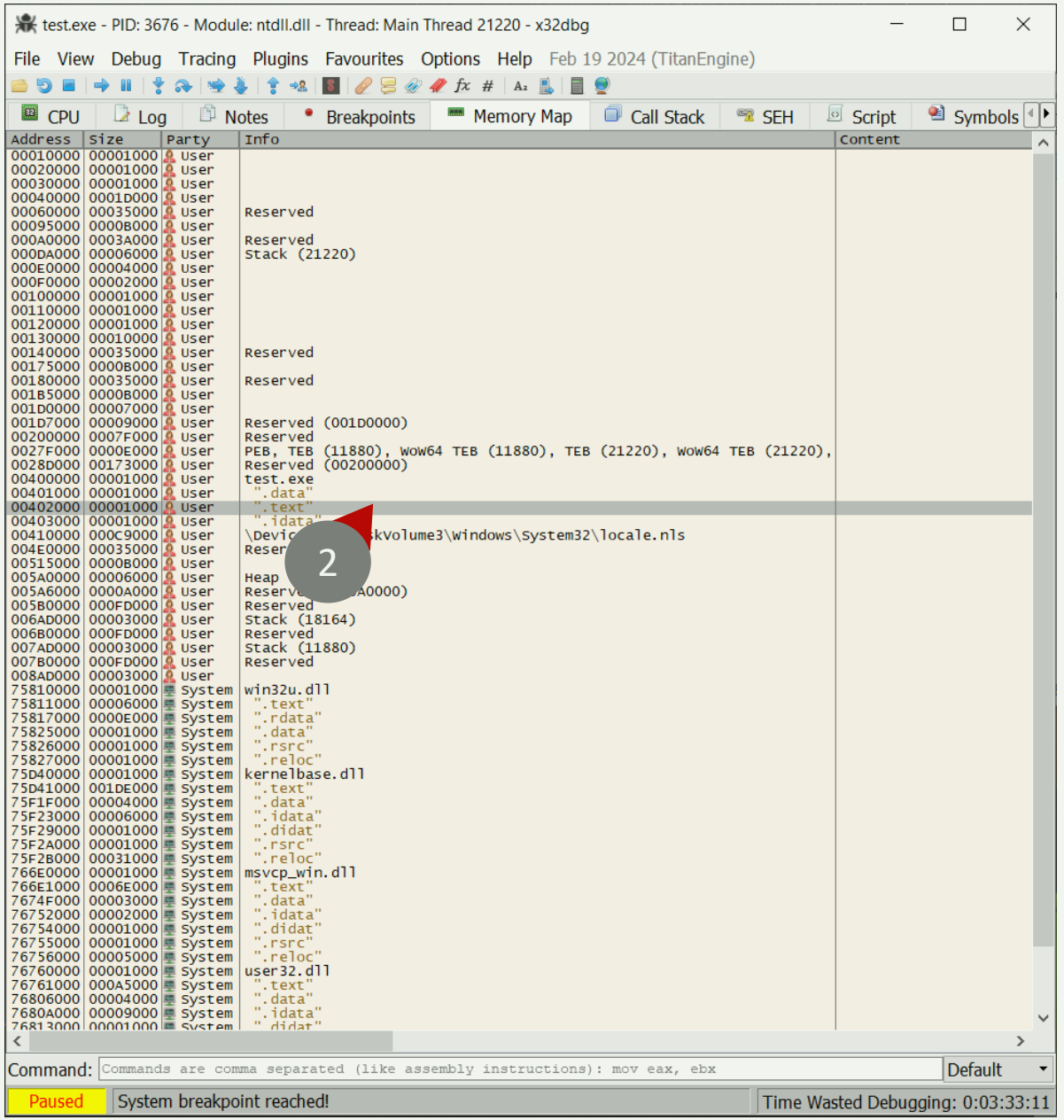
MODIFICARE ENTRY POINT
PUNCTUL DE INTRARE CĂTRE ALT SEGMENT

1. Schimbați punctul de intrare la un alt segment
2. Rulați segmentul în regiunea selectată
3. Salt la adresa de sub punctul de intrare

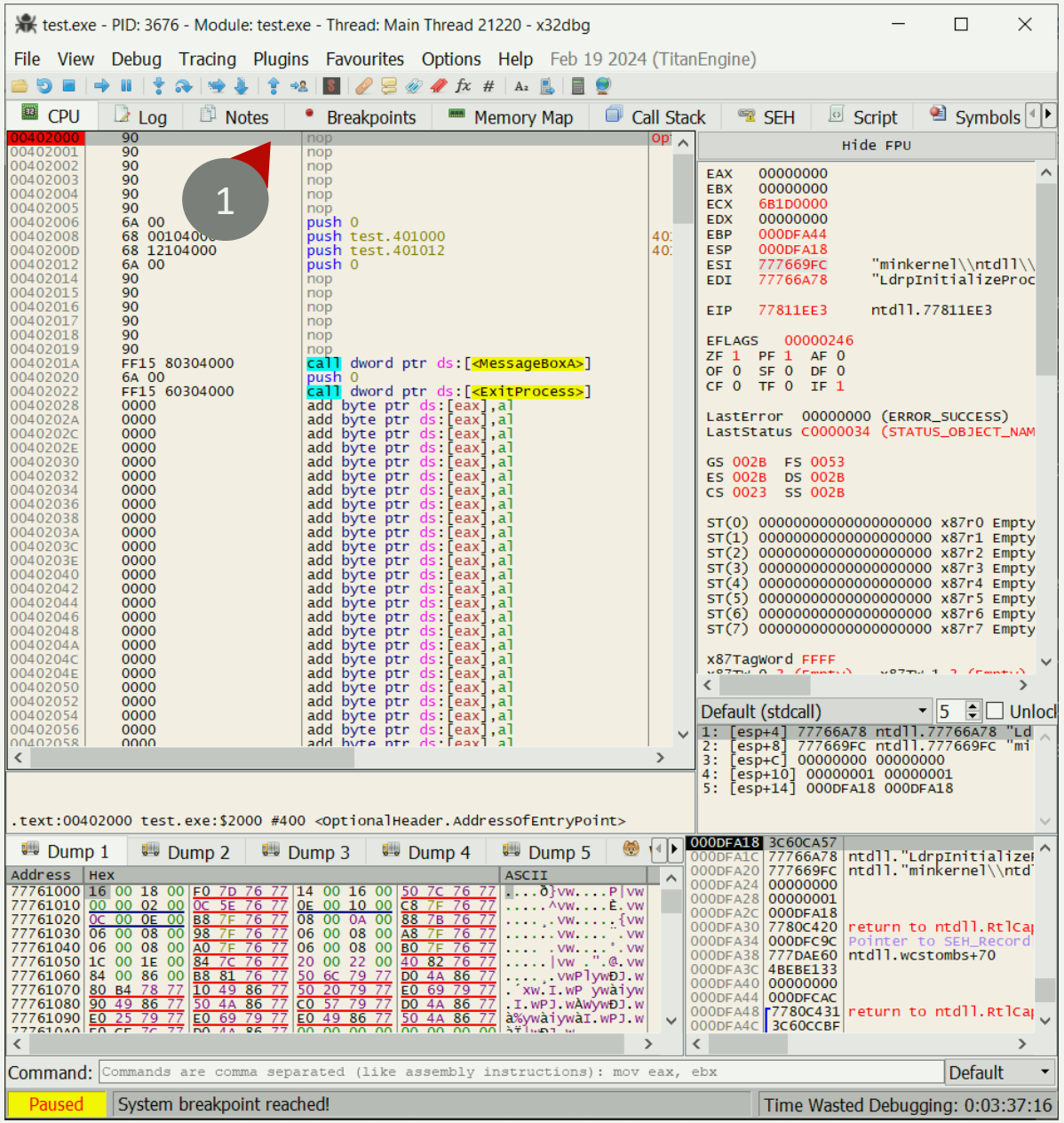
X32dbg afișează punctul de inițializare (nu punctul de intrare):



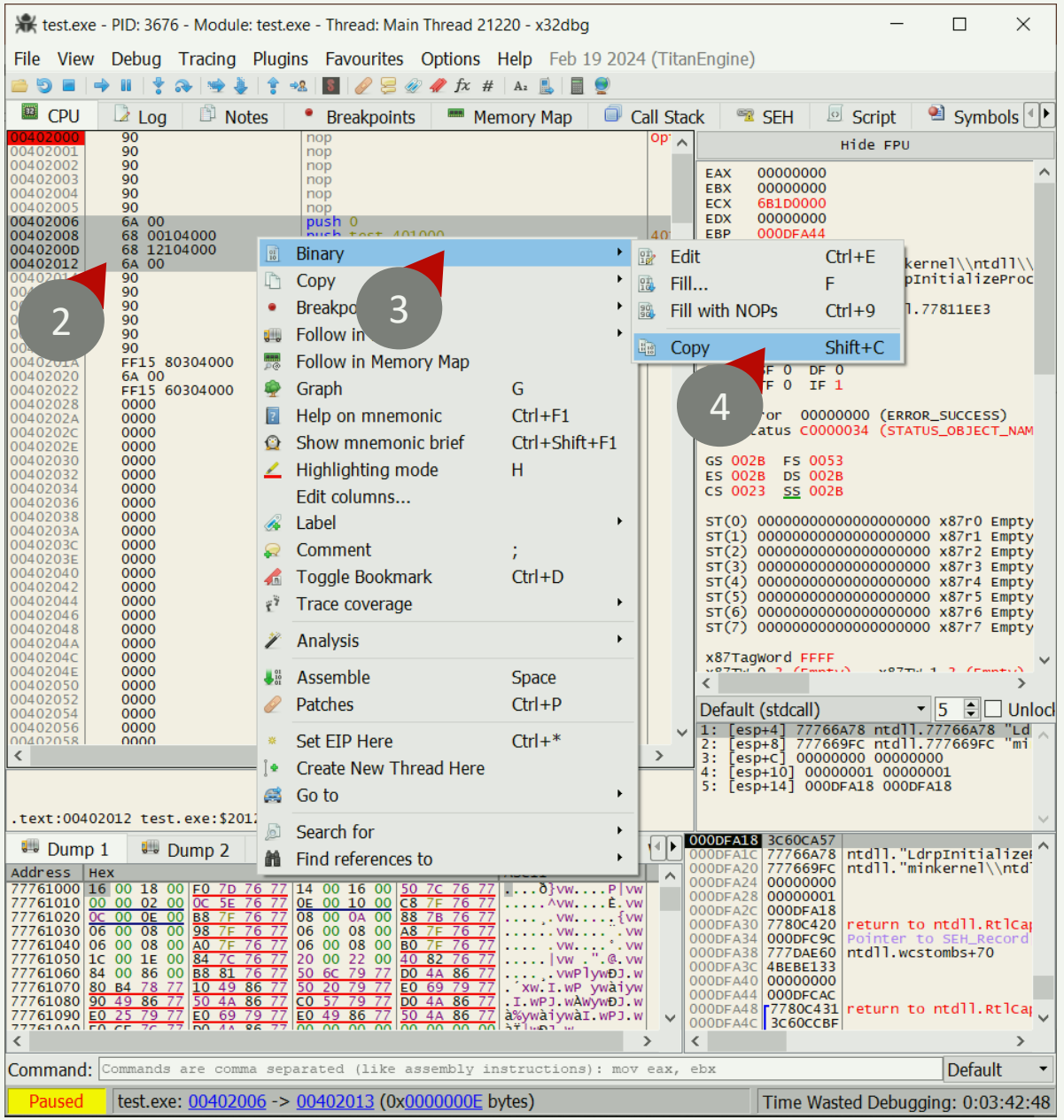
Selectați fila „Memory map” și accesați secțiunea **.text**:



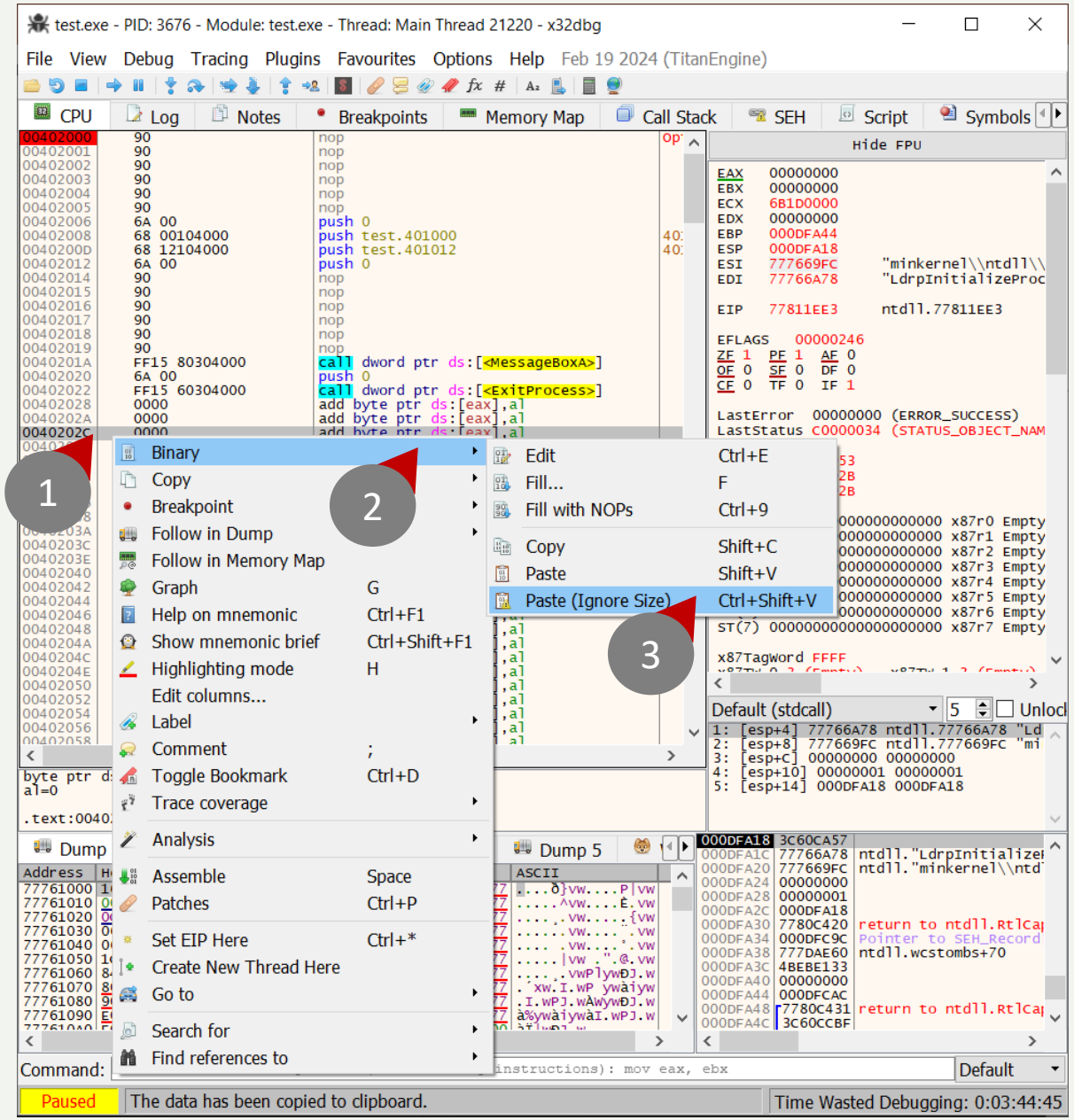
După accesarea secțiunii .text, x23dbg afișează punctul de intrare:



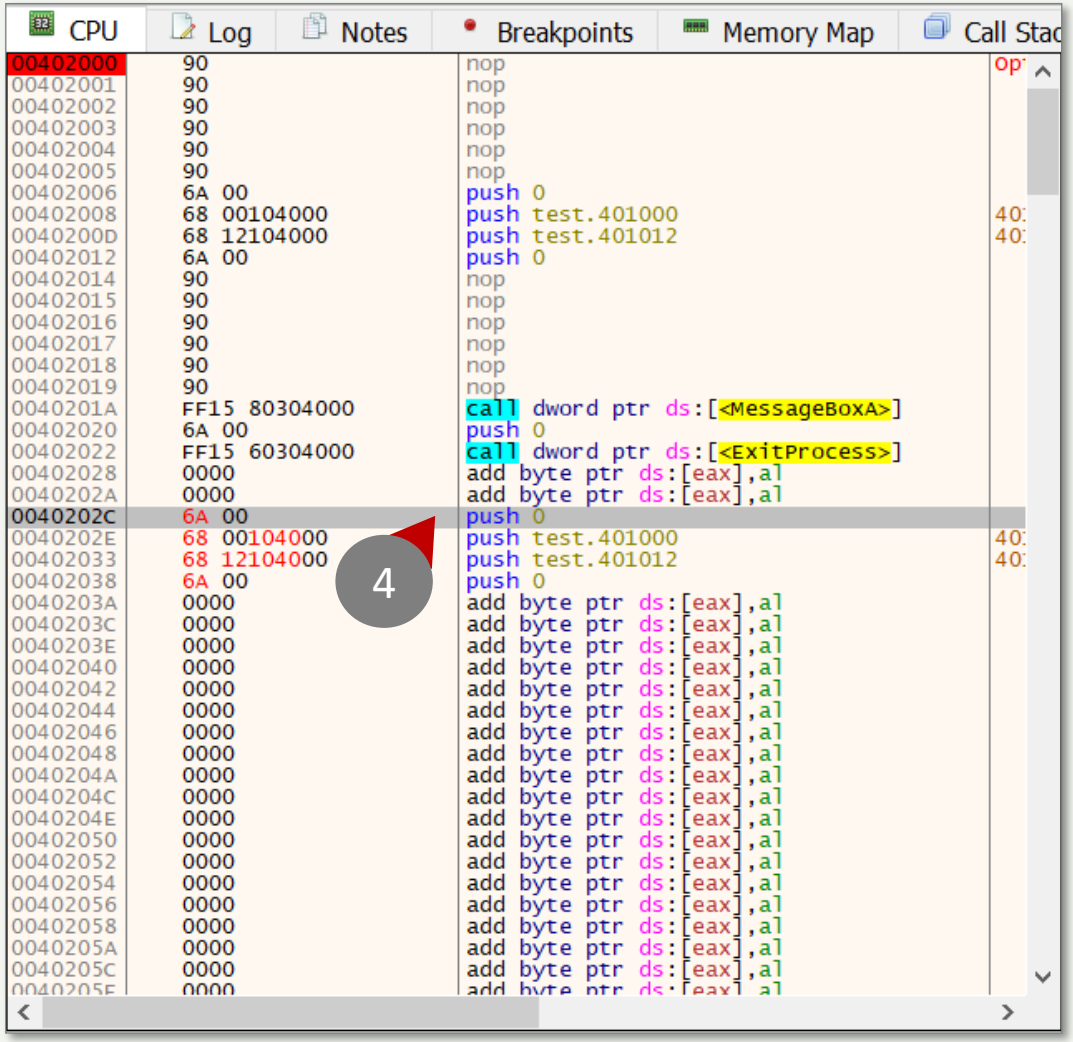
Selectați partea de cod, pe care apoi o copiați:



Clonăm cu „paste” codul într-o zonă liberă nedefinită:

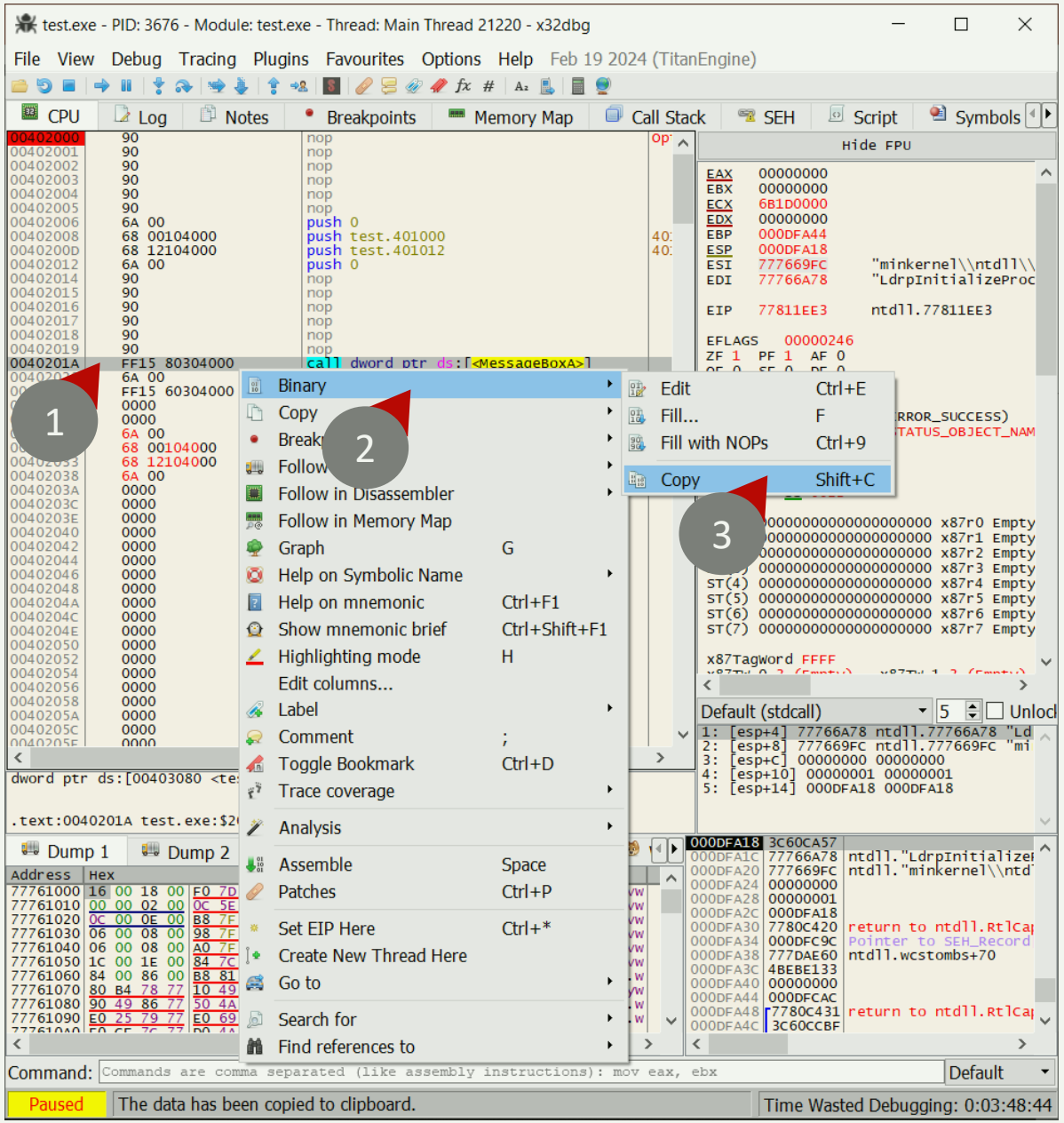


Codul clonat este afișat la adresa selectată:

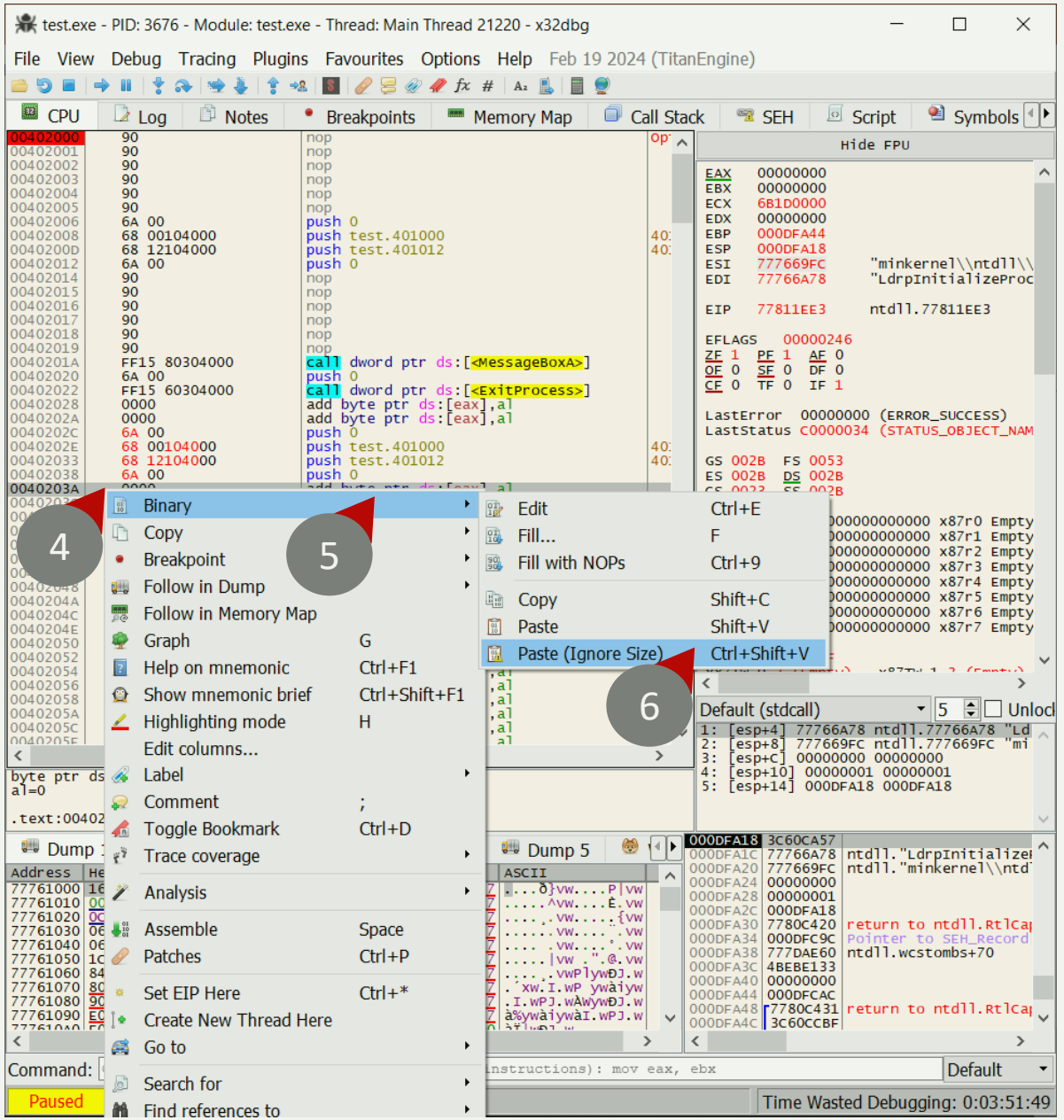


Copiați codul de la adresa 00402006 - 00402012 și inserați codul la adresa 0040202C - 00402038

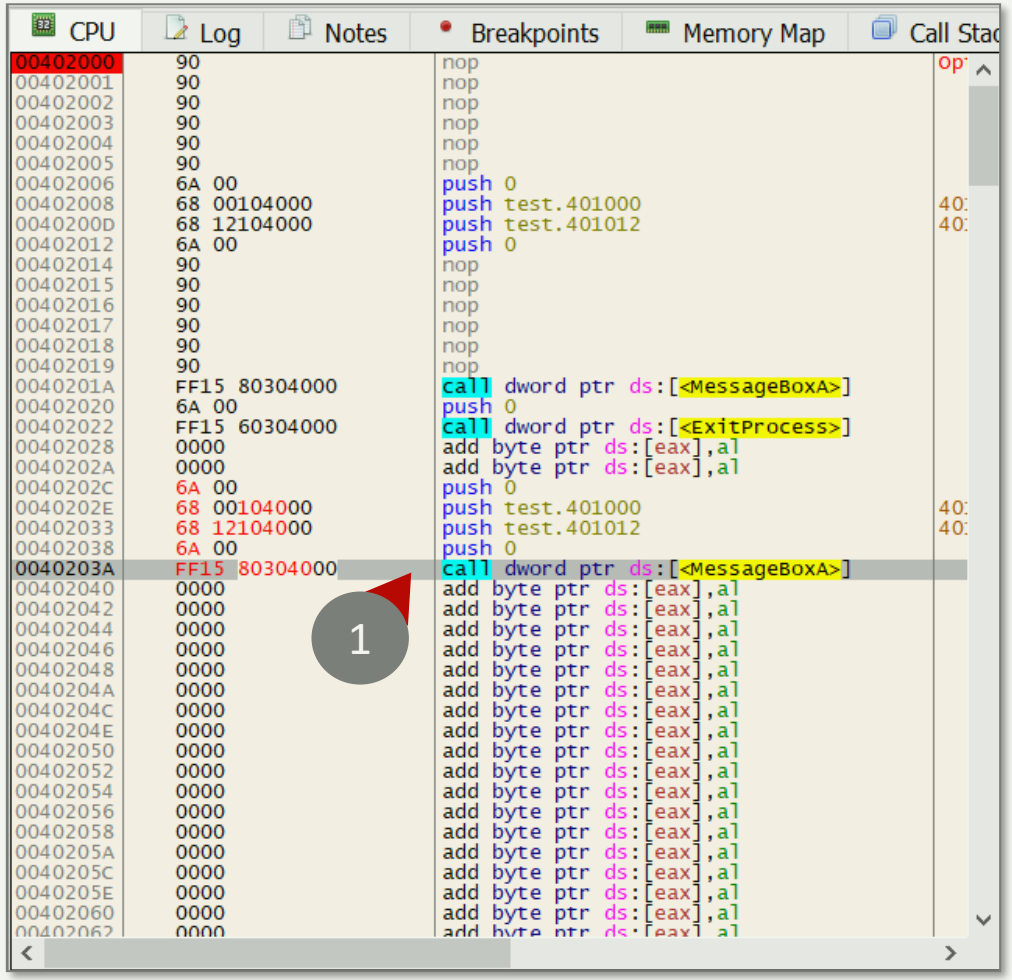
Copiem linia care apelează funcția MessageBox:



Clonăm această linie imediat după codul clonat anterior:

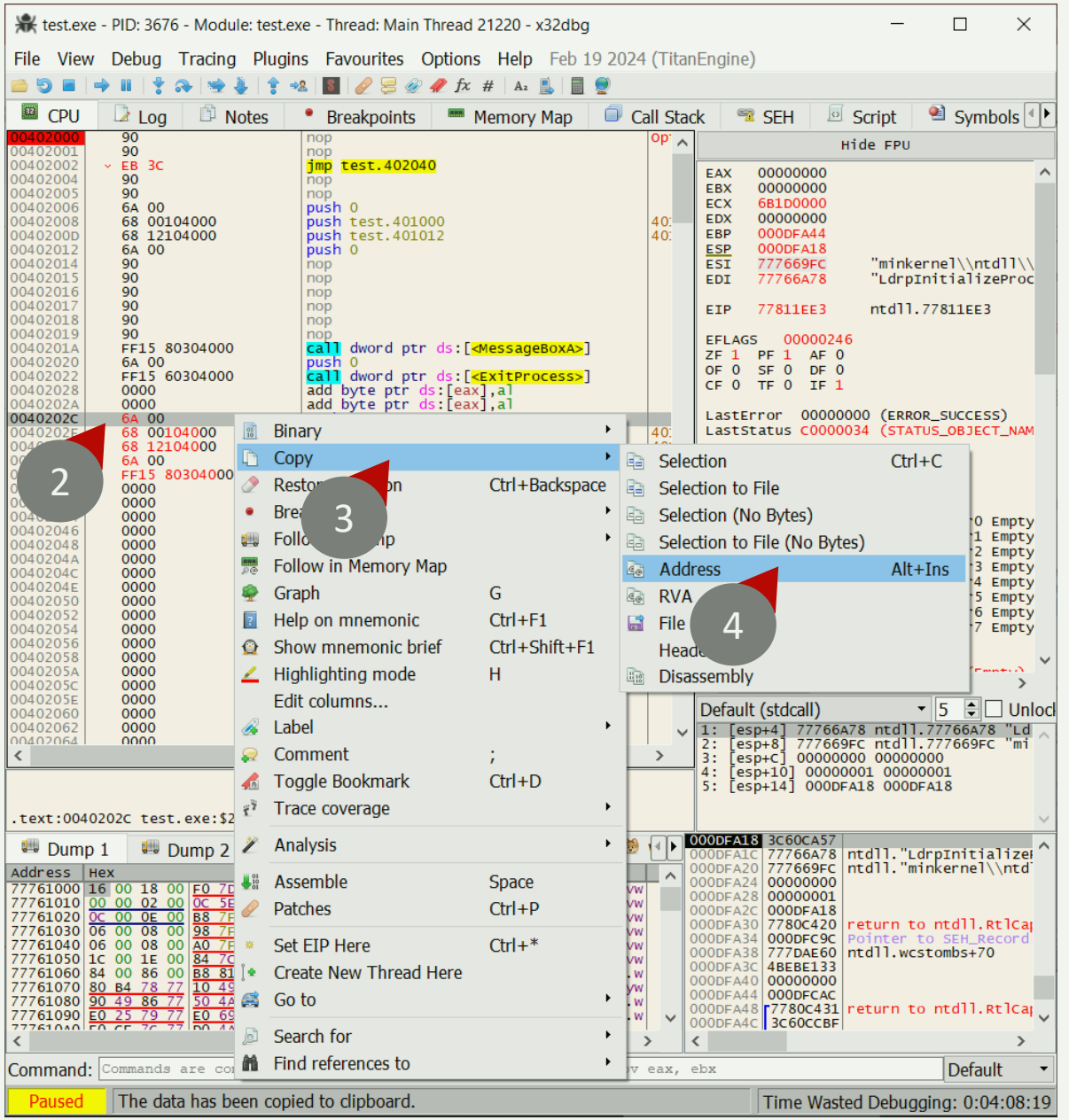


Avem cod clonat care nu poate fi accesat (ExitProcess):

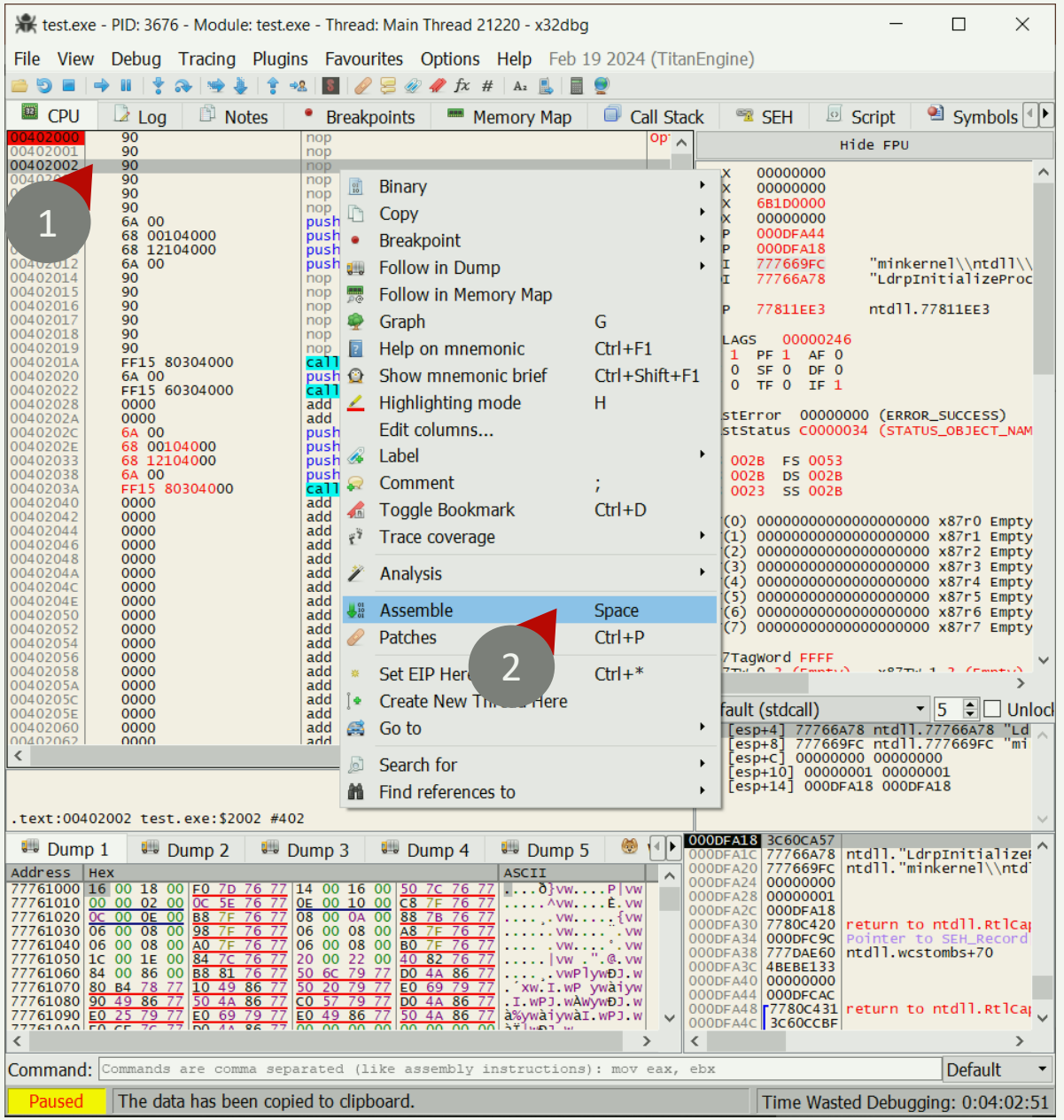


Copiați linia de cod de la adresa 0040201A și inserați linia de cod la adresa 0040203A.

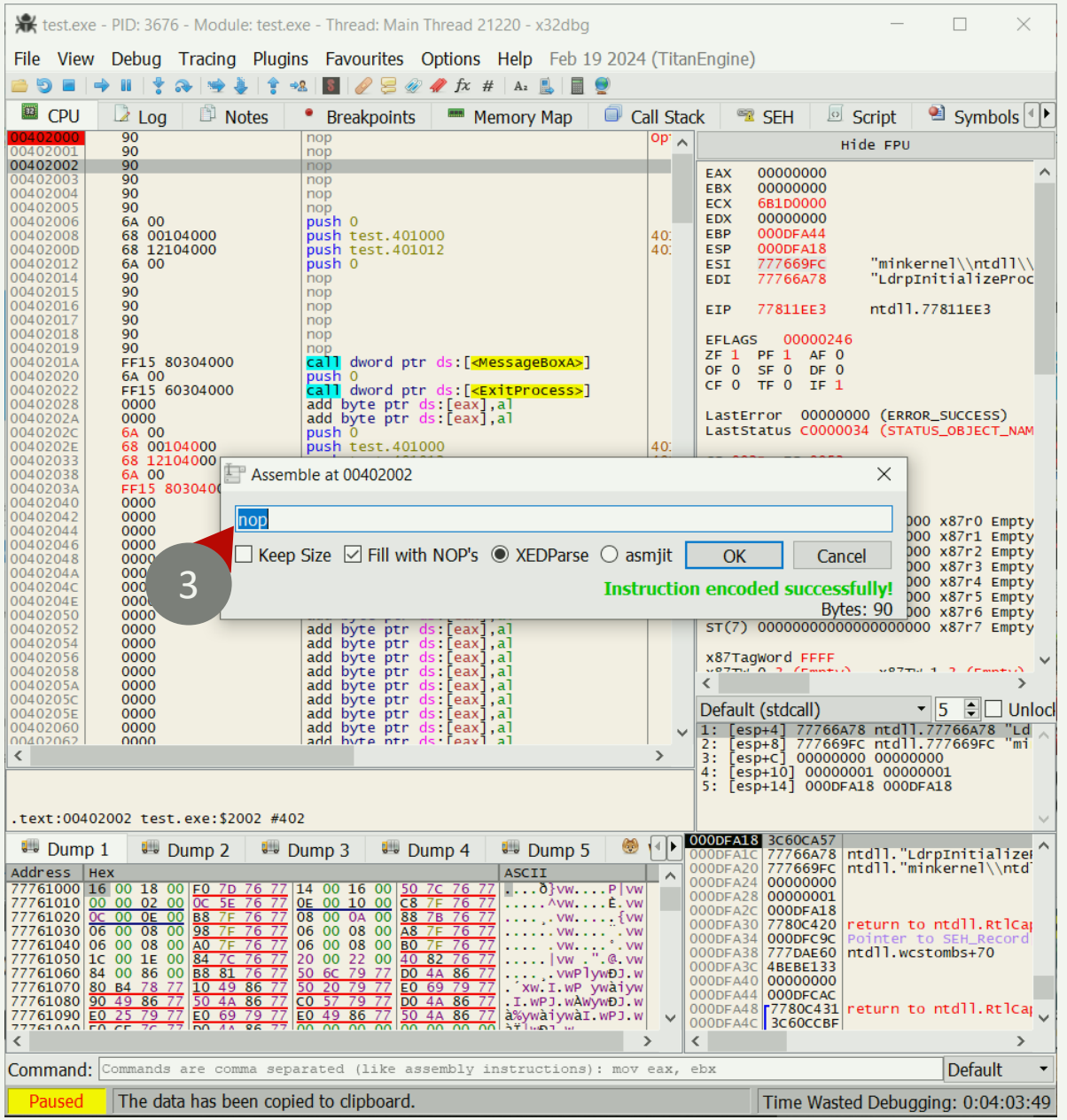
Copiem adresa de unde începe codul clonat:



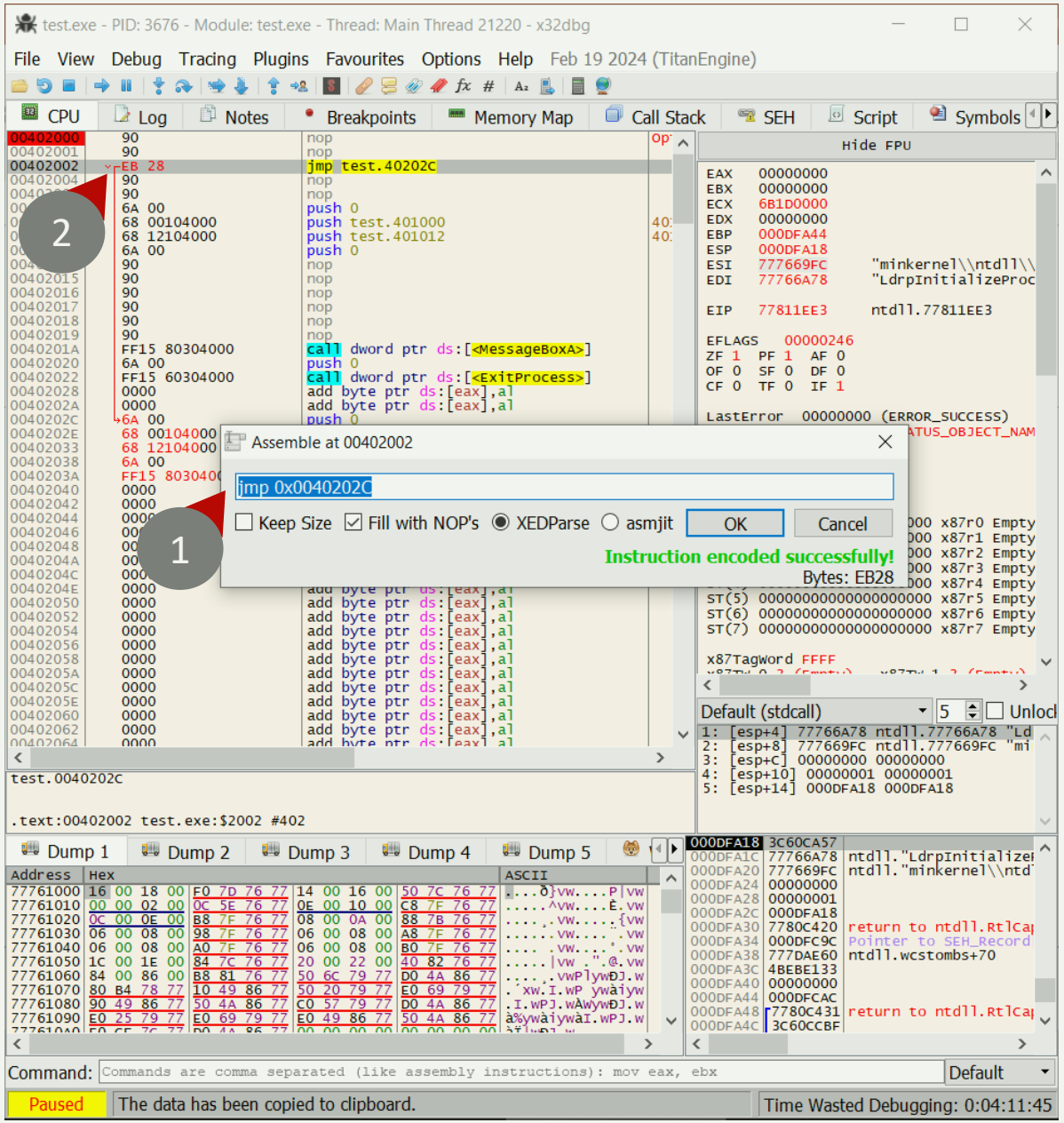
Solicităm o introducere cu instrucțiuni:



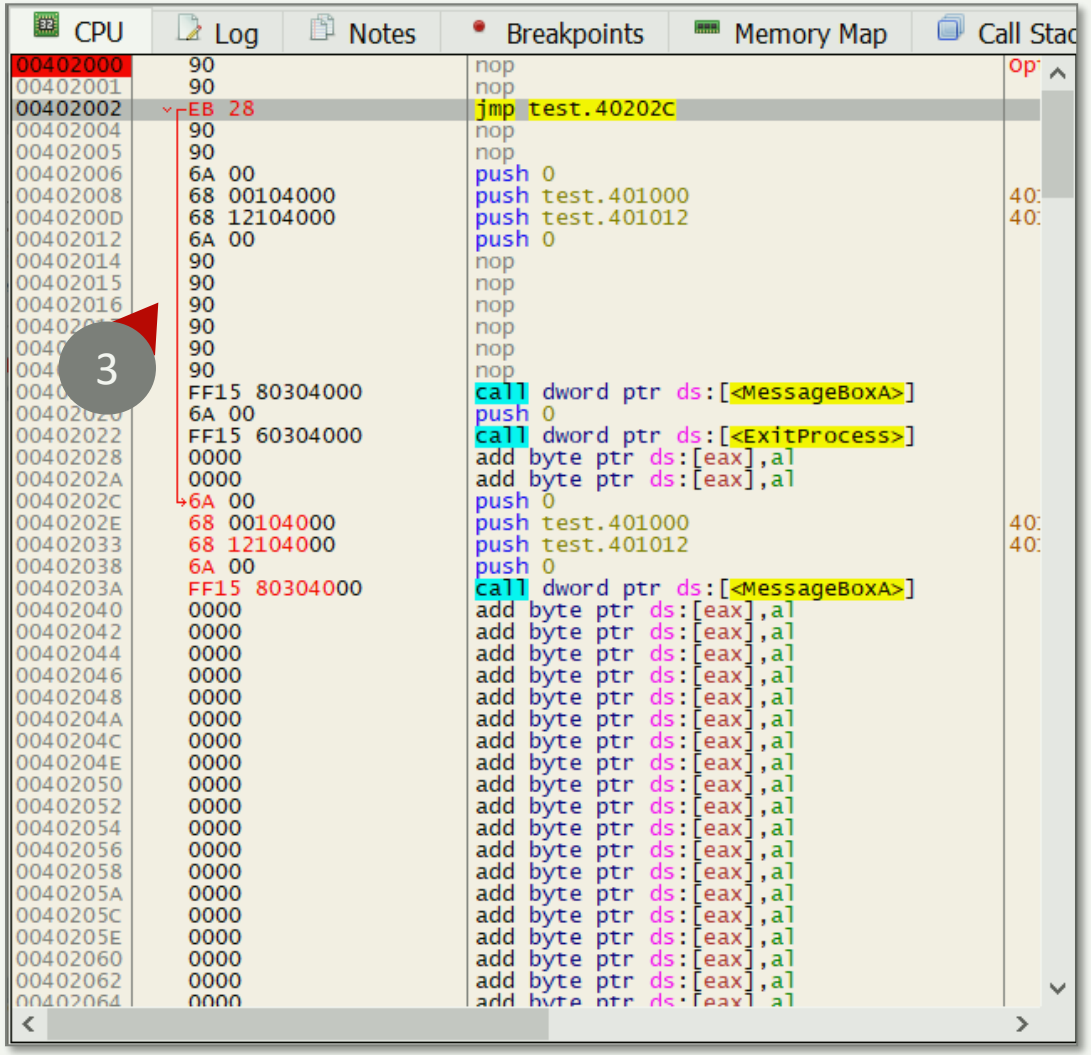
Apare fereastra unde instrucțiunea **nop** este înlocuită cu **jmp**:



Introducem un salt necondiționat la adresa copiată anterior:

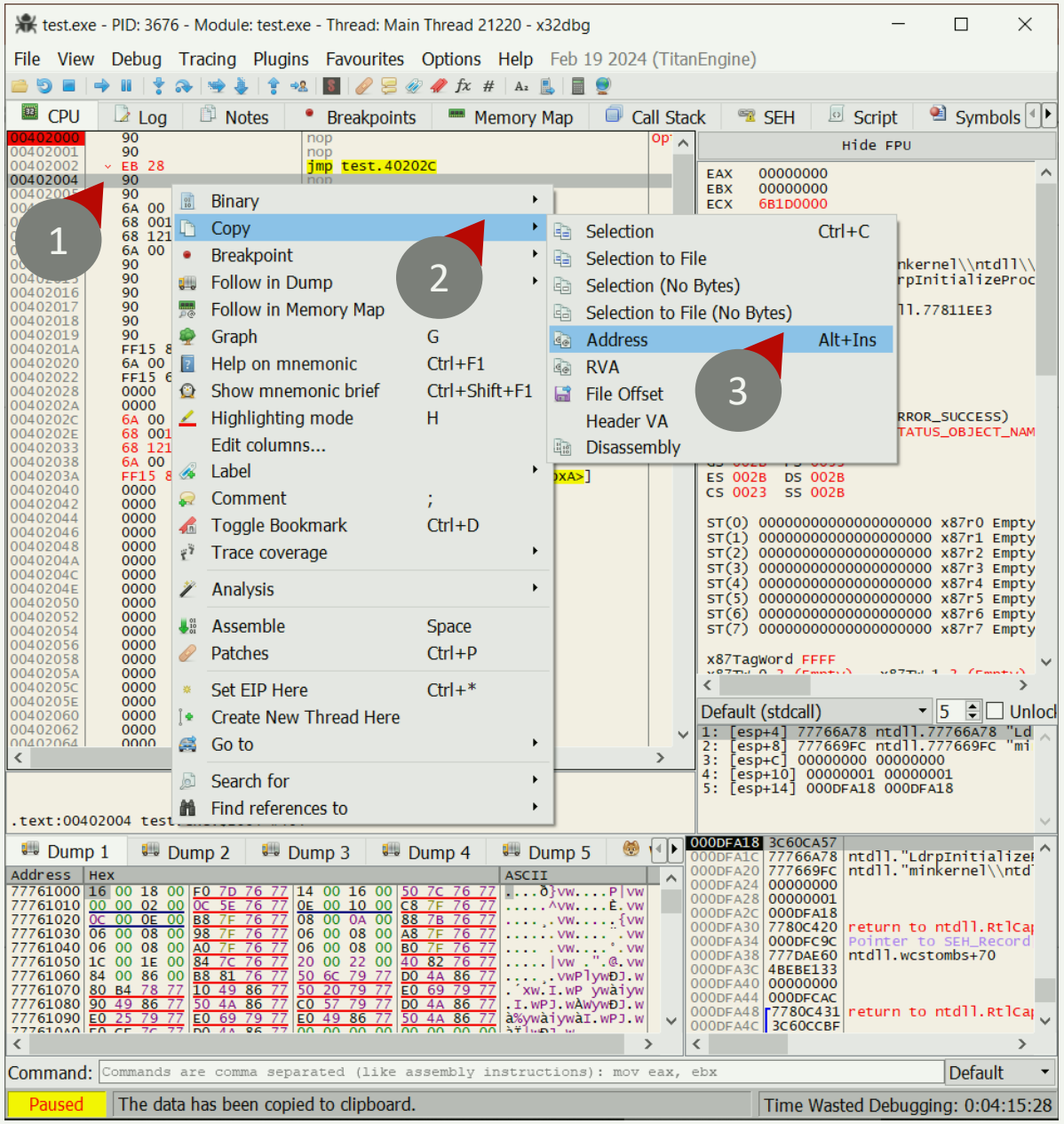


Clonăm această linie imediat după codul clonat anterior:

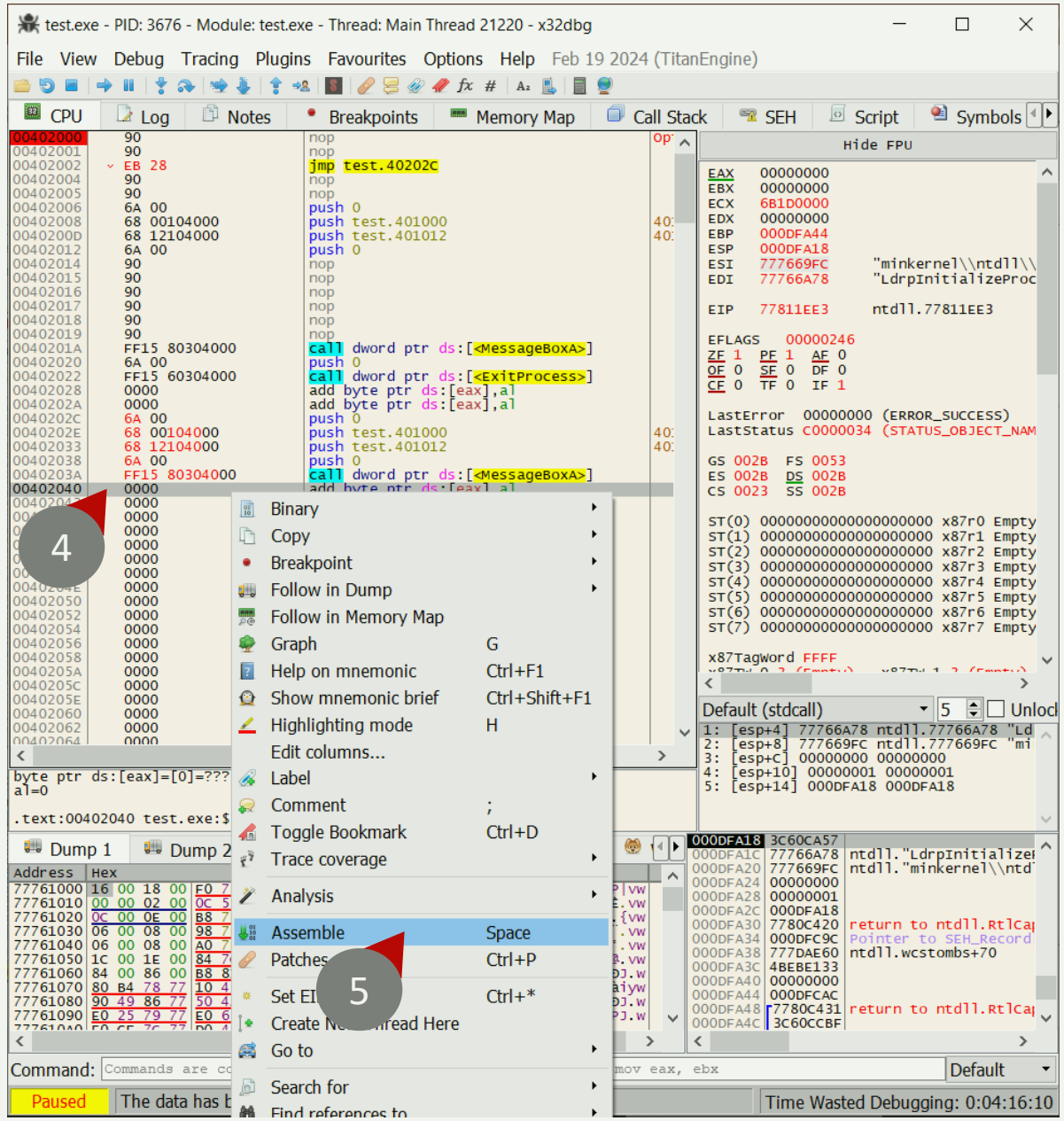


Facem un salt de la adresa 00402002 la adresa 0040202C.

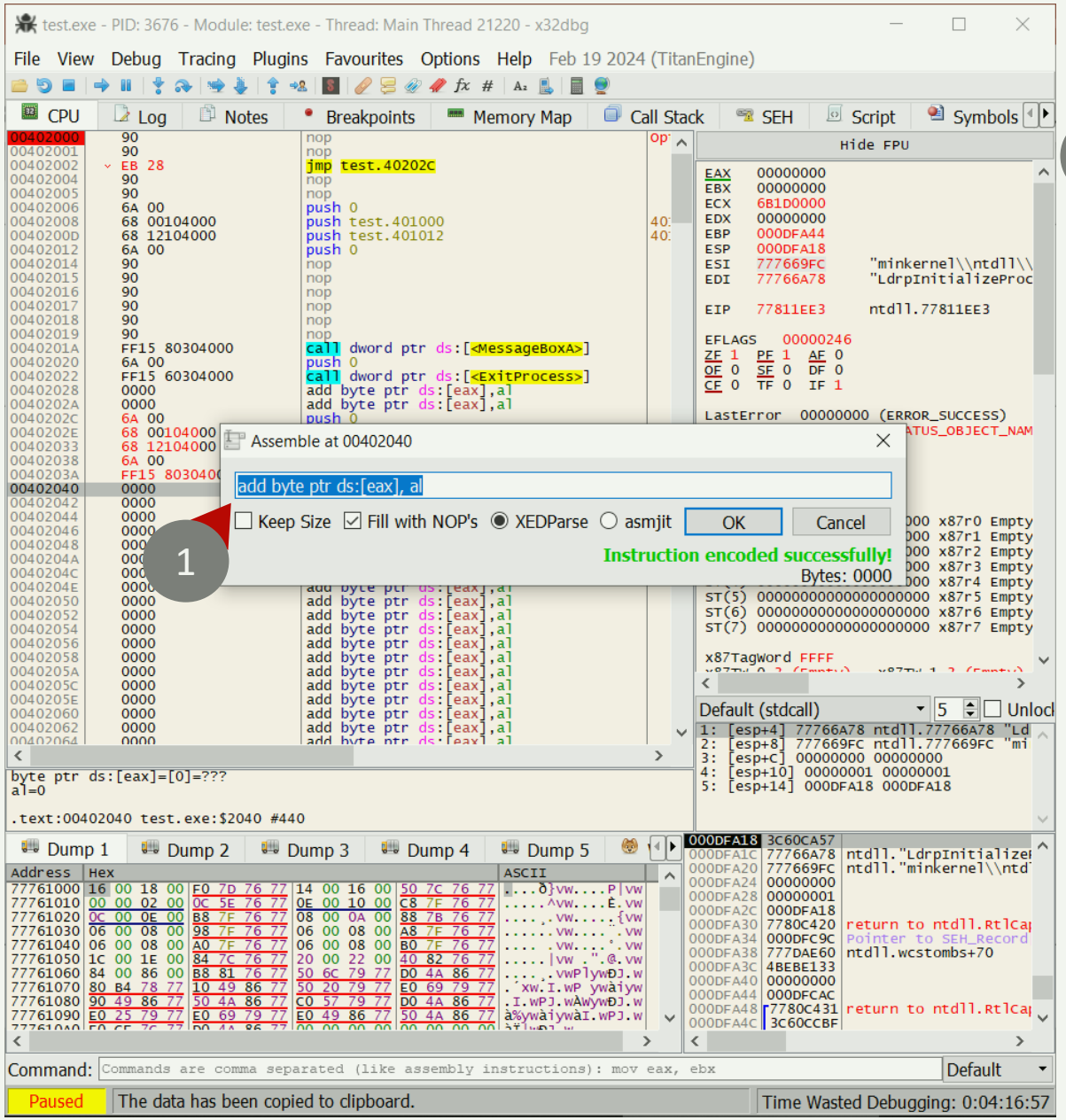
Adresa după saltul inițial la codul nostru este copiată:

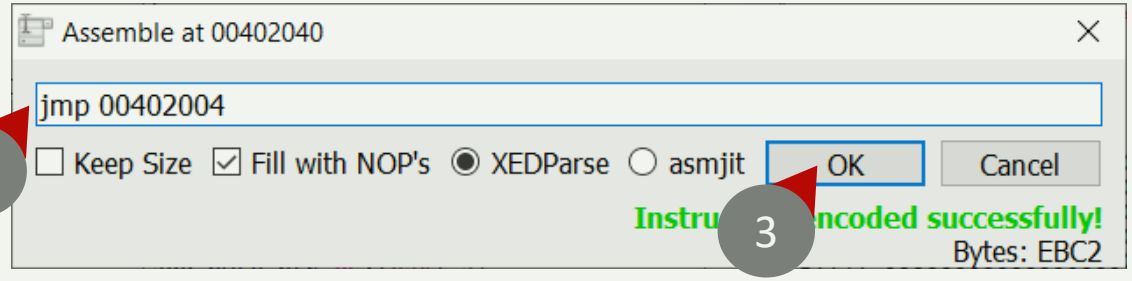


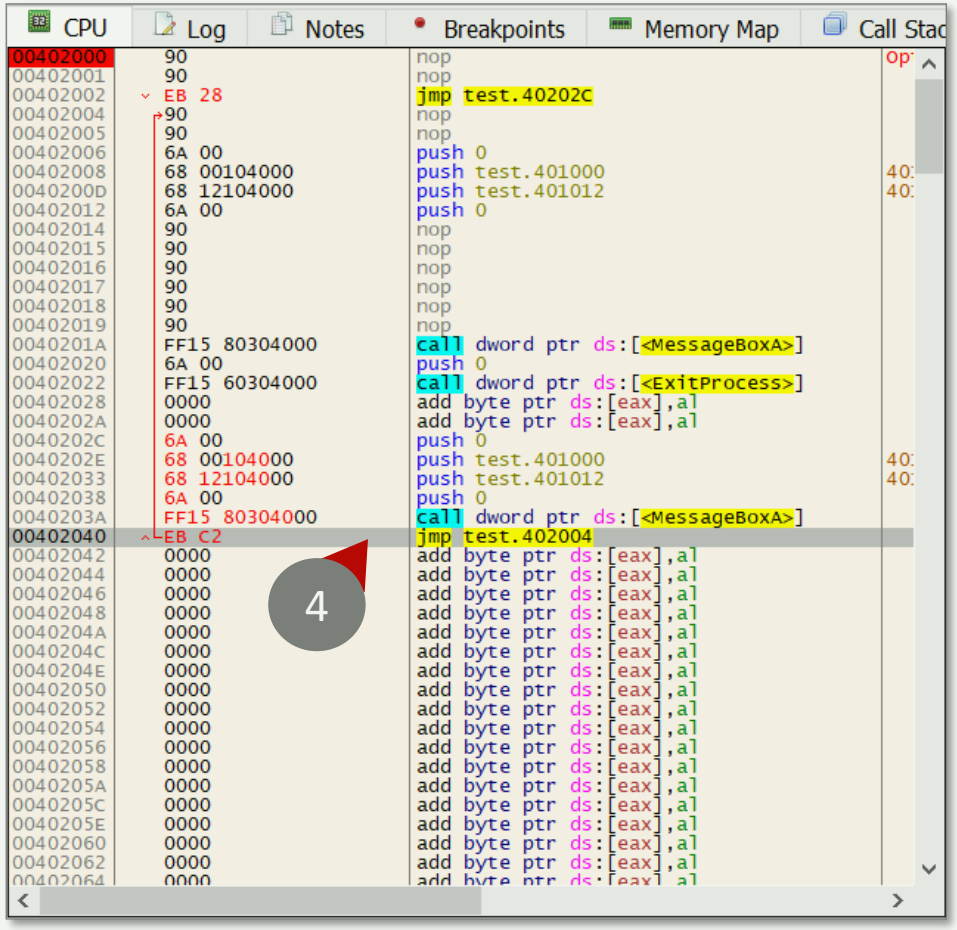
După codul clonat, se solicită o introducere de instrucțiuni:



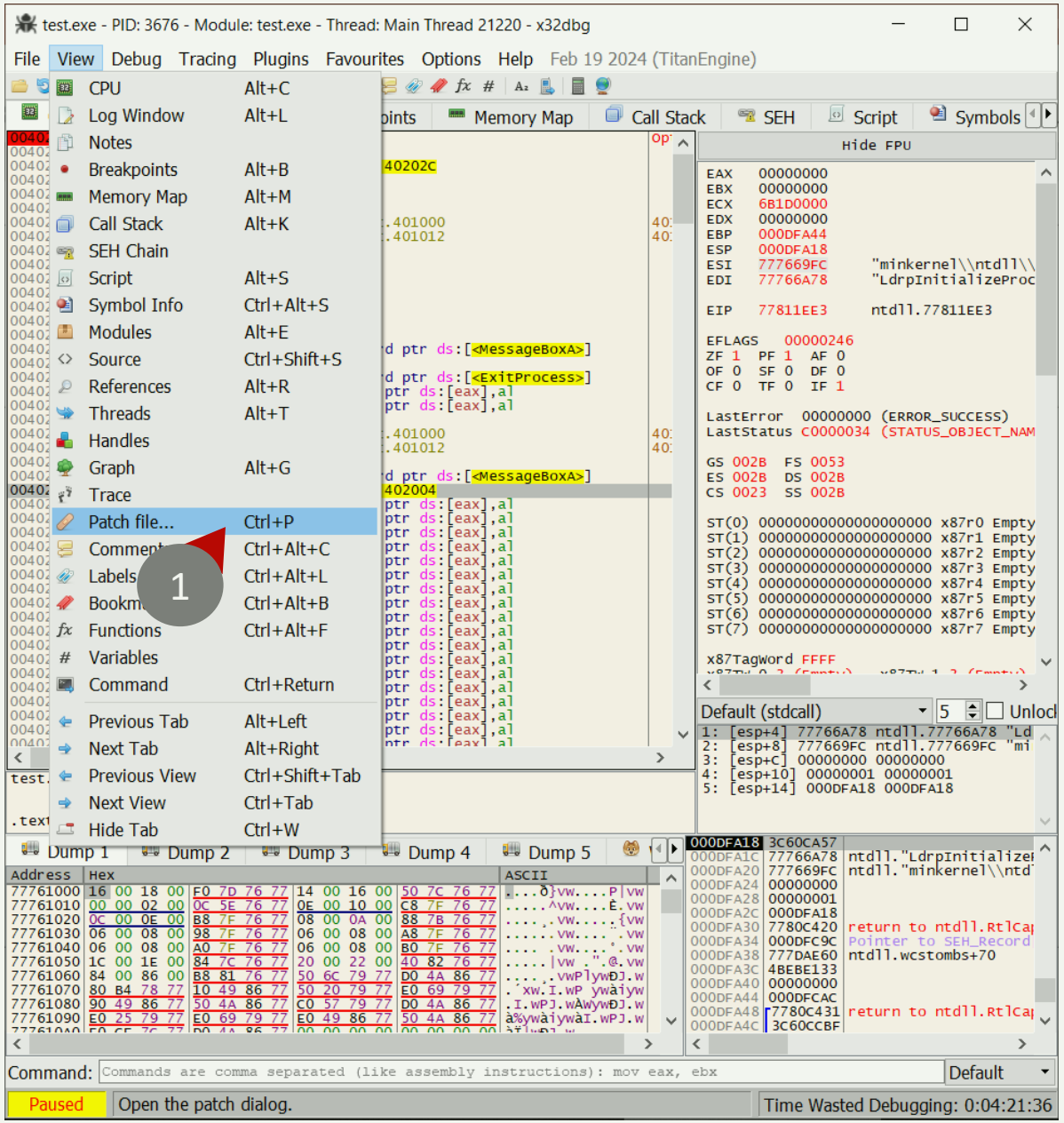
Codul redundant este înlocuit cu un salt la adresa copiată:

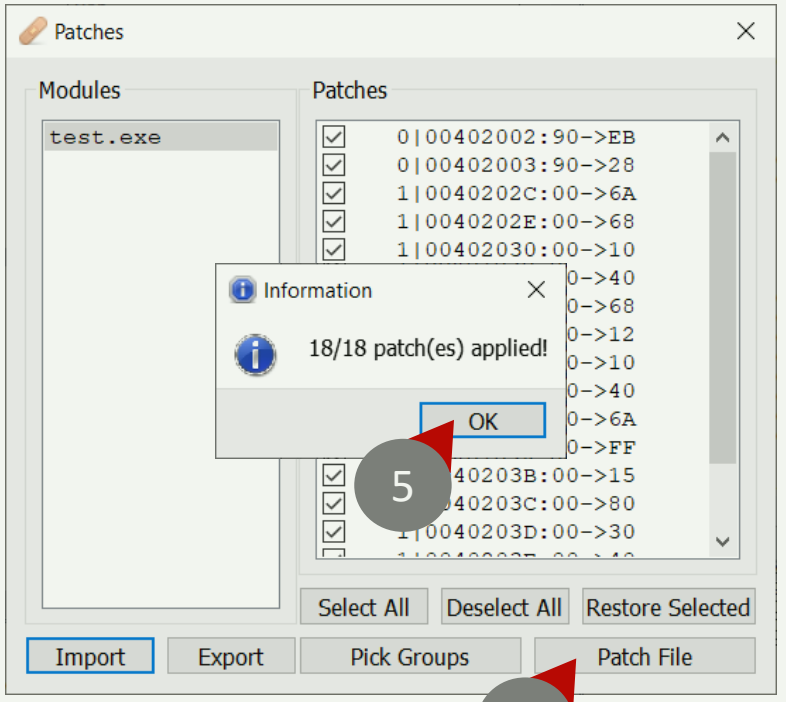


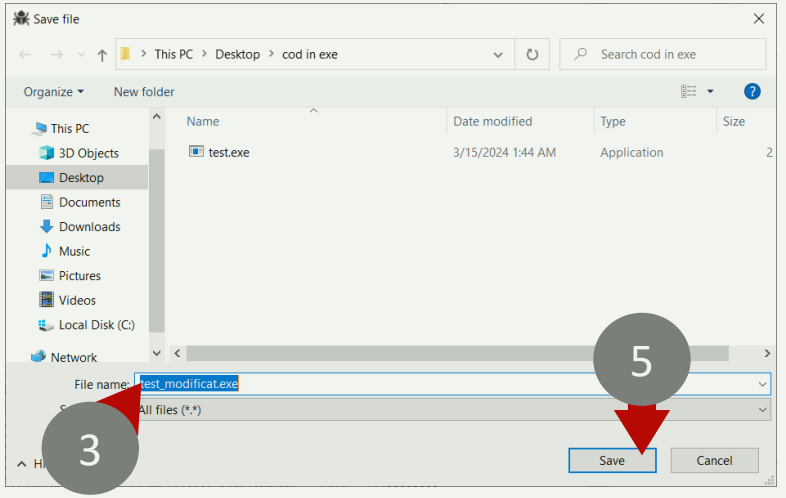




Noua versiune a executabilului este salvată:

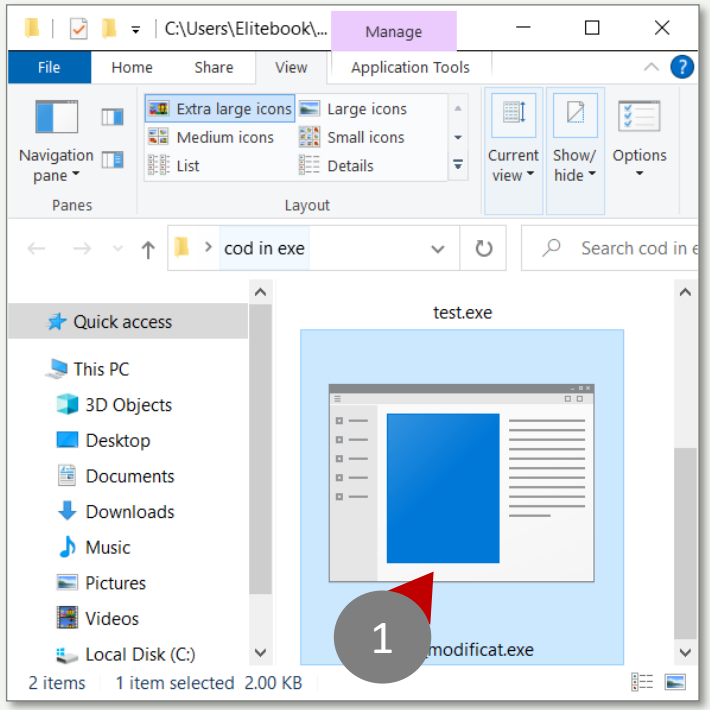






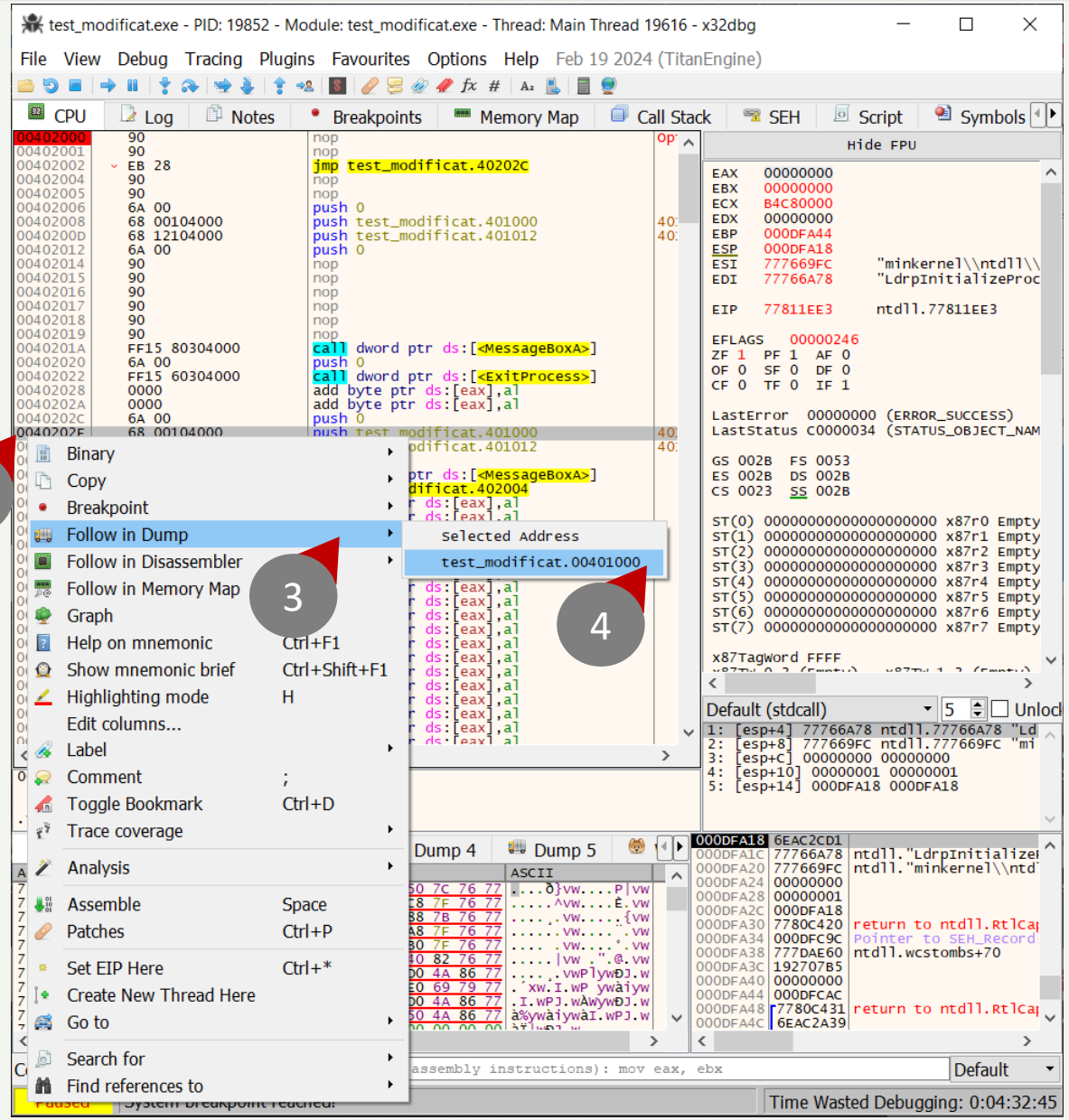
C.8.4

ADAUGAREA DE STRUCTURI DE DATE IN EXECUTABILE

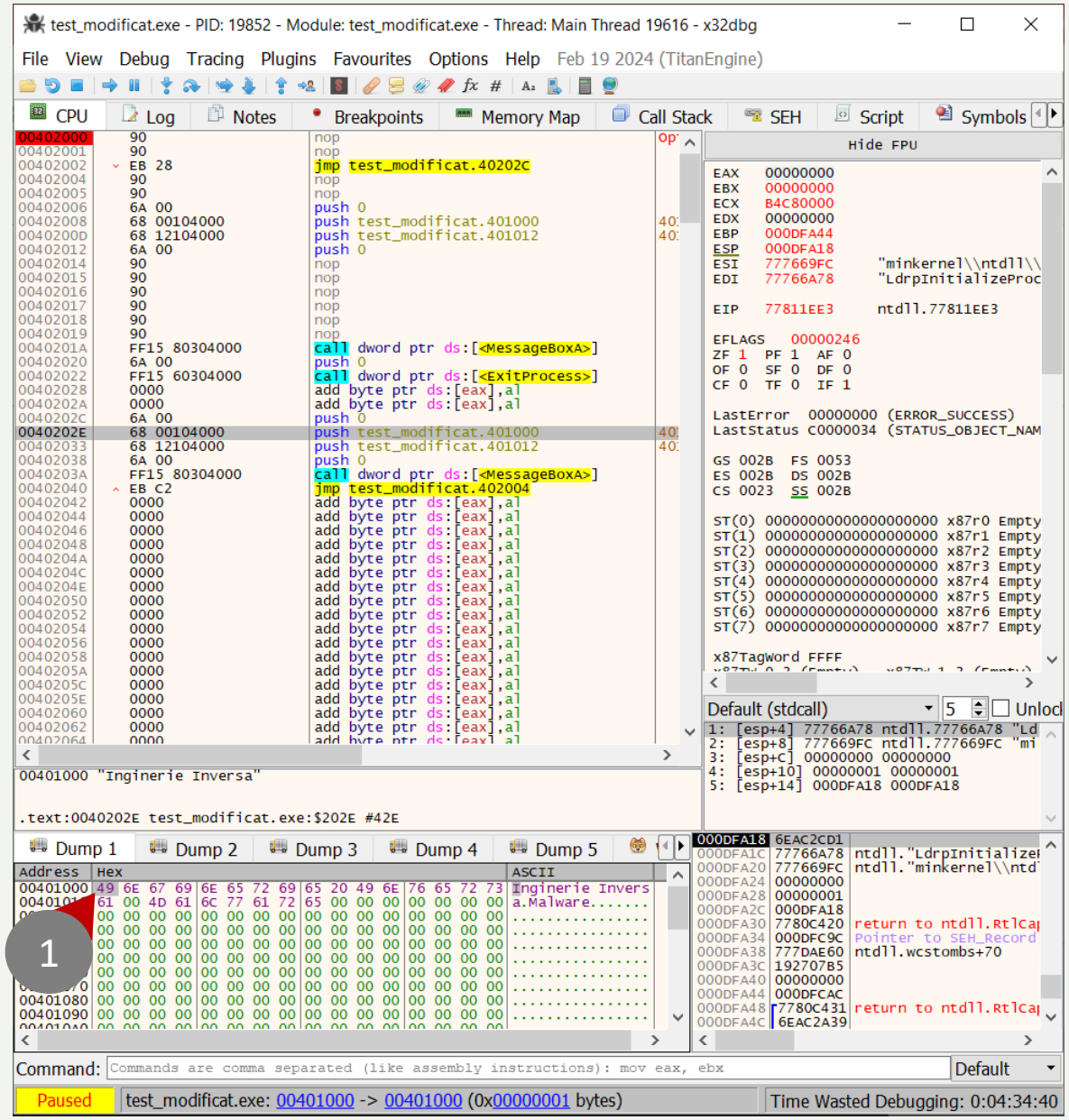


1. Schimbați punctul de intrare la un alt segment
2. Rulați segmentul în regiunea selectată
3. Salt la adresa de sub punctul de intrare

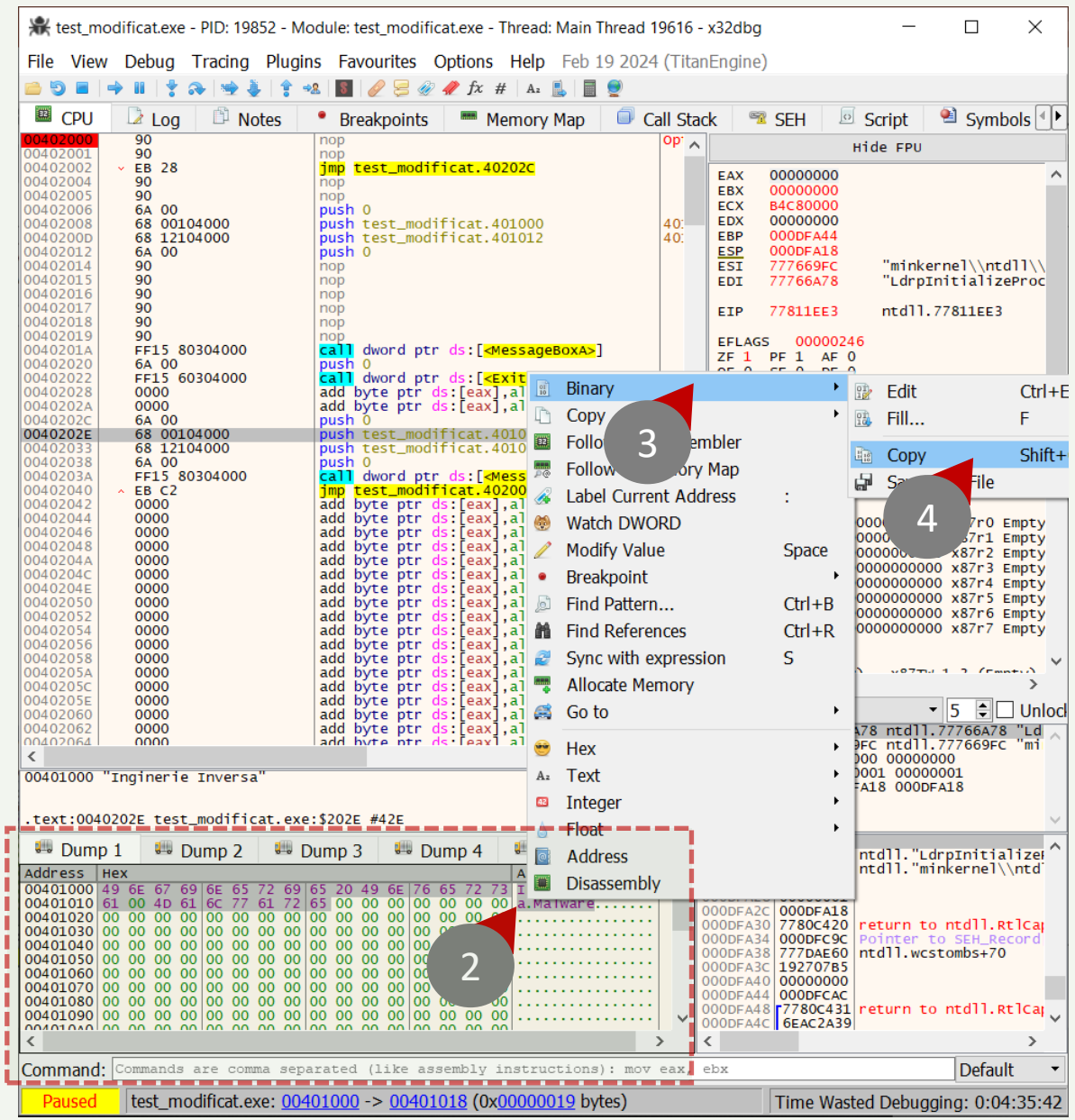
Selectați adresa și urmărim eticheta în *dump* la adresa specificată:



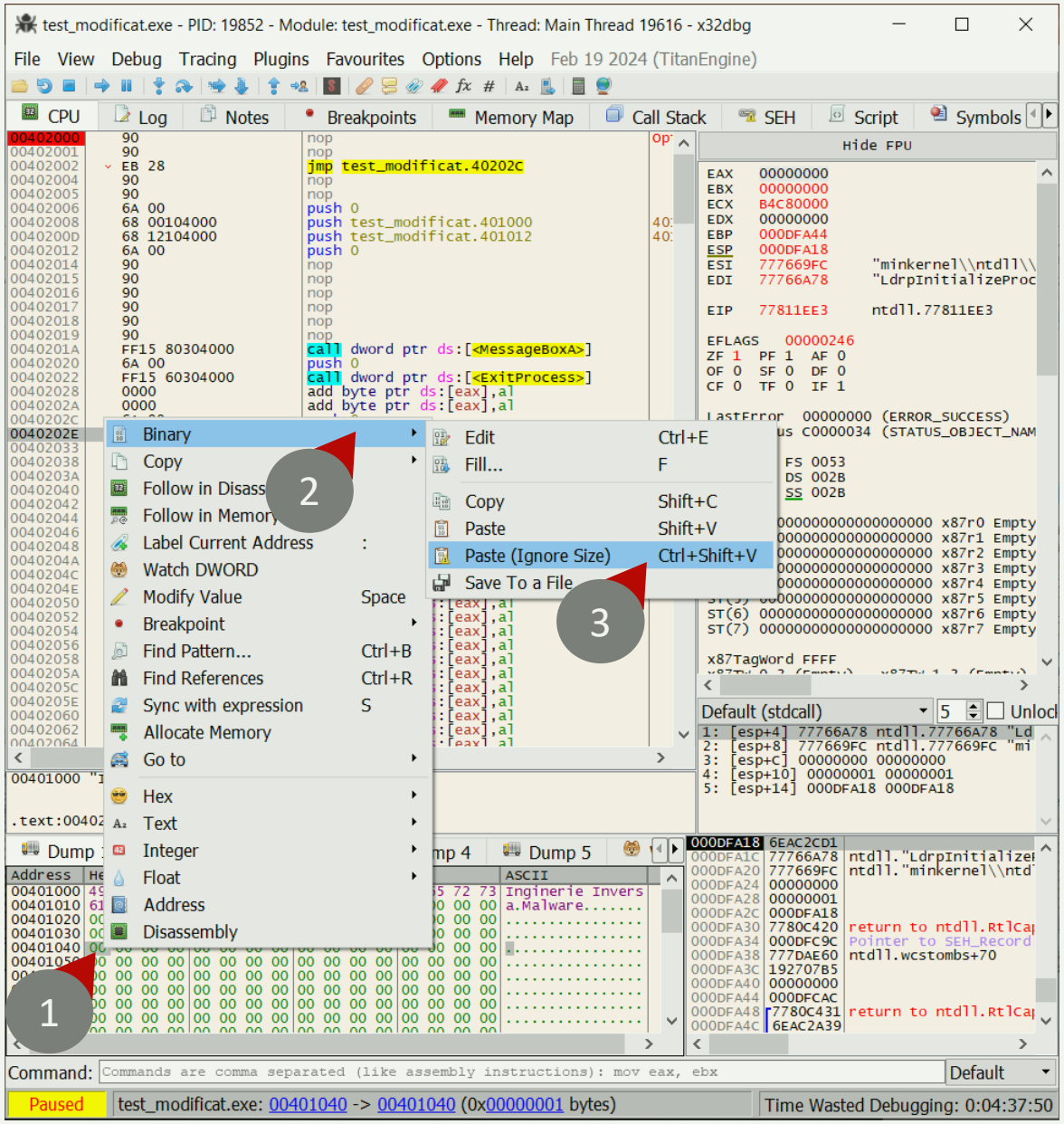
Ajungem la adresa 0040100 în *dump* și observăm șirul:



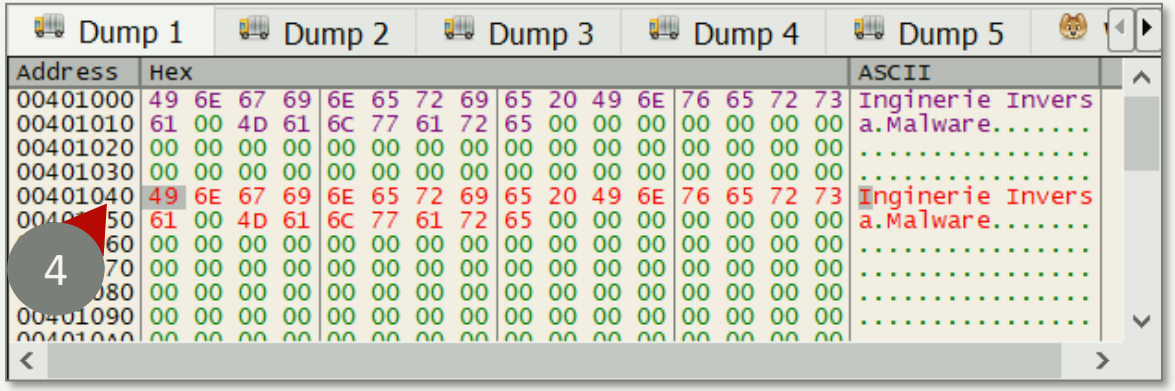
Selectați șirul și copiați datele:



Găsim zona liberă și dublăm datele folosind comanda paste:

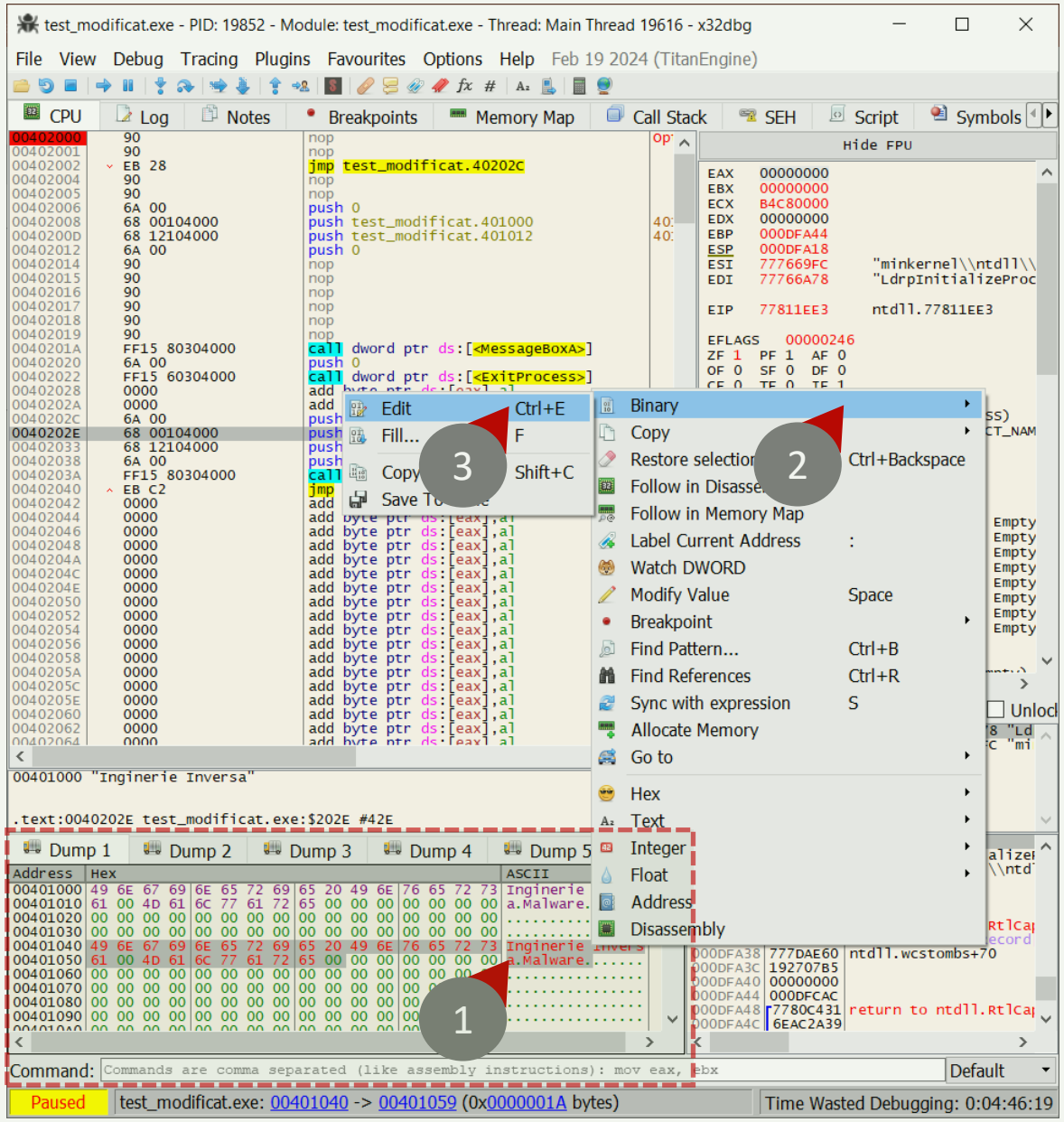


Avem date clonate la diferite adrese:

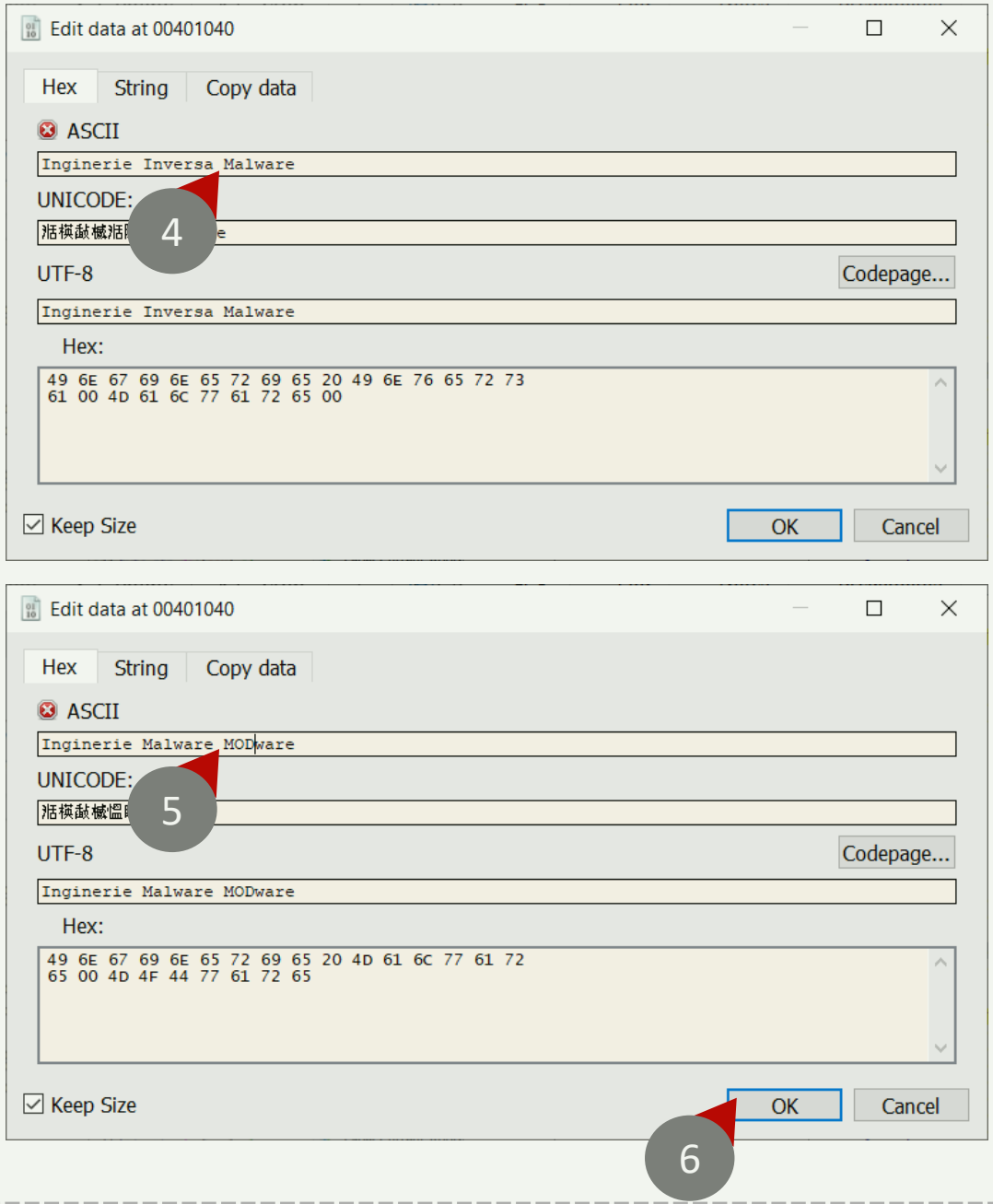


Clonăm datele pentru a diversifica argumentele pentru prima și a doua casetă de mesaj.

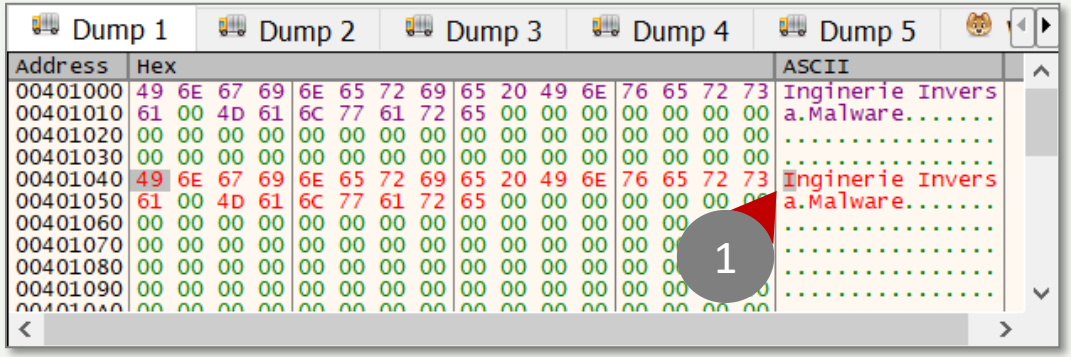
Editați șirul în *dump* la noua adresă unde se află clona:



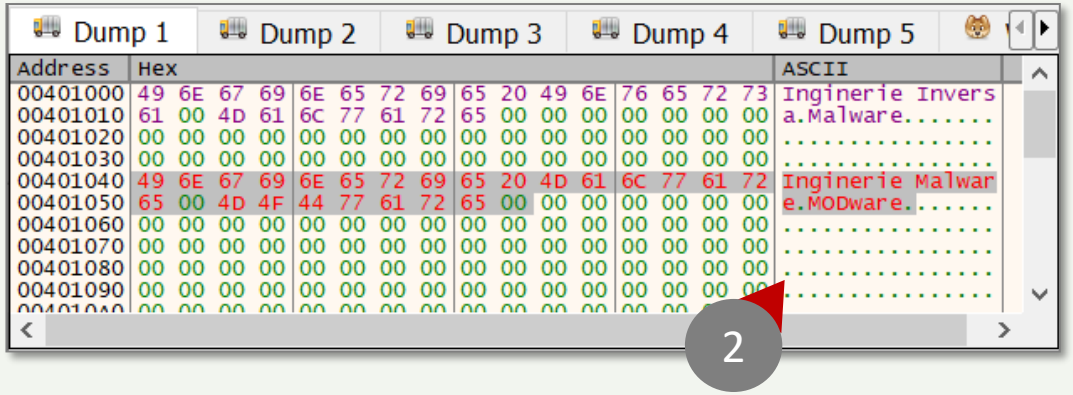
Schimbați șirul, dar mențineți dimensiunea pentru moment:



Ambele șiruri din clonă sunt editate:

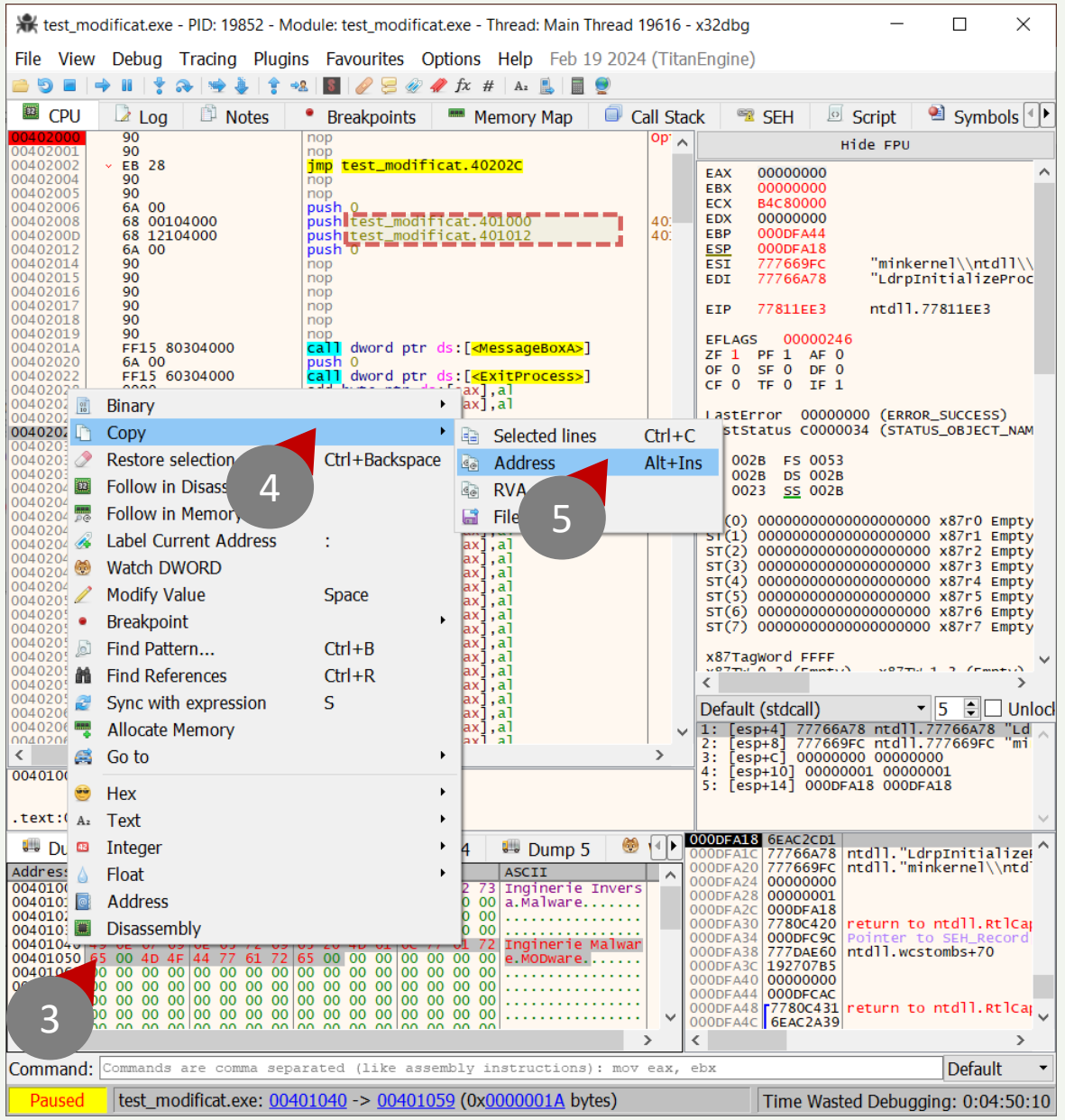


- “Inginerie Inversa” devine “Inginerie Malware”
- “Malware” devine “MODware”

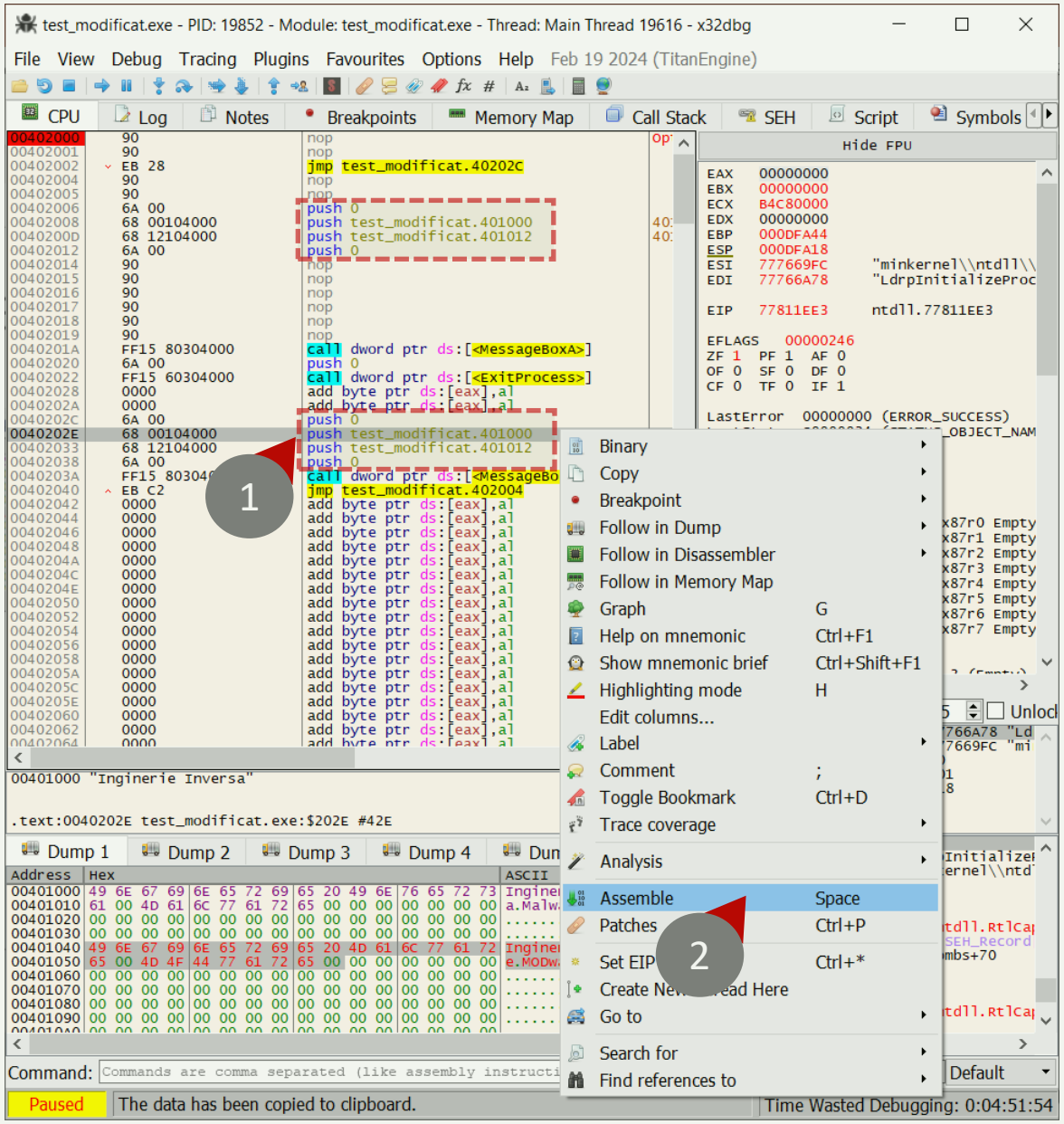


Putem scrie un șir mai mare atâta timp cât deplasăm adresa pentru cea de-a doua etichetă!

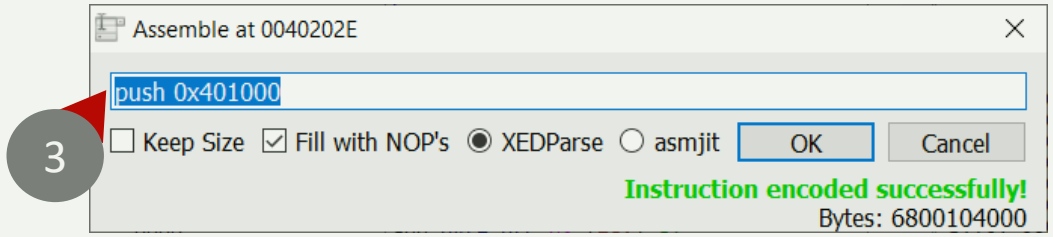
Copiați adresa la care se află șirul din clonă:

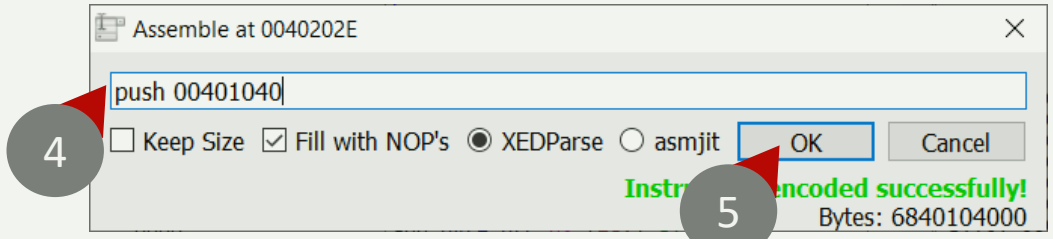


Schimbăm adresa etichetei originale cu adresa clonei modificate:

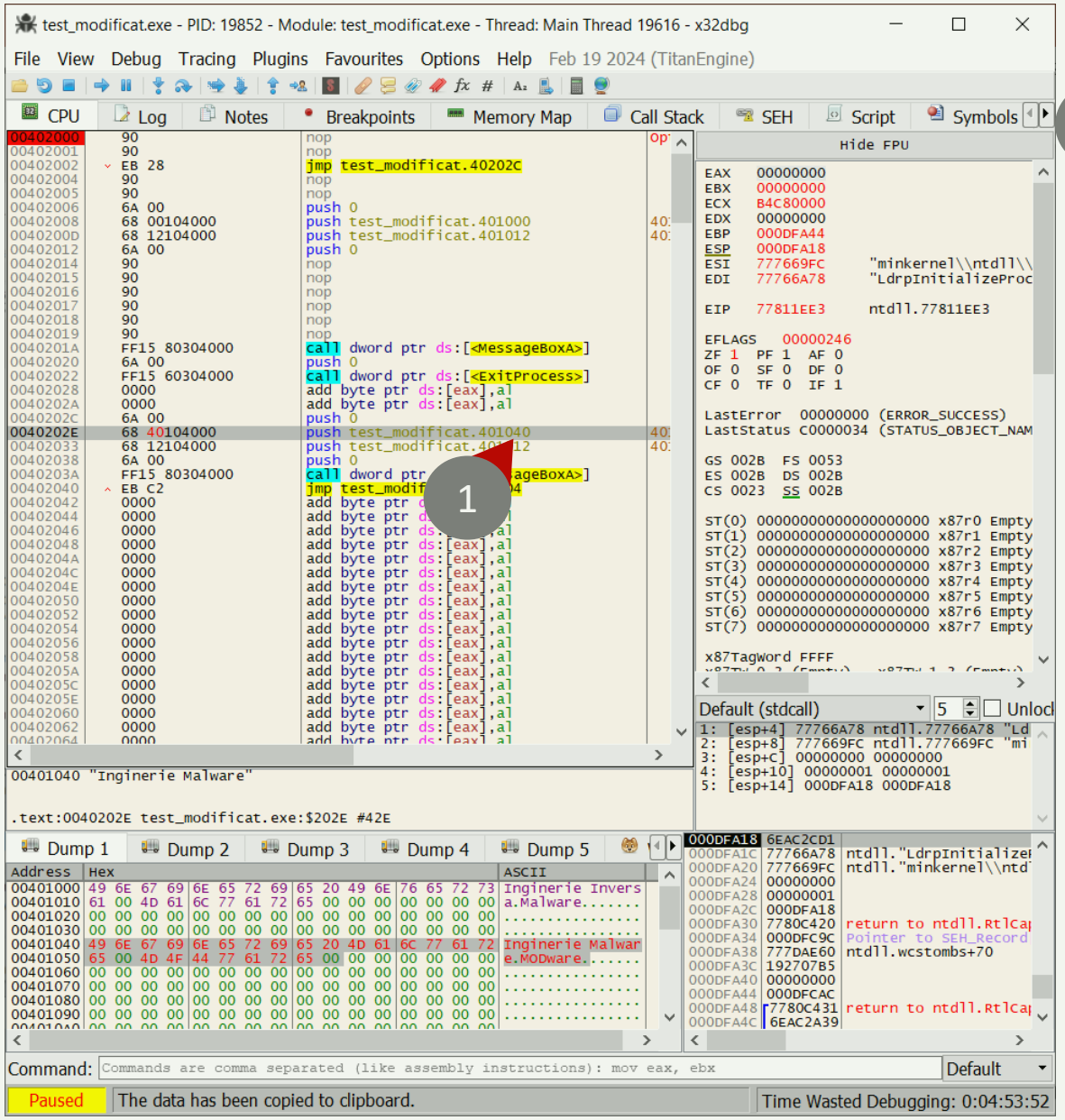


Adresa originală este înlocuită cu cea nouă a clonei:

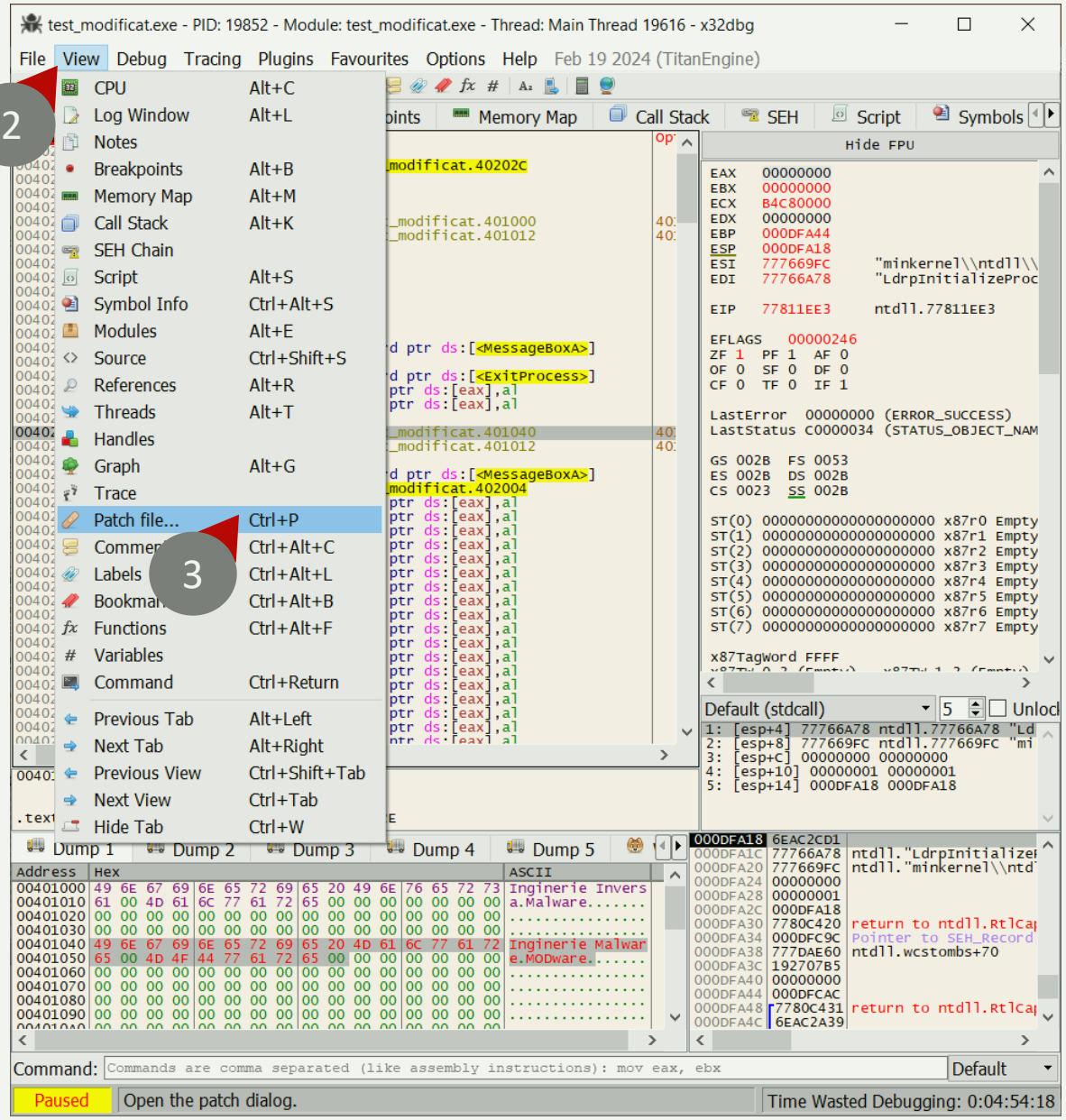




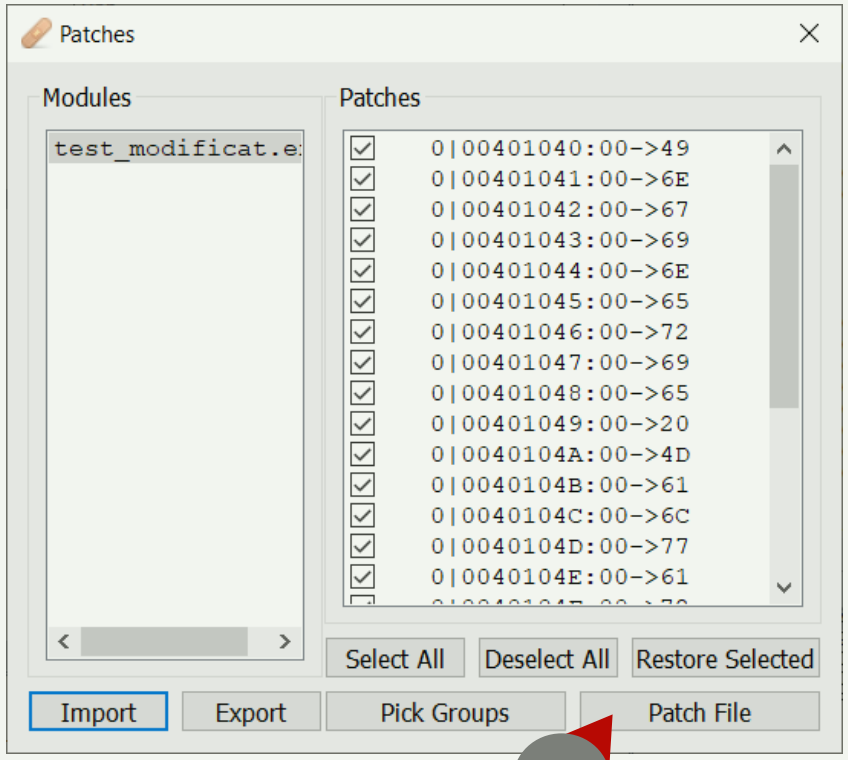
Observați că noua adresă la al doilea argument este schimbată:

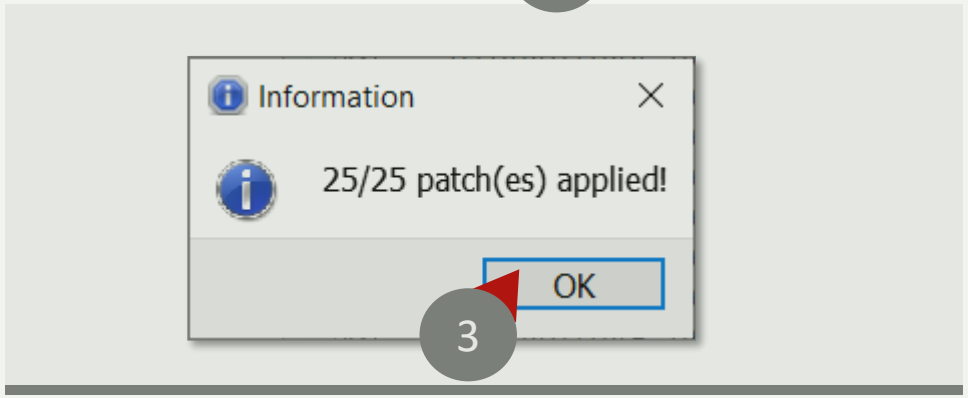


Peticim executabilul:

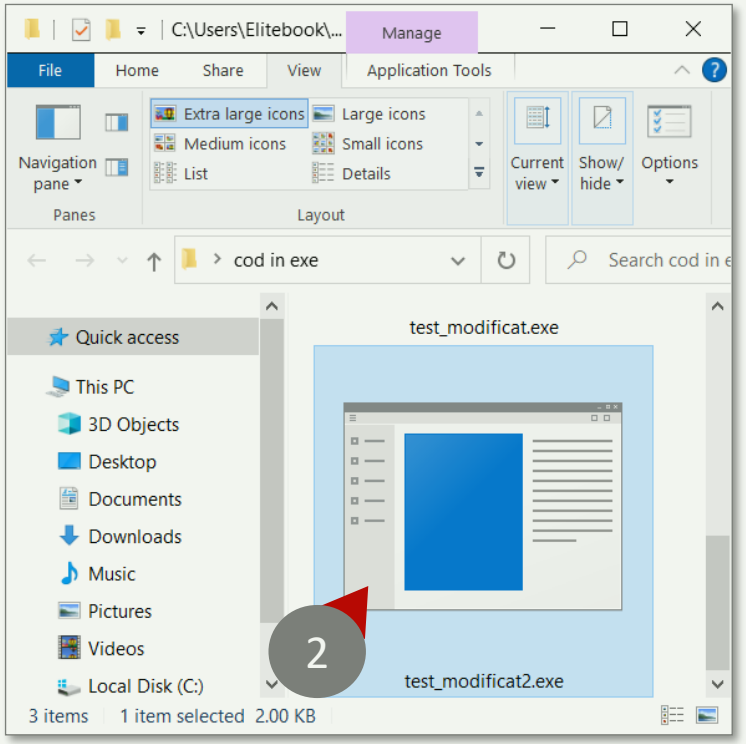


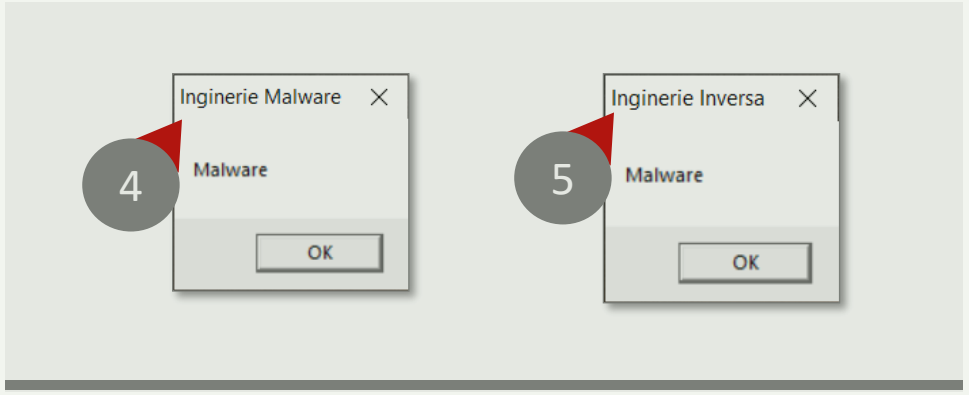
Modificările sunt prezentate orientativ:





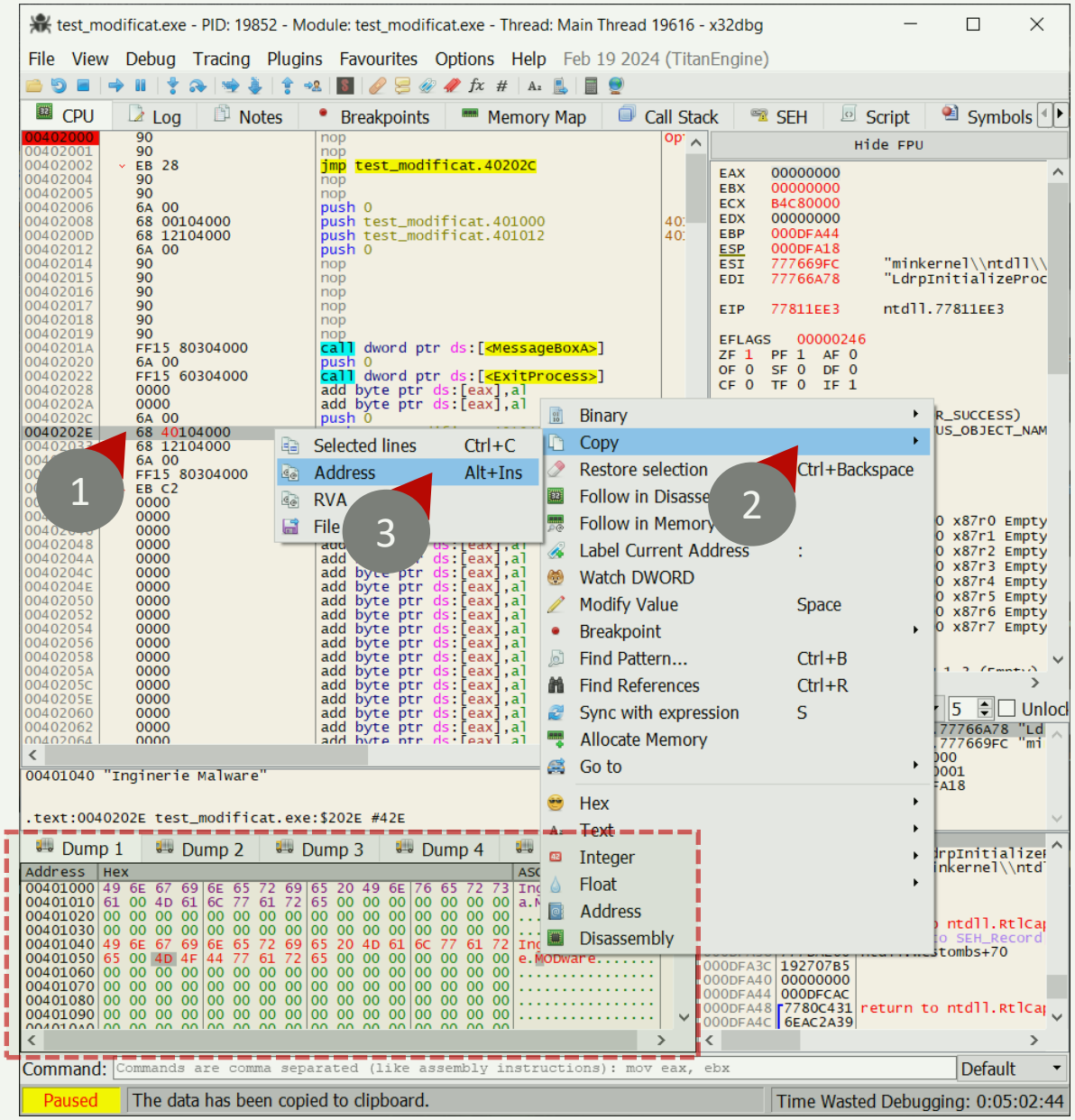
Noua versiune poate fi testată:



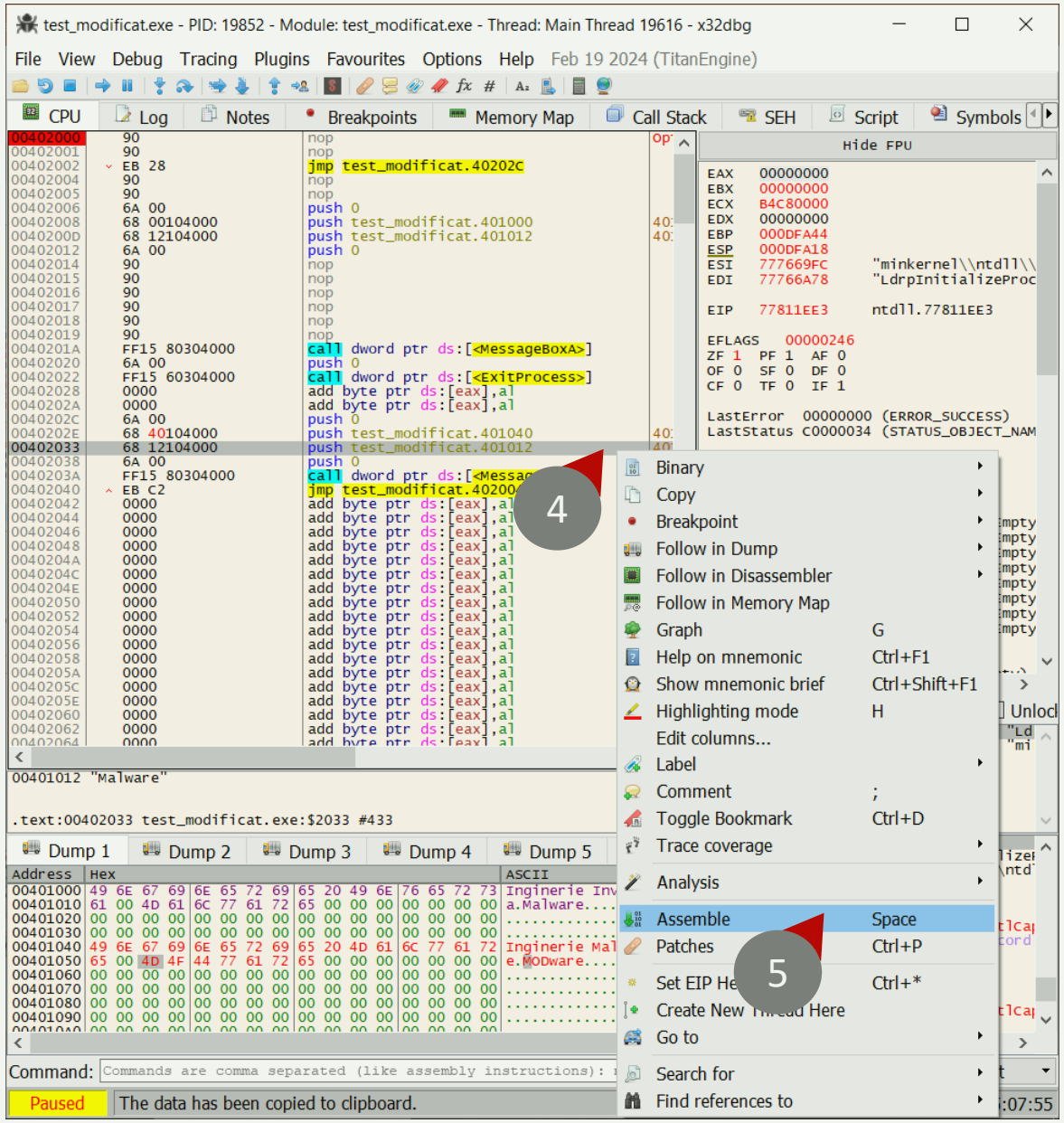


C.8.5
MODIFICĂRI DE ADRESE CĂTRE ALTE ARGUMENTE

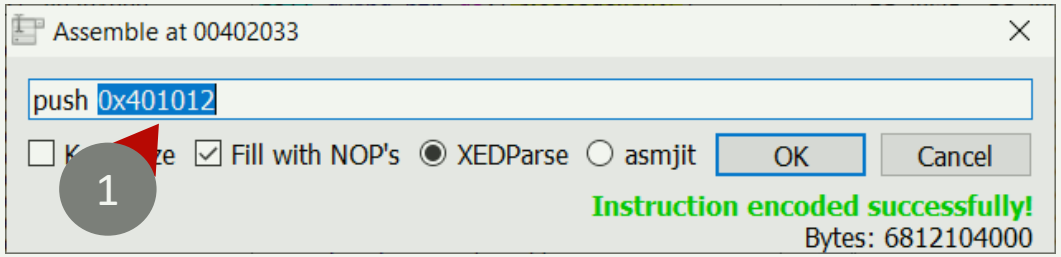
Copiați adresa pentru direcționarea către cel de-al treilea argument:



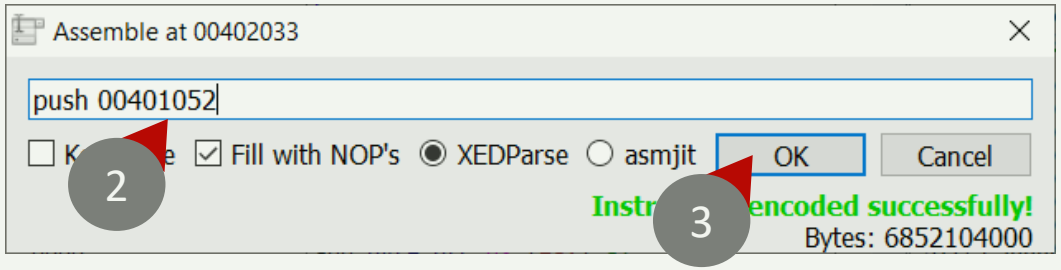
Schimbarea adresei etichetei:



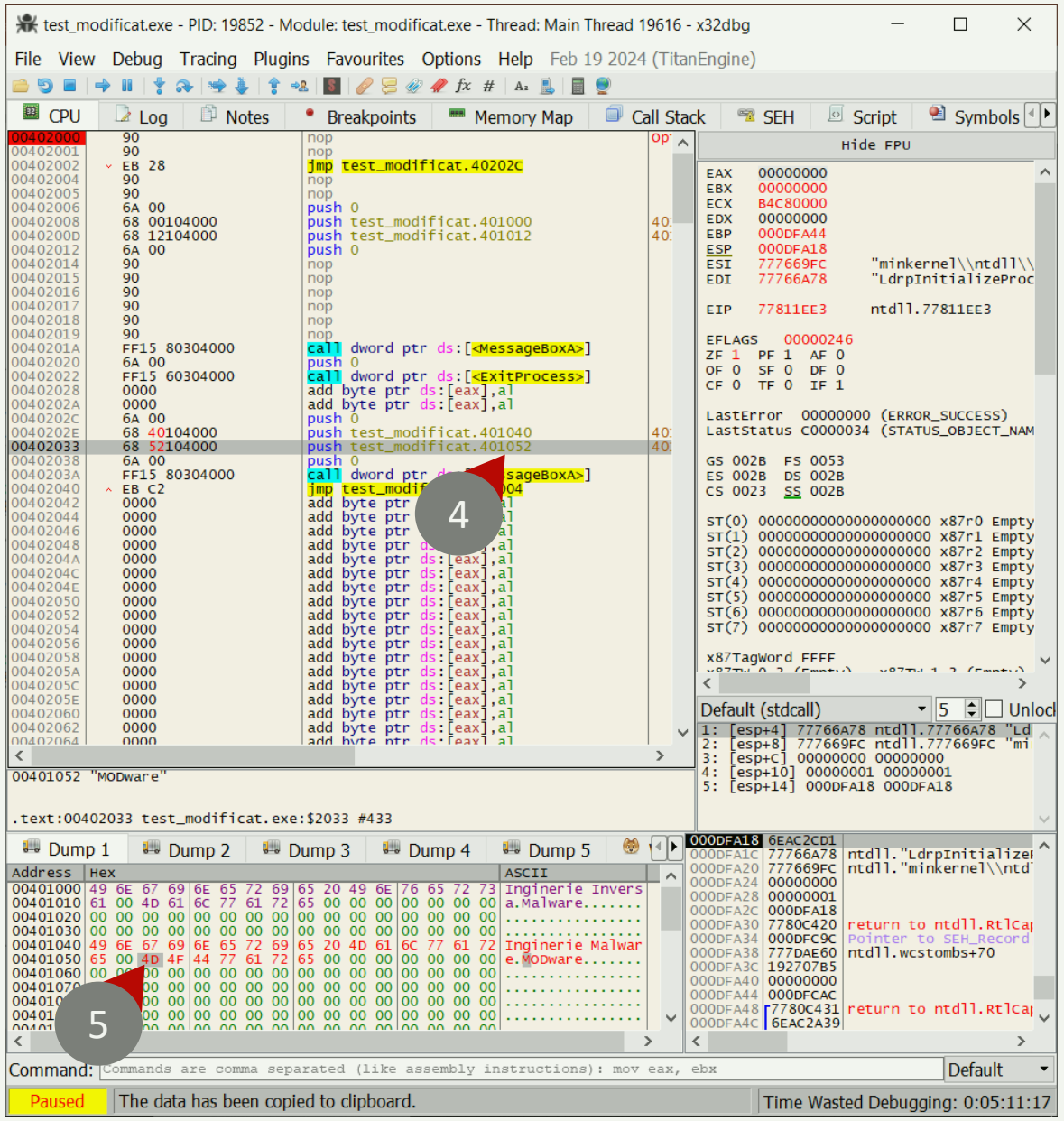
Schimbarea adresei originale:



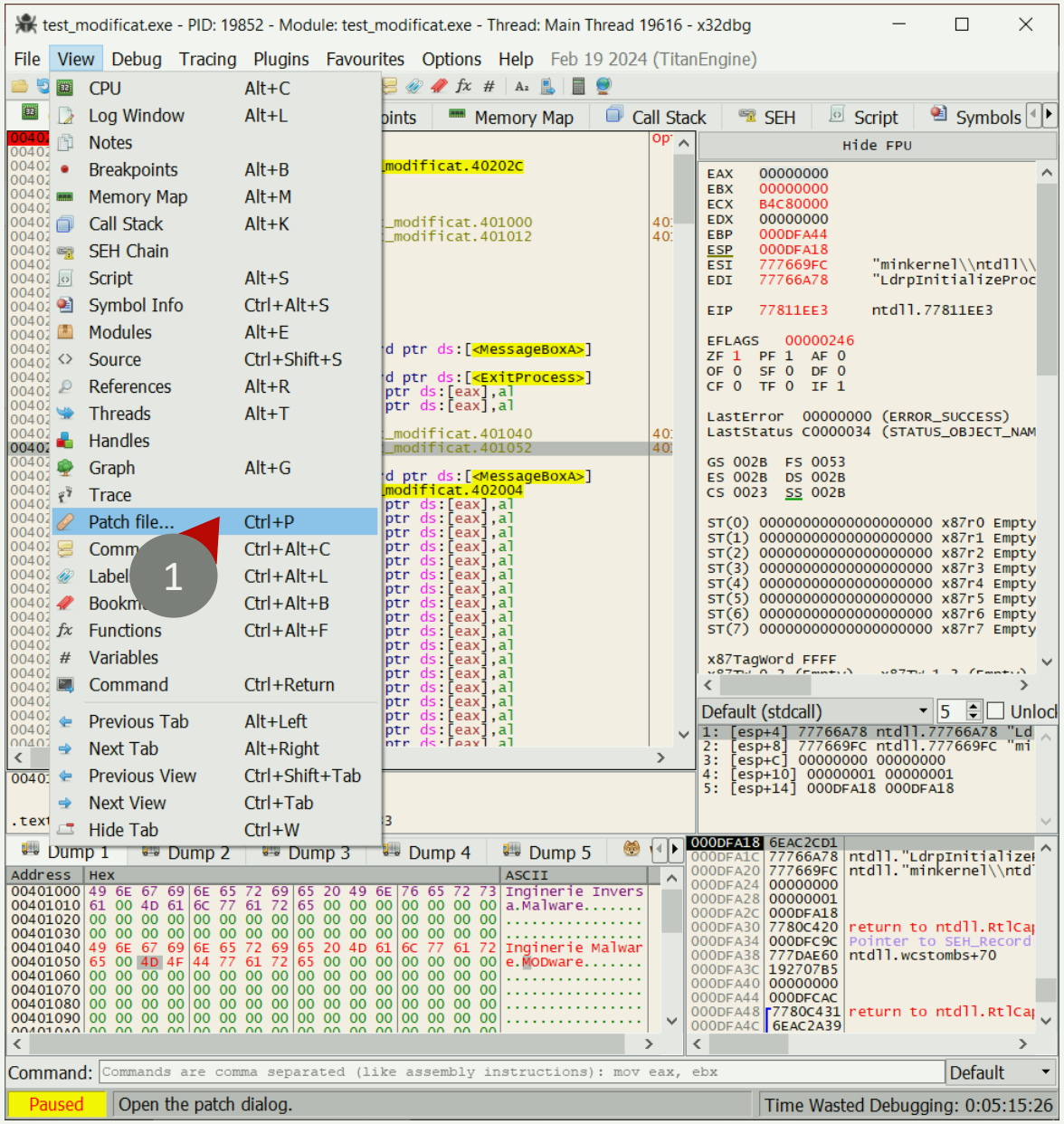
Cu adresa nou din clona:



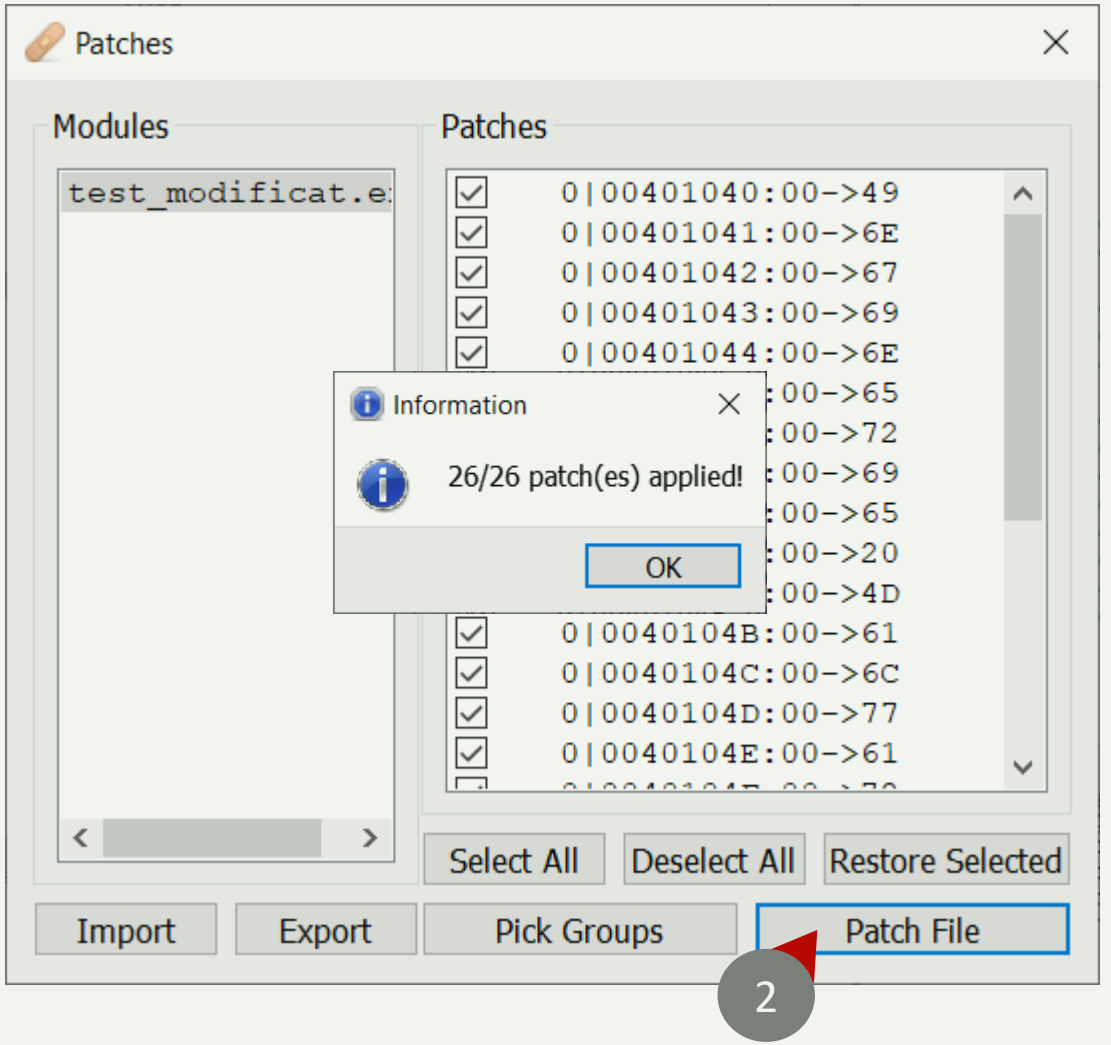
Adresa a fost schimbată:



Se trece la peticirea executabilului:



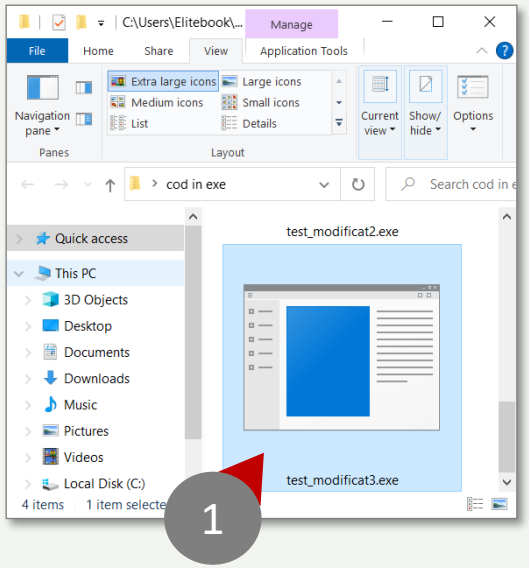
Noua versiune este scrisă pe disc și poate fi testată:

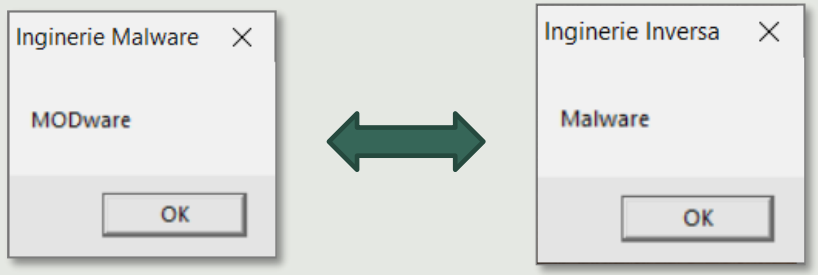


ÎN CAZUL CODULUI MALIȚIOS

CE PUTEM REALIZA CU ACESTE TIPURI DE SCHIMBĂRI?

Câteva activități de ghidare:





1. Putem vedea legături între serverele de comandă și control (C&C).

2. Putem dilua infecția mașinilor din rețelele mari creând variante ale aceluiași malware (ex. poate exista o luptă in procesul de infecție; unele aplicații malware răspund la C&C original, iar altele la un honeypot).

3. De cele mai multe ori, există un marker pentru infecție (o serie de octeți). Odată ce o mașină este infectată cu malware-ul corectat, malware-ul original nu va infecta acea mașină (este similar cu un **vaccin biologic**).

4. Abaterea de la secțiunea **.text** poate duce la confuzia unui virus dacă se așteaptă la denumirea clasica **.text**

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