

# **C.6 FORMATELE DE FIȘIERE ȘI REGIMUL DE COMPILARE**

**PAUL A. GAGNIUC**



**Academia Tehnică Militară „Ferdinand I”**

# PRINCIPALELE PĂRȚI ALE PREZENTĂRII

## C.6 Formatele de fișiere și regimul de compilare:

- C.6.1 FORMATELE DE FIȘIERE ȘI DETECTAREA PE BAZA DE SEMNATURI
- C.6.2 FIȘIERE EXECUTABILE INTERPRETATE VS COMPILATE
- C.6.3 TIPUL DE INFORMAȚII, OFUSCAREA ȘI DETECTAREA CRIPTĂRII
- C.6.4 IDENTIFICAREA PUNCTULUI DE INTRARE ÎN EXECUTABILE
- C.6.5 REGIMUL DE COMPILARE ÎN DIFERITE LIMBAJE DE PROGRAMARE
- C.6.6 ARHITECTURA CPU ȘI INSTRUCȚIUNILE DE BAZĂ

C.6.1

# FORMATELE DE FIȘIERE ȘI DETECTAREA PE BAZA DE SEMNATURI



# CELE MAI RELEVANTE FORMATE DE FIȘIERE ÎMPĂRȚITE PE CATEGORII:

Cele mai importante și comune formate pot fi grupate în mai multe categorii, în funcție de tipul de date pe care îl reprezintă.

## Cod sursă și executabile

**.exe:** Executabil pentru Windows.  
**.app:** Aplicație pentru macOS.  
**.py:** Script Python.  
**.java:** Cod sursă Java.  
**.html și .css:** Cod pentru pagini web și stilizarea acestora.

## Tablouri de date și prezentări

**.xls și .xlsx:** Formate Microsoft Excel, pentru foi de calcul.  
**.csv:** Valori separate prin virgulă, pentru a stoca date tabulare.  
**.ppt și .pptx:** Formate Microsoft PowerPoint, pentru prezentări.

## Imagini și grafice

**.jpg sau .jpeg:** Format comun pentru fotografii, folosește compresie cu pierdere.  
**.png:** Suportă transparența și este folosit pentru imagini web și grafice, compresie fără pierdere.  
**.gif:** Folosit pentru imagini animate și grafice simple, suportă transparența.  
**.svg:** Graphics Vectoriale Scalabile, pentru grafică vectorială.

## Documente

**.doc și .docx:** Formate Microsoft Word, pentru documente text.  
**.pdf:** Formatul Portable Document Format de la Adobe.  
**.txt:** Text simplu, fără formatare.  
**.rtf:** Rich Text Format, suportă formatare text, dar mai puțin complex decât .docx.

## Audio și video

**.mp3:** Standard pentru audio comprimat, folosit în mod larg pentru muzică.  
**.wav:** Audio ne-comprimat, calitate înaltă dar fișiere mari.  
**.mp4:** Format video larg utilizat, compatibil cu multe dispozitive.  
**.mov:** Format video dezvoltat de Apple, similar cu MP4.  
**.avi:** Audio Video Interleave, un format video tradițional de la Microsoft.

## Arhive

**.zip:** Format de arhivare și compresie de date.  
**.rar:** Un alt format popular de arhivare, oferă o rată de compresie mai bună decât .zip.  
**.7z:** Format de arhivare cu o rată de compresie foarte înaltă.



- Aceste formate sunt esențiale în diverse domenii, de la crearea și distribuția de conținut digital, la dezvoltarea software și gestionarea datelor. Fiecare format are avantajele și limitările sale, fiind ales în funcție de cerințele specifice ale proiectului sau activității.

# FORMATELE PRINCIPALE DE FIȘIERE EXPLOATATE DE MALWARE

Malware-ul poate exploata o varietate largă de formate de fișiere pentru a-și răspândi codul rău intenționat sau pentru a executa acțiuni dăunătoare. Unele formate de fișiere sunt mai predispuse la exploatare datorită popularității lor, complexității și funcționalităților pe care le oferă, ceea ce poate permite ascunderea și executarea codului malware.

## Documente Office

- **.doc și .docx**: Documente Microsoft Word care pot conține macro-uri rău intenționate.
- **.xls și .xlsx**: Fișiere Excel care, similare cu documentele Word, pot include macro-uri dăunătoare.
- **.ppt și .pptx**: Prezentări PowerPoint, unde macro-urile pot fi, de asemenea, o sursă de malware.



## Fișiere PDF

- **.pdf**: Pot include JavaScript sau alte forme de cod executabil care pot fi exploatate de atacatori.

## Arhive și fișiere comprimate

- **.zip, .rar, .7z**: Pot fi utilizate pentru a ascunde natura dăunătoare a unui fișier față de software-ul de securitate.

## Executabile și fișiere de sistem

- **.exe, .dll**: Fișiere executabile și biblioteci dinamice Windows care pot fi malware sau pot conține payload-uri malware.

## Scripturi și cod executabil

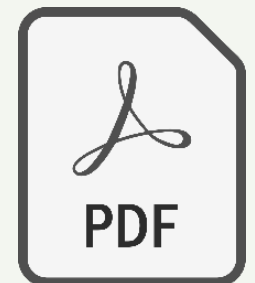
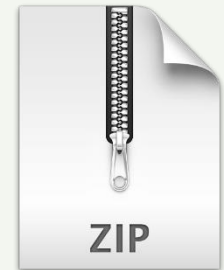
- **.js (JavaScript), .vbs (VBScript), .ps1 (PowerShell)**: Scripturi care pot fi executate pe sistemele Windows pentru a descărca sau executa cod rău intenționat.
- **.bat, .cmd**: Fișiere batch și scripturi de comandă care pot executa secvențe de comenzi dăunătoare.

## Fișiere web

- **.html, .htm, .php, .asp**: Pagini web care pot conține scripturi malițioase sau redirectiona utilizatorii către site-uri dăunătoare.
- **.swf**: Fișiere Flash care, în trecut, au fost o sursă comună de exploatare, (retragerea suportului pentru Adobe Flash Player).

## Alte formate

- **.lnk**: Scurtături care pot fi configurate să execute comenzi dăunătoare.
- **.iso, .img**: Imagini de disc care, odată montate, pot conține fișiere executabile dăunătoare.





# FIȘIERELE EXECUTABILE BINARE



(ce procent de mașini folosesc ce sistem de operare ...)

Market share: 92% 5% 3%

## Windows (92%)

**.exe:** Formatul standard pentru fișierele executabile pe Windows. Acesta include aplicații standalone și instalatori.

**.dll:** Biblioteci dinamice legate (Dynamic Link Libraries) sunt folosite pentru a stoca cod și date care pot fi utilizate de multiple programe simultan.

**.sys:** Driver de sistem specifice pentru Windows, care rulează la un nivel de acces foarte înalt în sistemul de operare.

**.scr:** Screen savers, care deși inițial erau folosiți pentru a salva ecranul de arderea pixelilor, acum pot executa orice cod, similar cu un fișier .exe.

**.com:** Un format mai vechi de executabil, originar din DOS, dar încă suportat de Windows pentru compatibilitate.

## macOS (5%)

**Mach-O:** Formatul standard pentru executabile, biblioteci dinamice, și bundle-uri în macOS și iOS.

**.app:** Un folder structurat care conține un program executabil și toate resursele necesare pentru ca aplicația să ruleze pe macOS. Deși apare ca un singur fișier în Finder, este de fapt un pachet de directoare.

**Universal Binaries:** Fișiere care conțin cod executabil pentru mai multe arhitecturi (de exemplu, Intel și ARM pentru Mac-uri cu chip M1), permițând aplicațiilor să ruleze pe diferite tipuri de hardware.

## Linux și Unix-like (3%)

**ELF (Executable and Linkable Format):** Formatul standard pentru executabile, biblioteci partajate, și dump-uri de core în sistemele bazate pe Linux și Unix.

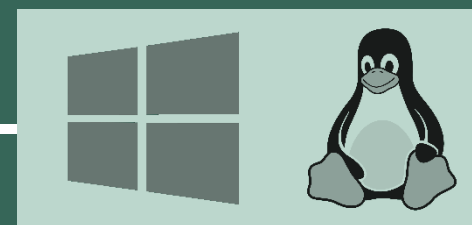
**.a:** Arhive statice, care sunt colecții de fișiere obiect compilate, utilizate pentru a crea biblioteci statice.

**.so:** Biblioteci partajate (shared objects) similare cu fișierele .dll din Windows.

**Scripturi shell:** Deși nu sunt executabile în sensul tradițional, scripturile (cum ar fi Bash scripts) pot fi executate și pot servi ca programe în sistemele Unix-like.

Aceste tipuri de fișiere executabile sunt esențiale pentru funcționarea sistemelor de operare și a aplicațiilor. Ele permit utilizatorilor să ruleze programe, driverele să comunice cu hardware-ul, și sistemele de operare să ofere funcționalități complexe prin intermediul unor interfețe software bine definite.

# FIȘIERELE EXECUTABILE SCRIPT



Termenul „executabile” poate fi interpretat într-un sens mai larg pentru a include nu doar fișierele binare native ale sistemului de operare, dar și scripturi care pot fi executate direct de către interpretorul sistemului sau alte programe de interpretare. În această gamă mai largă includem:

## Scripturi Windows (92%)

**.bat** și **.cmd**: Fișiere batch DOS/Windows, care conțin o serie de comenzi DOS care sunt executate de către interpretorul de comandă al Windows (cmd.exe).

**.ps1**: Scripturi PowerShell, executate de către motorul PowerShell, oferind acces la funcționalități avansate de automatizare și gestionare a sistemului.

**.vbs**: Scripturi VBScript, care pot fi executate în Windows prin Windows Script Host sau în alte medii care suportă VBScript, precum unele aplicații web.

## Linux și Unix-like (3%)

**Scripturi Shell**: Acestea includ scripturi Bash (.sh), Korn shell (.ksh), C shell (.csh), și alte shell-uri disponibile pe sisteme Unix și Linux. Sunt executate de interpretorul shell corespunzător.

**.py**: Deși Python este multi-platformă, scripturile Python (.py) pot fi executate ca scripturi pe sisteme Unix, Linux, și macOS, precum și pe Windows, având nevoie de interpretorul Python instalat.

**.pl**: Scripturi Perl, care, similar cu Python, necesită interpretorul Perl pentru a fi executate, disponibil pe multe platforme.

## Altele

**.rb**: Scripturi Ruby, care pot fi executate pe orice sistem cu interpretorul Ruby instalat.

**.php**: Deși frecvent asociat cu dezvoltarea web și executat pe un server web, scripturile PHP pot fi, de asemenea, executate direct de la linia de comandă pentru diverse sarcini de scriptare.

Aceste tipuri de „executabile” reprezintă un mod flexibil și puternic de a automatiza sarcini, procesa date, gestiona sisteme și dezvolta aplicații. Deși nu sunt compilabile în cod mașină specific unei arhitecturi hardware, cum sunt fișierele binare, scripturile pot fi totuși executate direct de un sistem prin intermediul interpretorilor corespunzători, făcându-le deosebit de utile pentru o gamă largă de aplicații și medii de dezvoltare.

# ALTE TIPURI DE EXECUTABILE BINARE ȘI SCRIPT

Distincția principală între tipurile de executabile se face între executabile binare și executabile sub formă de scripturi, dar aceste categorii pot fi extinse sau detaliate în mai multe moduri, în funcție de context și de modul în care sunt utilizate sau executate.

## Bytecode Executables

- Aceste fișiere sunt o formă intermediară între codul sursă și codul mașină. Bytecode-ul este adesea asociat cu limbaje de programare cum ar fi Java (.class sau .jar) și .NET (asemblări .dll sau .exe), și necesită o mașină virtuală (cum ar fi Java Virtual Machine sau .NET Runtime) pentru a-l interpreta și executa. Aceste fișiere nu sunt nici pur binare (deși sunt stocate într-un format binar), nici scripturi text, ci reprezintă un set de instrucțiuni standardizate care sunt executate într-un mediu controlat.

## Web Executables

- Pe lângă scripturile PHP menționate anterior, alte tehnologii web, cum ar fi JavaScript (JS) pentru clienți web sau Node.js pentru server, permit executarea codului în browsere sau pe servere. Deși acestea sunt de obicei scripturi text, mediul lor de execuție le conferă o portabilitate și o capacitate de execuție comparabilă cu executabilele.

## Containerized Executables

- În era cloud și a microserviciilor, conceptul de containere, cum ar fi Docker, a devenit esențial. Acestea permit pachetarea aplicațiilor și a dependențelor lor în unități standardizate pentru dezvoltare software. Deși nu sunt "executabile" în sensul tradițional, containerele pot fi executate pe orice sistem care suportă tehnologia de containerizare, oferind o formă de execuție portabilă și izolată.

## Pseudo-Executables

- Unele formate de fișiere, cum ar fi documentele Office cu macro-uri sau fișierele PDF, pot conține sau face referire la cod executabil (de exemplu, macro-uri VBA sau JavaScript în PDF-uri). Aceste fișiere, în timp ce în esență sunt documente, pot deveni vehicule pentru execuția de cod malițios și sunt adesea exploatate în scopuri de securitate.

## SaaS Executables

- În contextul Software-as-a-Service (SaaS) și al aplicațiilor web, conceptul de „executabil” se extinde la aplicații accesate și executate printr-un browser web, fără a fi necesară instalarea locală. Deși aceasta este o interpretare mai largă și diferită, reflectă evoluția modului în care gândim despre software și execuția sa în era digitală.



# SEMNATURILE DE FIȘIERE CUNOSCUTE ȘI CA „MAGIC NUMBERS”

## Imagini

- JPEG:** **FF D8 FF** - Extensie: .jpg sau .jpeg
- PNG:** **89 50 4E 47 0D 0A 1A 0A** - Extensie: .png
- GIF:** **47 49 46 38** - Extensie: .gif

## Documente

- PDF:** **25 50 44 46** - (începe cu %PDF) - Extensie: .pdf
- Microsoft Word/Excel/PowerPoint (2007+):** **50 4B 03 04** - Extensii: .docx, .xlsx, .pptx (Acesta este, de fapt, formatul unui fișier ZIP, deoarece documentele Office moderne sunt pachete ZIP care conțin XML și alte tipuri de fișiere.)

## Executabile și Sisteme de Fișiere

- Executabile Windows (PE):** **4D 5A** - Extensii: .exe, .dll, .sys
- Linux și alte Unix executables (ELF):** **7F 45 4C 46** - Extensii: .elf, fișiere fără extensie pentru executabile
- Fișiere script Shell:** Nu au o semnătură hex specifică, identificarea se face prin shebang (de ex., #!/bin/bash) la începutul fișierului: **23 21 2F 62 69 6E 2F 62 61 73 68**.

## Arhive

- ZIP:** **50 4B 03 04** - Extensii: .zip, .jar, .docx, .xlsx, .pptx, etc.
- RAR:** **52 61 72 21** - Extensie: .rar
- 7z:** **37 7A BC AF 27 1C** - Extensie: .7z

## Media

- MP3:** Variază, dar fișierele MP3 cu tag-uri ID3 încep cu **49 44 33** - Extensie: .mp3
- Wave:** **52 49 46 46** urmat de dimensiune și **57 41 56 45** - Extensie: .wav
- MPEG** - **00 00 01 BA** sau **00 00 01 B3** pentru fișiere video.
- MPEG-4 video file:** **66 74 79 70 69 73 6F 6D** (mai în interiorul fișierului) - Extensii: .mp4, .m4v
- Windows shortcut** files: **4C 00 00 00 01 14 02 00** - Extensie: .lnk

Semnăturile de fișiere, cunoscute și ca „magic numbers”, sunt secvențe de bytes la începutul fișierelor care ajută la identificarea formatului acestora chiar și în absența extensiei.



Rețineți că aceste semnături sunt doar primele câțiva bytes din fișier și că, pentru unele tipuri de fișiere (precum MP3 sau executabile), modul în care sunt interpretate poate varia. De asemenea, în cazul documentelor Office sau a arhivelor ZIP, același antet poate indica mai multe tipuri de fișiere, deci contextul și conținutul complet al fișierului sunt necesare pentru o identificare precisă.

Aceste semnături sunt utile pentru identificarea rapidă a tipului de fișier, dar nu sunt singura metodă de determinare a tipului de fișier și pot fi, de asemenea, ușor falsificate în anumite cazuri de malware sau fișiere corupte.

# EXAMPLE IN CUTTER

## MAGIC HEADERS (OBSERVAȚI CĂ .PPTX ȘI .ZIP SUNT IDENTICE)

.pdf  
25 50 44 46

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000000000	25	50	44	46	2d	31	2e	36	0d	25	e2	e3	cf	d3	0d	0a	%PDF-1.6.%.....
0x0000000000000010	33	35	20	30	20	6f	62	6a	0d	3c	3c	2f	4c	69	6e	65	35 0 obj.<</Line
0x0000000000000020	61	72	69	7a	65	64	20	31	2f	4c	20	36	30	31	30	36	arized 1/L 60106
0x0000000000000030	34	2f	4f	20	33	39	2f	45	20	31	32	37	34	31	36	2f	4/O 39/E 127416/
0x0000000000000040	4e	20	38	2f	54	20	36	30	30	33	30	32	2f	48	20	5b	N 8/T 600302/H [
0x0000000000000050	20	31	31	39	38	20	34	33	37	5d	3e	3e	0d	65	6e	64	1198 437]>>.end
0x0000000000000060	6f	62	6a	0d	20	20	20	20	20	20	20	20	20	20	20	20	obj.
0x0000000000000070	20	20	0d	0a	78	72	65	66	0d	0a	33	35	20	34	34	0d	..xref..35 44.
0x0000000000000080	0a	30	30	30	30	30	30	30	30	31	36	20	30	30	30	30	..000000016 0000
0x0000000000000090	30	20	6e	0d	0a	30	30	30	30	30	30	31	36	33	35	20	0 n..000001635
0x00000000000000a0	30	30	30	30	30	2e	0d	0a	30	30	30	30	30	30	30	31	00000 n..0000001
0x00000000000000b0	37	37	34	20	30	30	30	30	20	6e	0d	0a	30	30	30	30	774 00000 n..000
0x00000000000000c0	30	30	30	32	30	34	36	20	30	30	30	30	20	6e	0d	0a	0002046 00000 n..

.exe  
4D 5A

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000000000	4d	5a	90	00	03	00	00	04	00	00	00	ff	ff	00	00	00	MZ.....@.....
0x0000000000000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.....@.....
0x0000000000000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....@.....
0x0000000000000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....@.....
0x0000000000000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	.....!..L..!Th
0x0000000000000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program canno
0x0000000000000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
0x0000000000000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode.....\$.....
0x0000000000000080	8f	8a	f9	db	cb	eb	97	88	cb	eb	97	88	cb	eb	97	88	.....
0x0000000000000090	48	f7	99	88	ca	eb	97	88	a2	f4	9e	88	ca	eb	97	88	H.....
0x00000000000000a0	22	f4	9a	88	ca	eb	97	88	52	69	63	68	cb	eb	97	88	".....Rich....
0x00000000000000b0	00	00	00	00	00	00	00	50	45	00	00	4c	01	03	00	00	.....PE..L....
0x00000000000000c0	f6	3f	f6	65	00	00	00	00	00	00	00	00	00	0f	01	00	?..e.....

.jpg  
FF D8 FF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000000000	ff	d8	ff	e0	00	10	4a	46	49	46	00	01	01	00	00	01	.....JFIF.....
0x0000000000000010	00	01	00	00	ff	e2	01	d8	49	43	43	5f	50	52	4f	46	.....ICC_PROF
0x0000000000000020	49	4c	45	00	01	01	00	01	c8	6c	63	6d	73	02	10	00	ILE.....lcms..
0x0000000000000030	00	00	6d	6e	74	72	52	47	42	20	58	59	5a	20	07	e2	..mnrRGB XYZ
0x0000000000000040	00	03	00	14	00	09	00	0e	00	1d	61	63	73	70	4d	53	.....acspMS
0x0000000000000050	46	54	00	00	00	00	73	61	77	73	63	74	72	6c	00	00	FT.....sawscrtl..
0x0000000000000060	00	00	00	00	00	00	00	00	00	00	00	00	f6	d6	00	01	.....hand...=.
0x0000000000000070	00	00	00	00	d3	2d	68	61	6e	64	9d	91	00	3d	40	80	.....=et,..."
0x0000000000000080	b0	3d	40	74	2c	81	9e	a5	22	8e	00	00	00	00	00	00	.....desc.
0x0000000000000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....cprt.
0x00000000000000a0	00	00	00	00	00	00	00	00	09	64	65	73	63	00	00	00	..wtpt.....rX
0x00000000000000b0	00	f0	00	00	00	5f	63	70	72	74	00	00	01	0c	00	00	
0x00000000000000c0	00	0c	77	74	70	74	00	00	01	18	00	00	00	14	72	58	

.pptx  
50 4B 03 04

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000000000	50	4b	03	04	14	00	06	00	08	00	00	00	21	00	cd	3d	PK.....!..=
0x0000000000000010	ca	4b	00	03	00	00	fd	2f	00	00	13	00	08	02	5b	43	.K...../[C
0x0000000000000020	6f	6e	74	65	6e	74	5f	00	00	00	70	65	73	5d	2e	78	ontent_Types].xm
0x0000000000000030	6c	20	a2	04	02	28	a0	00	02	00	00	00	00	00	00	00	l.....(.....
0x0000000000000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x00000000000000a0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x00000000000000b0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x00000000000000c0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

.zip  
50 4B 03 04

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000000000	50	4b	03	04	0a	00	00	00	00	00	da	40	6c	54	00	00	PK.....@!T..
0x0000000000000010	00	00	00	00	00	00	00	00	00	00	1a	00	00	00	42	69	.....@!T..
0x0000000000000020	6e	61	72	79	20	6d	65	74	6f	73	6f	72	70	68	6f	73	Binary metamorphos
0x0000000000000030	69	73	20	56	32	2e	30	2f	50	4b	03	04	0a	00	00	00	is V2.0/PK.....
0x0000000000000040	00	00	da	40	6c	54	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000000050	00	00	1e	00	00	00	42	69	6e	61	72	79	20	6d	65	74	.....
0x0000000000000060	61	6d	6f	72	70	68	6f	73	69	73	20	56	32	2e	30	2f	.....
0x0000000000000070	62	69	6e	2f	50	4b	03	04	14	00	00	00	08	00	d4	40	amorphosis V2.0/
0x0000000000000080	6c	54	71	47	1a	03	c9	33	00	00	00	b0	00	00	2b	00	bin/PK.....@
0x0000000000000090	00	00	42	69	6e	61	72	79	20	6d	65	74	61	6d	6f	72	lTqG...3.....+
0x00000000000000a0	70	68	6f	73	69	73	20	56	32	2e	30	2f	62	69	6e	2f	..Binary metamor
0x00000000000000b0	42	69	6e	5f	54	6f	5f	56	42	2e	65	78	65	ec	5b	7d	phosis V2.0/bin/
0x00000000000000c0	74	5c	c5	75	bf	ab	5d	c9	fa	da	0f	c9	a6	18	23	db	Bin_To_VB.exe.[

.lnk  
4C 00 00 00 01

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000000000	4c	00	00	00	01	14	02	00	00	00	00	c0	00	00	00	00	.....F.....F.
0x0000000000000010	00	00	00	00	4f	40	00	00	20	00	00	da	ac	46	ec	00	.....F.....F.
0x0000000000000020	4a	6d	da	01	32	07	80	00	00	00	da	01	4c	29	3d	8c	Jm...2...Jm..L)=.
0x0000000000000030	d9	62	da	01	e0	54	02	00	00	00	00	00	00	00	00	00	.....T.....
0x0000000000000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000000050	1f	50	e0	4f	d0	20	ea	3a	69	10	a2	d8	08	00	2b	30	..P.O.....
0x0000000000000060	30	9d	3a	00	2e	80	3a	cc	bf	b4	2c	db	4c	42	b0	29	0.....,LB..)
0x0000000000000070	7f	e9	9a	87	c6	41	26	00	01	00	26	00	ef	be	11	00	.....A&.....&
0x0000000000000080	00	00	e4	86	36	11	08	d7	01	f4	2d	e0	eb	4a	6d	00	.....6.....-..Jm
0x0000000000000090	da	01	83	0d	21	0c	4b	6d	da	01	14	00	54	00	31	00	.....!Km...T..l
0x00000000000000a0	00																

# DETECTIA DE FORMAT

## PYTHON FILE TYPE DETECTOR

```
import os

# magic numbers pentru tipul de fisier.
file_signatures = {
    b'\xff\xd8\xff\xe0': 'JPEG image',
    b'\x89\x50\x4e\x47\x0d\x0a\x1a\x0a': 'PNG image',
    # adaugare semnaturi dupa dorinta
}

def detect_file_type(file_path):
    with open(file_path, 'rb') as file:
        header = file.read(8) # citeste primii 8 bytes
        for signature, file_type in file_signatures.items():
            if header.startswith(signature):
                return file_type
    return "Unknown file type"

def search_files(directory):
    for root, __, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            file_type = detect_file_type(file_path)
            print(f'File "{file_path}" is a {file_type}.')

directory = input("Calea catre directorul de interes: ")
search_files(directory)
```

Acest script Python definește un dicționar de semnături de fișiere pentru diferite tipuri și oferă o funcție pentru a detecta tipul de fișier citind începutul fișierului. Apoi caută într-un director specificat și tipărește căile fișierelor împreună cu tipurile detectate.

Această abordare este mult mai fiabilă pentru manipularea datelor binare și permite o extensie ușoară pentru a accepta mai multe tipuri de fișiere prin adăugarea semnăturilor acestora în dicționar.

### Rezultat consolă:

- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\s(3).gif" is a Unknown file type.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\s(4).gif" is a Unknown file type.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\s(5).gif" is a Unknown file type.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\s(6).gif" is a Unknown file type.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\spectral forecast.png" is a PNG image.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\t.png" is a PNG image.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\waveform.png" is a PNG image.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\0\signals.zip" is a Unknown file type.
- File "C:\Users\Elitebook\Desktop\IMG\0\0\New folder\0\sp.html" is a Unknown file type.

# CE NE INDICĂ

## DISCREPANȚĂ ÎNTRE SEMNĂTURA MAGICĂ ȘI EXTENSIE?

- Discrepanță între semnătura magică și extensie?
- Care asociere este suspectă și care nu?

nume\_fisier.jpg

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000400000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZ.....
0x0000000000400010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.....@.....
0x0000000000400020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000400030	00	00	00	00	00	00	00	00	00	00	00	00	00	b8	00	00	.....
0x0000000000400040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	.....!..L.!Th
0x0000000000400050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program canno
0x0000000000400060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
0x0000000000400070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....\$......
0x0000000000400080	8f	8a	f9	db	cb	eb	97	88	cb	eb	97	88	cb	eb	97	88	.....
0x0000000000400090	48	f7	99	88	ca	eb	97	88	a2	f4	9e	88	ca	eb	97	88	H.....
0x00000000004000a0	22	f4	9a	88	ca	eb	97	88	52	69	63	68	cb	eb	97	88	".....Rich....
0x00000000004000b0	00	00	00	00	00	00	00	00	50	45	00	00	4c	01	03	00	.....PE..L....
0x00000000004000c0	f6	3f	f6	65	00	00	00	00	00	00	00	00	e0	00	0f	01	..?.e.....

nume\_fisier

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0000000000400000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZ.....
0x0000000000400010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.....@.....
0x0000000000400020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000000400030	00	00	00	00	00	00	00	00	00	00	00	00	00	b8	00	00	.....
0x0000000000400040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	.....!..L.!Th
0x0000000000400050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program canno
0x0000000000400060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
0x0000000000400070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....\$......
0x0000000000400080	8f	8a	f9	db	cb	eb	97	88	cb	eb	97	88	cb	eb	97	88	.....
0x0000000000400090	48	f7	99	88	ca	eb	97	88	a2	f4	9e	88	ca	eb	97	88	H.....
0x00000000004000a0	22	f4	9a	88	ca	eb	97	88	52	69	63	68	cb	eb	97	88	".....Rich....
0x00000000004000b0	00	00	00	00	00	00	00	00	50	45	00	00	4c	01	03	00	.....PE..L....
0x00000000004000c0	f6	3f	f6	65	00	00	00	00	00	00	00	00	e0	00	0f	01	..?.e.....

# CUM OBTINEM SEMATURILE MAGICE?

## ALINIEM FISIERE CU EXTENSII DE ACELASI FEL

```
import os

def citeste_primii_30_bytes(fisier):
    with open(fisier, 'rb') as f:
        return f.read(30)

def gaseste_sematura_magica(fisiere):
    semnatura_magica = list(citeste_primii_30_bytes(fisiere[0]))
    for fisier in fisiere[1:]:
        bytes_curenti = list(citeste_primii_30_bytes(fisier))
        for i in range(len(semnatura_magica)):
            if semnatura_magica[i] != bytes_curenti[i]:
                semnatura_magica[i] = None
    semnatura_magica = [byte for byte in semnatura_magica if byte is not None]
    return bytes(semnatura_magica)

def afiseaza_sematura_magica_director(director):
    fisiere = [os.path.join(director, fisier) for fisier in os.listdir(director) \
               if os.path.isfile(os.path.join(director, fisier))]

    for fisier in fisiere:
        bytes_fisier = citeste_primii_30_bytes(fisier)
        print(f"{os.path.basename(fisier)}: {bytes_fisier.hex()}")
        semnatura_magica = gaseste_sematura_magica(fisiere)
        print(f"\nSemnatura pentru setul '{director}': {semnatura_magica.hex()}")

director = 'C:\\\\Users\\Elitebook\\Desktop\\IMG\\1'
afiseaza_sematura_magica_director(director)
```

### Consola:

- 1.jpg: ffd8ffe20bf84943435f50524f46494c4500010100000be800000000200
- 2.jpg: ffd8ffe000104a46494600010200000100010000ffe00042a00ffe20bf8
- 3.jpg: ffd8ffe000104a46494600010101007800780000ffe10022457869660000
- 4.jpg: ffd8ffe20bf84943435f50524f46494c4500010100000be800000000200
- 5.jpg: ffd8ffe000104a46494600010100000100010000ffe201d84943435f5052
- 6.jpg: ffd8ffe000104a46494600010100000100010000ffe201d84943435f5052
- 7.jpg: ffd8ffe000104a46494600010100000100010000ffe201d84943435f5052
- 8.jpg: ffd8ffe000104a46494600010100000100010000ffe201d84943435f5052
- 9.jpg: ffd8ffe000104a46494600010100000100010000ffe201d84943435f5052
- Semnatura pentru setul 'C:\\Users\\Elitebook\\Desktop\\IMG\\1': ffd8ff

Aceste semnături sunt utile pentru identificarea rapidă a tipului de fișier, dar nu sunt singura metodă de determinare a tipului de fișier și pot fi, de asemenea, ușor falsificate în anumite cazuri de malware sau fișiere corupte.



C.6.2

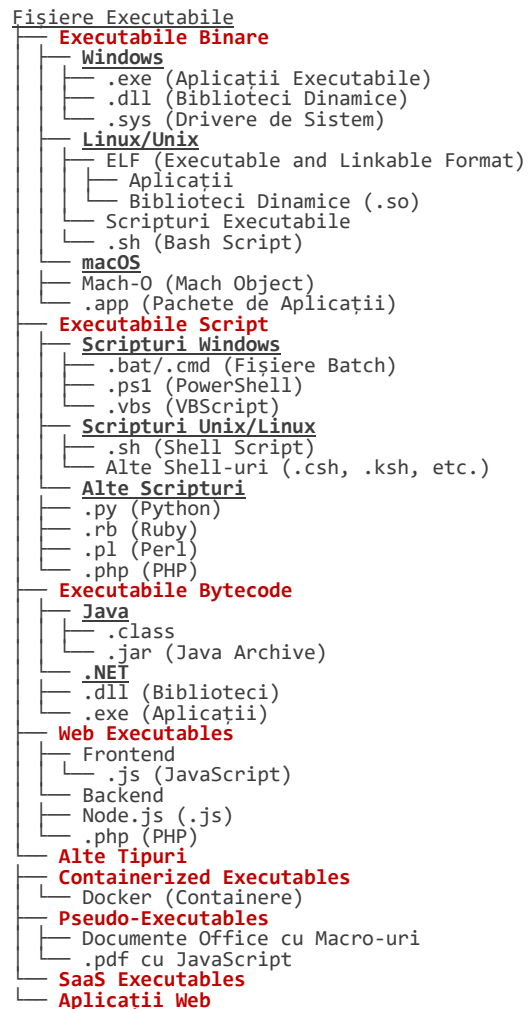
# FIȘIERE EXECUTABILE INTERPRETATE VS COMPILATE





# DIFERITELE TIPURI DE FIȘIERE EXECUTABILE

## SUB FORMA UNUI ARBORE



Această diagramă prezintă o viziune generală a tipurilor de fișiere executabile, grupate în categorii largi.

- **Executabile Binare** reprezintă programele și bibliotecile compilate nativ pentru un sistem de operare.
- **Executabile Script** cuprind scripturi interpretate la runtime de diferite motoare de scripting sau shell-uri.
- **Executabile Bytecode** sunt un tip intermediar, necesitând o mașină virtuală pentru execuție.
- **Web Executables** evidențiază rolul limbajelor de programare web atât pe partea de client, cât și pe cea de server.
- **Alte Tipuri** acoperă categorii mai noi sau mai puțin tradiționale de executabile, reflectând diversitatea și evoluția tehnologică din domeniul IT.

# FIȘIERELE .EXE (EXECUTABLE FILES)

## STRUCTURA GENERALĂ A UNUI FIȘIER .EXE ÎN FORMAT PE

Un executabil Windows, de obicei în formatul PE (Portable Executable), poate conține mai multe tipuri de secțiuni, fiecare având un rol specific în funcționarea aplicației. Aceste secțiuni sunt definite în antetul executabilului.

### Fișier .exe (Executable File)

- **Antetul DOS** (DOS Header)
  - **POINȚEază către Antetul PE**
- **Antetul PE** (Portable Executable Header)
  - **Antetul File** (Informații generale despre fișier, cum ar fi tipul mașinii, numărul de secțiuni, timestamp-ul, etc.)
  - **Antetul Optional** (Adresa punctului de intrare, versiunile minime ale OS necesare, dimensiunile stivei și ale heap-ului, etc.)
  - **Tabelele de Secțiuni** (Describe secțiunile fișierului, incluzând permisiunile de memorie și dimensiunile lor)
- **Secțiuni**
  - **.text** (Codul executabil al aplicației)
  - **.data** (Date inițializate, inclusiv variabile globale)
  - **.rdata** (Date doar pentru citire, cum ar fi constantele și șirurile literale)
  - **.bss** (Secțiune pentru date neinițializate, rareori prezent direct în .exe)
  - **.edata** (Exporturile fișierului, rareori utilizat în .exe)
  - **.idata** (Importurile fișierului, inclusiv funcții din DLL-uri externe)
  - **.reloc** (Informații de realocare, important pentru aplicațiile care suportă ASLR - Address Space Layout Randomization)
  - **.rsrc** (Resurse ale aplicației, cum ar fi icoane, meniuri, și dialoguri)
  - **Alte secțiuni specifice** (Pot include secțiuni personalizate de dezvoltatori)
- **Tabelul de Import** (Optional)
- **Lista de funcții din alte DLL-uri necesare pentru execuție**

Fișierele .exe (Executable files) în Windows reprezintă aplicațiile executabile propriu-zise, fiind principalul tip de fișiere folosit pentru a lansa programe. La fel ca fișierele .dll și .sys, și acestea utilizează formatul Portable Executable (PE) pentru structura lor, care este flexibil și suportă o varietate de scenarii de utilizare, de la aplicații simple la jocuri complexe și aplicații cu interfață grafică.

Fișierele .exe sunt, în esență, pachetele care conțin tot codul și resursele necesare pentru a rula o aplicație pe un sistem Windows. Punctul de intrare specificat în antetul opțional indică locul din secțiunea .text unde începe execuția programului. În timp ce secțiunile .data și .rdata stochează datele necesare rulării, secțiunea .rsrc este folosită pentru a încorpora resurse statice în executabil. Un aspect important al fișierelor .exe este că, deși pot include informații pentru importuri și exporturi (prin .idata și .edata), de obicei, ele nu exportă funcții către alte programe în modul în care o fac fișierele .dll. În schimb, ele sunt punctul de start pentru execuția aplicațiilor și pot face apeluri către funcțiile din bibliotecile dinamice partajate pentru a accesa funcționalități suplimentare.

# FIȘIERELE .SYS

## DIAGRAMA GENERALĂ A STRUCTURII UNUI FIȘIER .SYS (ȘI, PRIN EXTENSIE, A ORICĂRUI FIȘIER PE)

### Fișier .sys (Driver de Sistem)

- **Antetul DOS** (DOS Header)
  - **POINȚEază către Antetul PE**
- **Antetul PE** (Portable Executable Header)
  - **Antetul File** (Informații generale despre fișier)
  - **Antetul Optional** (Setări specifice, adrese de intrare, etc.)
  - **Tabelele de Secțiuni** (Descrie secțiunile fișierului)
- **Secțiuni**
  - **.text** (Codul executabil)
  - **.data** (Date inițializate)
  - **.rdata** (Date doar pentru citire)
  - **.bss** (Date neinițializate)
  - **.edata** (Exporturile fișierului)
  - **.idata** (Importurile fișierului)
  - **.reloc** (Informații pentru realocare)
  - **.rsrc** (Resurse, ex. icoane)
  - **Alte secțiuni specifice** (Depinde de driver)
- **Tabelul de Import** (Optional)
- **Lista de funcții din alte DLL-uri necesare**

Fișierele .sys din Windows sunt, de obicei, drivere de sistem sau fișiere de driver de dispozitiv care permit hardware-ului să comunice cu sistemul de operare Windows. Structura lor internă este similară cu cea a fișierelor executabile PE (Portable Executable), care este formatul standard pentru executabile, DLL-uri și drivere în sistemele de operare Windows. Aceasta înseamnă că fișierele .sys conțin atât cod executabil, cât și date, și sunt organizate într-o serie de secțiuni cu diferite scopuri.

Fișierele .sys pot conține, de asemenea, informații specifice driverelor, cum ar fi tabelul de realocare, care este folosit când driverul este încărcat într-o adresă de memorie diferită de cea presupusă inițial. De asemenea, pot include secțiuni pentru datele specifice driverului, cum ar fi configurări sau informații despre dispozitiv.

Structura detaliată a fișierului PE este complexă și permite o mare flexibilitate în ceea ce privește modul în care datele și codul sunt organizate și stocate. Această structură facilitează, de asemenea, utilizarea funcțiilor din alte biblioteci, încărcarea dinamică a resurselor și adaptabilitatea la diferite versiuni ale sistemului de operare Windows.

# FIȘIERELE .DLL (DYNAMIC LINK LIBRARIES)

STRUCTURA UNUI FIȘIER .DLL ESTE, PRIN URMARE, SIMILARĂ CU CEA A ALTOR FIȘIERE PE

## Fișier .dll (Dynamic Link Library)

- **Antetul DOS (DOS Header)**
  - **POINȚEază către Antetul PE**
- **Antetul PE (Portable Executable Header)**
  - **Antetul File** (Informații generale despre fișier)
  - **Antetul Optional** (Setări specifice, adrese de intrare, etc.)
  - **Tabelele de Secțiuni** (Describe secțiunile fișierului)
- **Secțiuni**
  - **.text** (Codul executabil)
  - **.data** (Date inițializate)
  - **.rdata** (Date doar pentru citire, inclusiv informații de import/export)
  - **.bss** (Date neinițializate, rareori prezent în DLL-uri)
  - **.edata** (Exporturile fișierului, funcțiile disponibile pentru alte programe)
  - **.idata** (Importurile fișierului, funcțiile utilizate din alte DLL-uri)
  - **.reloc** (Informații pentru realocare, DLL-uri care trebuie să fie flexibile la adresele de memorie)
  - **.rsrc** (Resurse, ex. icoane, șiruri de caractere)
  - **Alte secțiuni specifice** (Depinde de bibliotecă)
- **Tabelul de Import (Optional)**
- **Lista de funcții din alte DLL-uri necesare**

Fișierele .dll (Dynamic Link Libraries) în Windows sunt biblioteci partajate care conțin cod și date care pot fi utilizate de mai multe programe simultan. La fel ca fișierele .sys și executabilele standard, fișierele .dll folosesc formatul Portable Executable (PE) pentru organizarea și stocarea conținutului lor.

Fișierele .dll pot fi utilizate pentru a oferi o varietate de funcții, de la operații simple de manipulare a datelor până la execuția de UI complexe sau operații de rețea. Avantajul principal al utilizării DLL-urilor este reutilizarea codului și eficiența memoriei, deoarece un singur DLL poate fi încărcat în memorie o singură dată, dar utilizat de mai multe aplicații simultan.

Diferența majoră între un fișier .dll și un executabil (.exe) sau un fișier de driver (.sys) constă în modul de utilizare și scop. DLL-urile sunt concepute pentru a fi încărcate dinamic de alte programe la nevoie, oferind astfel o modularitate crescută și permitând actualizări ale funcționalităților fără a necesita recompilarea aplicațiilor care le folosesc.

# DIFERENȚE SEMNIFICATIVE .EXE, .DLL, ȘI .SYS



## Scop și Utilizare

- .exe (Executable Files)**: Sunt fișierele executabile propriu-zise, care conțin programe ce pot fi rulate direct de utilizatori sau de alte aplicații. Fiecare fișier .exe reprezintă o aplicație separată sau un proces executabil.
- .dll (Dynamic Link Libraries)**: Conțin cod și date care pot fi utilizate de mai multe programe simultan, facilitând reutilizarea codului și eficiența memoriei prin partajarea funcționalităților comune. Bibliotecile .dll nu pot fi executate direct de utilizator; în schimb, ele sunt încărcate de aplicațiile care necesită funcțiile sau datele pe care le oferă.
- .sys (System Files)**: Sunt, în general, drivere de dispozitiv sau fișiere de sistem care interacționează la un nivel scăzut cu hardware-ul sau cu nucleul sistemului de operare. Fișierele .sys sunt esențiale pentru funcționarea hardware-ului și pentru facilitarea comunicării între componentele hardware și restul sistemului de operare.

## Execuție și Încărcare

- .exe**: Sunt lansate direct de sistemul de operare ca procese noi, cu un punct de intrare definit care inițializează execuția programului.
- .dll**: Sunt încărcate în spațiul de memorie al unui proces existent, fie la lansarea programului care le referă, fie dinamic, în timpul execuției. Ele nu au un punct de intrare ca fișierele .exe, deși pot defini funcții de inițializare și curățare.
- .sys**: Sunt încărcate de sistemul de operare la nivel de kernel, de obicei în timpul procesului de boot sau când un nou dispozitiv este conectat, pentru a permite interacțiunea dintre hardware și sistemul de operare.

## Exporturi și Importuri

- .exe**: Deși pot importa funcții și date din .dll, de obicei nu exportă funcții către alte programe.
- .dll**: Proiectate pentru a exporta funcții și date către alte programe, inclusiv alte .dll și fișiere .exe.
- .sys**: Similar cu .dll, pot exporta funcționalități, dar acestea sunt în mare parte orientate spre furnizarea de interfețe la nivel de kernel sau drivere de dispozitiv.

## Contextul de Securitate și Stabilitate

- .exe** și **.dll**: Rulează în spațiul de utilizator, separându-le de nucleul sistemului de operare pentru a asigura stabilitatea și securitatea.
- .sys**: Rulează în spațiul de kernel, având acces complet la sistemul de operare și hardware, ceea ce le conferă un nivel ridicat de privilegiu și, prin urmare, necesită precauții de securitate.

# FIȘIERELE EXECUTABLE & EXTENSIILE

Fișierele Portable Executable (PE) în sistemele Windows pot avea mai multe extensii, reflectând diferite scopuri sau moduri de utilizare, chiar dacă structura lor internă este similară. Diferențele principale între aceste tipuri de fișiere sunt, de obicei, în modul în care sunt utilizate de sistemul de operare și în intenția lor specifică, mai degrabă decât în structura lor internă. Iată o listă a extensiilor comune pentru fișierele PE, în afară de .exe:

- **.dll** - Dynamic Link Libraries: Biblioteci partajate care conțin cod și date care pot fi folosite de mai multe programe simultan. Sunt esențiale pentru reutilizarea codului și pentru modularitatea aplicațiilor Windows.
- **.sys** - Fișiere de sistem sau drivere de dispozitiv: Sunt folosite de sistemul de operare pentru a interacționa cu hardware-ul PC-ului. Acestea rulează la un nivel de privilegiu mai înalt în sistemul de operare.
- **.scr** - Screen savers: Tehnic, sunt executabile, dar sunt folosite de Windows ca protecții de ecran. Windows le tratează special, permițându-le să fie activate ca protecții de ecran.
- **.ocx** - OLE Control Extension: Folosite pentru controalele ActiveX, care sunt obiecte reutilizabile ce pot fi încorporate în aplicații, în special în mediul web pentru Internet Explorer.
- **.cpl** - Control Panel items: Extensie folosită pentru elementele din Panoul de Control, permitând utilizatorilor să acceseze diferite setări ale sistemului de operare.
- **.drv** - Vechi drivere de dispozitiv: În sistemele mai vechi Windows, acestea erau folosite pentru driverele de dispozitiv, deși în sistemele moderne Windows, driverele sunt de obicei .sys.
- **.pif** - Program Information Files: Folosite în sistemele mai vechi pentru a defini cum ar trebui executate programele DOS în Windows, inclusiv setările de memorie, setările de tastatură etc. Deși nu sunt executabile în sine, ele indică și configurează execuția unui fișier executabil.

Toate aceste tipuri de fișiere sunt bazate pe formatul PE și pot conține cod executabil, dar sunt distinse prin modul în care sunt utilizate de Windows. În cazul fișierelor .scr și .pif, deși se bazează pe formatul PE, modul lor specific de utilizare le diferențiază de fișierele .exe standard. Fișierele .scr sunt tratate ca protecții de ecran și pot fi configurate prin dialogul de proprietăți al protecției de ecran, în timp ce fișierele .pif sunt folosite mai degrabă pentru a oferi informații despre cum să ruleze o aplicație în medii specifice decât pentru a conține cod executabil propriu-zis.



# FIȘIERELE .COM



Fișierele .com sunt un tip vechi de executabil folosit în sistemele de operare DOS și primele versiuni de Windows. Spre deosebire de formatul Portable Executable (PE) utilizat de .exe, .dll, .sys și alte extensii în sistemele de operare moderne Windows, fișierele .com sunt mult mai simple din punct de vedere structural.

## Structura Fișierelor .com

**Simplă și liniară.** Fișierele .com nu au antete sau secțiuni. Ele sunt executate în modul real al procesorului, care oferă acces direct și complet la hardware și la toate adresele de memorie.

**Dimensiune limitată.** Fișierele .com trebuie să încapă într-un singur segment de memorie de 64 KB, deoarece sunt executate în modul real, care nu suportă adrese de memorie peste această limită. Aceasta include codul, datele și stiva programului.

**Cod de start la început.** Execuția unui fișier .com începe de la primul byte din fișier, fără nicio instrucțiune de inițializare specială sau setup, spre deosebire de fișierele .exe PE, care au un punct de intrare specificat în antet.

**Lipsa funcționalităților avansate.** Nu suportă caracteristici moderne precum bibliotecile dinamice, execuția în modul protejat sau alte funcționalități complexe disponibile în formatul PE.

Principala diferență între fișierele .com și cele în format PE (cum ar fi .exe, .dll, .sys) este complexitatea. Fișierele .com sunt mult mai simple, fără structură internă complexă, menite pentru execuție directă și rapidă în medii limitate, precum DOS-ul. Pe de altă parte, formatul PE suportă o multitudine de caracteristici avansate necesare pentru aplicațiile moderne, inclusiv modularitatea, compatibilitatea cu mai multe platforme de hardware și suportul pentru execuție în medii de operare protejate și multi-tasking.

# FIȘIERELE .COM COMPATIBILITATE



În teorie, Windows menține o compatibilitate pentru a executa fișiere .com tradiționale din era DOS, dar în practică, modul în care aceste fișiere sunt tratate și executate în versiunile moderne de Windows, cum ar fi Windows 10 sau Windows 11, este diferit față de modul în care erau executate în DOS sau primele versiuni de Windows.

Când încerci să rulezi un fișier .com pe un sistem Windows modern:

**Compatibilitatea.** Windows încearcă să ruleze fișierul .com într-un mediu virtualizat sau emulat pentru DOS, de obicei prin intermediul NTVDM (NTVirtual DOS Machine) pe versiunile de 32 de biți ale Windows. Pe sistemele de 64 de biți, suportul NTVDM este absent, ceea ce înseamnă că fișierele .com nu pot fi rulate direct fără un emulator de terțe părți, cum ar fi DOSBox.

**Securitatea.** Sistemele moderne de operare, inclusiv Windows, impun restricții stricte de securitate și izolare a proceselor, limitând accesul direct la hardware și la spațiul de memorie. Aceasta înseamnă că chiar dacă un fișier .com este executat, el nu va avea același nivel de acces direct la resursele sistemului cum avea în DOS.

**Compatibilitatea cu hardware.** Majoritatea hardware-ului modern și a mediilor de sistem de operare sunt construite în jurul unor modele de securitate și arhitectură care nu mai suportă modul real sau tehnicile de programare utilizate în fișierele .com. Deci, chiar dacă teoretic poți plasa cod mașină într-un fișier .com și încerci să îl execuți pe un sistem Windows modern, există mai multe straturi de compatibilitate și securitate care vor influența cum (și dacă) acel cod va fi executat. Pe un sistem de 64 de biți, vei avea nevoie de un emulator pentru a rula fișierele .com.

C.6.3

# TIPUL DE INFORMAȚII, OFUSCAREA ȘI DETECTAREA CRIPTĂRII



# CUM DETERMINĂM DACĂ UN FIȘIER ESTE TEXT (HUMAN-READABLE)?

Determinarea dacă un fișier este text (adică, lizibil de către om) implică câteva verificări și euristici, deoarece nu există o semnătură unică care să distingă în mod clar fișierele text de cele binare. Iată câțiva pași pe care îi poți urma pentru a determina dacă un fișier este text:

## 1. Verificarea Caracterelor de Control

Fișierele text sunt de obicei compuse din caractere imprimabile și câteva caractere de control specifice, cum ar fi newline (LF, \n, hex 0A) sau carriage return (CR, \r, hex 0D). O metodă comună este să verifici dacă fișierul conține în principal caractere imprimabile (în codificarea ASCII, caracterele imprimabile sunt în intervalul hex 20 până la 7E, plus 0A și 0D pentru noi linii și întoarcere la începutul liniei) și să ai un număr foarte mic sau deloc de caractere din afara acestui interval.

## 2. Verificarea Prezenței Caracterelor Non-Text

Fișierele binare vor conține adesea un număr mare de caractere în afara intervalului de caractere imprimabile ASCII. Dacă găsești o cantitate semnificativă de date în afara intervalului 20-7E, 0A, și 0D într-un eșantion de date din fișier, este probabil ca fișierul să fie binar.

## 3. Verificarea Antetului Fișierului

Unele fișiere binare încep cu semnături specifice (magic numbers), cum am menționat anterior. Poți verifica primele câteva octeți ai fișierului pentru aceste semnături pentru a identifica rapid dacă fișierul este de un tip cunoscut non-text (de exemplu, imagini, arhive, executabile).

## 4. Utilizarea Utilităților Sistemului

Sistemele de operare oferă adesea utilitare care pot ajuta la identificarea tipului de fișier. De exemplu, pe sistemele Unix-like (inclusiv Linux și macOS), poți folosi comanda `file` pentru a analiza un fișier și a obține o descriere a tipului său, bazată pe conținutul său și pe semnăturile de fișiere cunoscute (`file [nume_fisier]`).

## 5. Tentative de Decodificare

Încercarea de a decoda fișierul folosind codificări de text comune (cum ar fi UTF-8, ASCII) și verificarea dacă rezultatul este sensibil poate oferi, de asemenea, indicii. Fișierele text ar trebui să decodeze corect într-un text lizibil, în timp ce fișierele binare adesea nu vor.

În practică, combinarea mai multor dintre aceste metode va oferi cele mai bune rezultate. Este important de reținut că unele fișiere pot fi teoretic binare, dar să conțină secțiuni lizibile de text (de exemplu, fișiere executabile cu șiruri de caractere lizibile). Prin urmare, determinarea dacă un fișier este „text” poate uneori să necesite un context suplimentar sau o analiză mai detaliată.

# CUM DETERMINĂM DACĂ UN FIȘIER ESTE TEXT (HUMAN-READABLE)?

```
def is_text_file(filepath, sample_size=512):  
  
    # Verifică dacă fișierul este text, bazat pe  
    # prezența caracterelor non-text într-un eșantion.  
  
    text_chars = bytearray({7, 8, 9, 10, 12, 13, 27} | \  
                           set(range(0x20, 0x100)) - {0x7f})  
  
    with open(filepath, 'rb') as file:  
        sample = file.read(sample_size)  
    if not sample: # Fișierul este gol  
        return True  
  
    # Verifică dacă sample conține doar  
    # caractere permise pentru fișiere text.  
  
    return all((c in text_chars) for c in sample)  
  
# Solicită utilizatorului să introducă calea către fișier  
filepath = input("Introduceți calea către fișierul de verificat: ")  
  
# Verifică dacă fișierul este text sau binar și afișează rezultatul  
if is_text_file(filepath):  
    print("Fișierul selectat este text (lizibil de om).")  
else:  
    print("Fișierul selectat este binar (pentru mașini/computere).")
```

- Pentru a crea un program Python cu o interfață grafică (GUI) care verifică dacă un fișier este text (lizibil de către om) sau binar, putem folosi biblioteca Tkinter, care este inclusă standard în majoritatea instalațiilor Python. Acest exemplu de cod va include o fereastră pentru selectarea unui fișier și va afișa rezultatul analizei.
- Codul va utiliza o euristică simplă bazată pe prezența caracterelor non-text într-un eșantion din fișier pentru a determina dacă este considerat text sau binar.
- Dacă dorești o versiune simplificată, fără GUI, care să verifice dacă un fișier este text (lizibil de către om) sau un șir de biți pentru mașini (binar), poți folosi următorul cod Python. Acest script va solicita utilizatorului să introducă calea către fișierul pe care dorește să îl verifice și va afișa rezultatul în linia de comandă:
- Cum să folosești acest cod:
  - Salvează codul într-un fișier .py, de exemplu, check\_file\_type.py.
  - Deschide un terminal sau prompt de comandă.
  - Navighează la directorul unde ai salvat scriptul.
  - Rulează scriptul folosind Python, de exemplu, python check\_file\_type.py.
  - Introdu calea completă a fișierului pe care dorești să îl verifici atunci când scriptul o solicită.
  - Scriptul va analiza fișierul și va afișa dacă este considerat a fi text sau binar.
- Acest cod este util pentru scenarii rapide de verificare a tipului de fișier direct din linia de comandă, fără necesitatea unei interfețe grafice.
- Introduceți calea către fișierul de verificat:  
C:\Users\Elitebook\Desktop\OffVis\DevExpress.XtraTreeList.v9.1.dll
- Fișierul selectat este binar (pentru mașini/computere).
- (base) PS C:\Users\Elitebook>

# CUM IDENTIFICĂM TIPUL DE CODIFICARE AL UNUI FIȘIER TEXT?

Diversele codificări de caractere există pentru a gestiona modul în care caracterele din text sunt reprezentate ca numere în memoria computerelor. Textul pe care îl citim și îl scriem, fie că este în limbajul nostru nativ sau în simboluri speciale, trebuie să fie stocat și procesat într-o formă binară pentru ca sistemele de computere să îl poată manipula. Fiecare codificare definește o hartă unică, sau un set de hărți, pentru conversia între caractere și numerele binare care le reprezintă.

## Codificări de caractere:

- **UTF-8.** Este o codificare variabilă a lungimii care folosește între unu și patru octeți (bytes) pentru a reprezenta fiecare caracter din Universal Character Set (UCS) sau Universal Coded Character Set (Unicode). Este compatibilă cu ASCII și este cea mai comună codificare folosită pe web, deoarece poate reprezenta fiecare caracter din Unicode și este eficientă pentru texte ce folosesc frecvent caractere ASCII.
  - **ISO-8859-1.** Cunoscută și ca Latin-1, este o codificare cu un singur octet care poate reprezenta până la 256 de caractere diferite. Este proiectată pentru a acoperi majoritatea limbilor vest-europene, inclusiv caracterele diacritice.
  - **CP1252.** O codificare cu un singur octet folosită de sistemele Windows în limba engleză și în alte limbi vest-europene. Este o extensie a ISO-8859-1 și include caractere suplimentare, cum ar fi simboluri tipografice și caractere cu diacritice.
  - **Latin-1.** Termenul "Latin-1" se referă adesea la ISO-8859-1, dar poate fi, de asemenea, folosit în mod informal pentru a se referi la CP1252, deoarece ambele sunt foarte similare și sunt uneori confundate.
  - **ASCII.** Este o codificare standardizată pentru caracterele englezești, folosind 7 biți pentru a reprezenta 128 de simboluri diferite, inclusiv literele din alfabetul englez, cifrele, simbolurile de punctuație și controalele de bază ale terminalului. Este cea mai simplă și mai veche codificare și este inclusă ca o submulțime în multe alte codificări, cum ar fi cele de mai sus.
- 
- Aceste codificări au fost create pentru a gestiona diversitatea limbilor și simbolurilor utilizate în computere, având în vedere că diferite regiuni și limbi au nevoie de seturi diferite de caractere. De exemplu, înainte de adoptarea pe scară largă a Unicode, mulți seturi de caractere erau limitate la un singur limbaj sau la un grup de limbi asemănătoare. În prezent, Unicode și în particular UTF-8 au devenit standardele dominante pentru că sunt capabile să reprezinte o gamă largă de caractere din diferite limbi și simboluri.
  - Descrierea codului dat este o funcție în Python care încearcă să identifice codificarea textului unui fișier dat. Ea încearcă să citească fișierul cu diferite codificări și, dacă reușește să citească fișierul fără să arunce o excepție, presupune că acea codificare este corectă. Funcția începe cu presupunerea că textul este codificat în UTF-8, apoi trece la ISO-8859-1, CP1252 și Latin-1. Dacă niciuna nu reușește, va presupune că fișierul este codificat în ASCII, dar cu o încredere redusă. Această abordare este euristică și nu garantează detectarea corectă a codificării pentru orice fișier text.



# CUM IDENTIFICĂM TIPUL DE CODIFICARE AL UNUI FIȘIER TEXT?

```
def detect_file_encoding(filepath):
    encodings = ['utf-8', 'iso-8859-1', 'cp1252', 'latin1'] # encodari commune.
    for enc in encodings:
        try:
            with open(filepath, 'r', encoding=enc) as file:
                file.read()
                return enc, 1.0 # daca nu avem exceptii, stim sigur ...
        except (UnicodeDecodeError, LookupError):
            continue
    return 'ascii', 0.5 # daca altceva nu este detectat presupunem ...

# Solicită utilizatorului să introducă calea către fișierul de verificat.
filepath = input("Introduceți calea către fișierul de verificat: ")

# Detectează codificarea și afișează rezultatul.
encoding, confidence = detect_file_encoding(filepath)
print(f"Codificarea detectată: {encoding} cu o confidență de {confidence*100:.2f}%")
```

## Output:

```
Introduceți calea către fișierul de verificat: C:\Users\Elitebook\Desktop\test.exe
Codificarea detectată: iso-8859-1 cu o confidență de 100.00%
(base) PS C:\Users\Elitebook>
```

**1.Funcția detect\_file\_encoding(filepath).** Aceasta este o funcție definită pentru a ghici codificarea unui fișier text. Ea acceptă un singur argument, filepath, care este calea către fișierul ce trebuie verificat.

**2.Lista encodings.** Conține o listă de codificări de caractere comune pe care scriptul le va testa. Encodările încercate sunt:

- utf-8. O codificare Unicode care poate reprezenta toate caracterele standard internaționale și este foarte des utilizată pe web și în documentele moderne.
- iso-8859-1. Cunoscută și ca Latin-1, este o codificare pentru limbile vest-europene și poate reprezenta caractere din aceste limbi.
- Cp1252. O codificare de caractere folosită și de Windows în sistemele sale occidentale; este similară cu ISO-8859-1, dar include și caractere grafice suplimentare.
- Latin1. Un alt nume pentru ISO-8859-1.

**3.Buclele for și try-except.** Scriptul încearcă să deschidă fișierul folosind fiecare codificare din listă. Dacă fișierul poate fi citit fără să declanșeze o excepție UnicodeDecodeError (care ar indica că codificarea este greșită pentru conținutul fișierului) sau LookupError (care apare atunci când se încearcă utilizarea unei codificări necunoscute), atunci codificarea este considerată corectă și funcția returnează numele codificării împreună cu un nivel de încredere de 100%.

**4.În caz de eșec.** Dacă niciuna dintre codificările testate nu reușește să deschidă și să citească fișierul corect, funcția returnează ascii ca o presupunere de bază, cu un nivel de încredere mai scăzut (50%).

**5.Interacțiunea cu utilizatorul.** Scriptul solicită utilizatorului să introducă calea către fișierul de verificat printr-o intrare la consolă.

**6.Apelul funcției și afișarea rezultatelor.** Funcția este apelată cu calea furnizată, și apoi scriptul afișează rezultatul detectării codificării, împreună cu nivelul de încredere asociat.

# CUM NE DĂM SEAMA DACĂ UN SCRIPT PYTHON (FIȘIER TEXT) A FOST OFUSCAT?

Ofuscarea este procesul de transformare a codului într-o formă care este dificil de înțeles pentru oameni, dar care rămâne complet funcțional. Scopul este adesea de a proteja proprietatea intelectuală sau de a îngreuna analiza și ingineria inversă a codului.

## 1. Nume de Variabile și Funcții Neobișnuite

Unul dintre cele mai comune semne ale ofuscării este utilizarea de nume de variabile, funcții și clase care sunt neobișnuite sau care par a fi generate aleatoriu (de exemplu, `aI_b2`, `xYz_123`, etc.).

## 2. Folosirea Excesivă a Codului Compact

Codul ofuscat poate conține linii foarte lungi de cod sau expresii încâlcite care efectuează operațiuni complexe într-o singură instrucțiune, ceea ce îl face dificil de citit și înțeles.

## 3. Encodarea sau Criptarea Datelor și Șirurilor

Codul ofuscat poate include șiruri de caractere sau date encodate sau criptate care sunt decodate la runtime, făcând analiza statică a codului dificilă.

## 4. Folosirea de Tehnici Avansate

Tehnici cum ar fi metaprogramarea, reflexia sau injecția de cod sunt uneori folosite în codul ofuscat pentru a executa dinamic părți de cod sau pentru a modifica comportamentul programului în moduri neașteptate.

## 5. Prezența Codului care Împiedică Decompilarea sau Analiza

Codul ofuscat poate conține și mecanisme proiectate să împiedice sau să îngreuneze decompilarea sau analiza statică, cum ar fi verificări ale mediului de execuție sau autodistrugerea codului în anumite condiții.

# DETECTARE AUTOMATĂ

## CUM NE DĂM SEAMA DACĂ UN SCRIPT PYTHON (FIȘIER TEXT) A FOST OFUSCAT?

```
import re
import token
import tokenize

def check_for_obfuscation(file_path):
    with open(file_path, 'rb') as file:
        tokens = tokenize.tokenize(file.readline)
        for toknum, tokval, _, _, _ in tokens:
            if toknum == token.NAME:
                if re.match(r'^[a-zA-Z][a-zA-Z0-9_]*$', tokval) is \
                    None or len(tokval) > 10:
                    print(f"Suspicious name found: {tokval}")

file_path = 'C:\\Users\\Elitebook\\Desktop\\ofcat.py'
check_for_obfuscation(file_path)
```

```
Suspicious name found: lDg0v5lX42t
Suspicious name found: l6DG0v5lX4Rt
Suspicious name found: lDg0v5lX42t
Suspicious name found: lDg0v5lX4Rt
Suspicious name found: lDg0v5lX42t
...
Suspicious name found: LdDG0v5lX42tf
Suspicious name found: Ld_DG0v5lX42tf
Suspicious name found: LdDG0v5lX42tx
Suspicious name found: LdDG0v5lX42tf
Suspicious name found: lDg0v5lX42t
Suspicious name found: l6DG0v5lX42t
Suspicious name found: l6DG0v5lX42t
Suspicious name found: lDg0v5lX42t
(base) PS C:\Users\Elitebook>
```

```
def lDg0v5lX42t(lDg0v5lX42t, lDg0v5lX42t=0x173, *lDg0v5lX42tf):

l6DG0v5lX4Rt="".join(reversed([chr(2*2*7*2),chr(0x66),chr(0o164),chr(0b11
10101),chr(0x20)]))).strip()
    if lDg0v5lX42t%2!=0 or len(lDg0v5lX42t*3)>-1:
        with open(lDg0v5lX42t,encoding=l6DG0v5lX4Rt) as lDg0v5lX42t:
            lDg0v5lX4Rt=lDg0v5lX42t.readlines()
            return lDg0v5lX4Rt
    elif lDg0v5lX42tf:
        while lDg0v5lX42tf:
            lDg0v5lX42tx=lDg0v5lX42tf.pop()
            lD_DG0v5lX42tf.append(lDg0v5lX42tx)
            return lDg0v5lX42tf
    else:
        lDg0v5lX42t=lDg0v5lX42t[len(lDg0v5lX42t)/2::3]*6
        return [l6DG0v5lX42t for l6DG0v5lX42t in lDg0v5lX42t]
```

Acest script simplu verifică numele de variabile și identifică cele care nu se potrivesc cu un pattern obișnuit sau care sunt neobișnuit de lungi, ce ar putea indica ofuscarea. Rețineți că acesta este un exemplu foarte simplu și nu acoperă toate tehnicile de ofuscare, servind mai mult ca un punct de pornire pentru analiza manuală.

C.6.4

# IDENTIFICAREA PUNCTULUI DE INTRARE ÎN EXECUTABILE



# CALCULAREA ENTRY POINT CE NE INTERESEAZA?

64 bit							
0	1	2	3	4	5	6	7
Signature 0x5A4D		DOS Header					
(0x0C) Pointer to PE Header							
DOS STUB							
0x0000	Signature 0x5A4D5000			Machine		#numberOfSections	
0x0008	TimeDateStamp			PointerToSymbolTable (deprecated)			
0x0010	# NumberOfSymbolTable (deprecated)			SizeOfOptionalHeader		characteristics	
0x0018	Magic		MajorLinker Version	MinorLinker Version	SizeOfCode (sum of all sections)		
0x0020	SizeOfInitializedData			SizeOfUninitializedData			
0x0028	AddressOfEntryPoint (RVA)			BaseOfCode (RVA)			
0x0030	BaseOfData (RVA)			ImageBase			
0x0038	SectionAlignment			FileAlignment			
0x0040	MajorOperating SystemVersion		Minor Operating SystemVersion		MajorImage Version		Minor Image Version
0x0048	MajorSubsystem Version		Minor Subsystem Version		Win32VersionValue (lower 16-bit)		
0x0050	SizeOfImage			SizeOfHeaders			
0x0058	Checksum (Images not checked)			Subsystem		DllCharacteristics	
0x0060	SizeOfStackReserve			SizeOfStackCommit			
0x0068	SizeOfHeapReserve			SizeOfHeapCommit			
0x0070	LoaderFlags (check Filled)			# NumberOfRvaAndSizes			
	ExportTable (RVA)			SizeOfExportTable			
	ImportTable (RVA)			SizeOfImportTable			
	ResourceTable (RVA)			SizeOfResourceTable			
	ExceptionTable (RVA)			SizeOfExceptionTable			
	CertificateTable (RVA)			SizeOfCertificateTable			
	BaseRelocationTable (RVA)			SizeOfBaseRelocationTable			
	Debug (RVA)			SizeOfDebug			
	ArchitectureData (RVA)			SizeOfArchitectureData			
	GlobalPtr (RVA)			00 00 00 00 00 00 00 00			
	TLSTable (RVA)			SizeOfTLSTable			
	LoadConfigTable (RVA)			SizeOfLoadConfigTable			
	BoundImport (RVA)			SizeOfBoundImport			
	ImportAddressTable (RVA)			SizeOfImportAddressTable			
	DelayImportDescriptor (RVA)			SizeOfDelayImportDescriptor			
	CLRRunTimeHeader (RVA)			SizeOfCLRRunTimeHeader			
	00 00 00 00 00 00 00 00			00 00 00 00 00 00 00 00			
				Name			
	VirtualSize (RVA)			VirtualAddress			
	SizeOfRawData (RVA)			PointerToRawData			
	PointerToRelocations (RVA)			PointerToLineNumbers			
	numberOfRelocations (RVA)			Characteristics			

64 bit							
0	1	2	3	4	5	6	7
Signature 0x5A4D		DOS Header					
DOS STUB							
Signature 0x50450000				Machine		#NumberOfSections	
TimeDateStamp				PointerToSymbolTable (deprecated)			
# NumberOfSymbolTable (deprecated)				SizeOfOptionalHeader		Characteristics	
Magic		MajorLinker Version	MinorLinker Version	SizeOfCode (sum of all sections)			
SizeOfInitializedData				SizeOfUninitializedData			
AddressOfEntryPoint (RVA)				BaseOfCode (RVA)			
BaseOfData (RVA)				ImageBase			

COFF Header

Standard COFF Fields

# În debugger \_

- Când deschideți un executabil (i.e. x64dbg), primul loc unde se oprește execuția este într-o funcție din **ntdll.dll** și nu în codul nostru din **.text**.

Când un fișier EXE este încărcat:

1. **Windows creează procesul** cu toate resursele și spațiul de adresă virtuală.
2. **Inițializează loader-ul intern** → cod care rulează din **ntdll.dll**.
3. **Abia apoi face salt către EntryPoint-ul definit în PE Header-ul executabilului.**

Funcții precum  
**RtlUserThreadStart**,  
**LdrpInitializeProcess**, etc.

Doar câteva dintre funcțiile din ntdll.dll care rulează exact în etapa 2:

RtlUserThreadStart este rutina de „bootstrap” care pornește firul principal al procesului. Ea apelează mai departe loader-ul și apoi transferă controlul către Entry Point-ul real. LdrpInitializeProcess face parte din loader-ul intern („Ldr”) și se ocupă de inițializarea procesului: încarcă toate DLL-urile necesare, rezolvă adresele funcțiilor importate și execută rutinele TLS/ CRT înainte să ajungă la codul nostru.

TLS în contextul încărcării unui proces Windows se referă la **Thread Local Storage**, mecanismul care alocă date private fiecărui fir de execuție, precum variabile statice declarate cu `__declspec(thread)` sau callback-urile de tip „TLS callbacks”.

CRT înseamnă **C Run-Time**, adică colecția de rutine de inițializare și biblioteci standard C/C++ (startup code, gestionarea heap-ului, printf, malloc, constructori/destructorii pentru obiecte globale etc.).



## Identificarea adresei antetului PE folosind imageBase + 0x3C

### X64dbg:

minimal.cpp (MessageBoxA).exe - PID: 17464 - Module: ntdll.dll - Thread: Main Thread 14028 - x64dbg

File View Debug Tracing Plugins Favourites Options Help Feb 19 2024 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols

RIP → 00007FF81EC00951 EB 00 jmp ntdll.7FF81EC00953

48:83C4 38 add rsp,38

00007FF81EC00953  
00007FF81EC00957  
00007FF81EC00958  
00007FF81EC00959  
00007FF81EC0095A  
00007FF81EC0095B  
00007FF81EC0095C  
00007FF81EC0095D  
00007FF81EC0095E  
00007FF81EC0095F  
00007FF81EC00960  
00007FF81EC00965  
00007FF81EC0096A  
00007FF81EC0096B  
00007FF81EC0096C  
00007FF81EC0096E  
00007FF81EC00976  
00007FF81EC0097D  
00007FF81EC00984  
00007FF81EC00987  
00007FF81EC0098E  
00007FF81EC00995  
00007FF81EC0099C  
00007FF81EC0099E  
00007FF81EC009A3  
00007FF81EC009A8  
00007FF81EC009B0  
00007FF81EC009B5  
00007FF81EC009B8  
00007FF81EC009C0  
00007FF81EC009C6  
00007FF81EC009CB  
00007FF81EC009CE  
00007FF81EC009D0  
00007FF81EC009D7  
00007FF81EC009DA  
00007FF81EC009DC  
00007FF81EC009DF

48:895C  
48:8974  
55  
57  
41:56  
48:8DAC  
48:81EC  
48:8B05  
48:33C4  
48:8985  
4C:8B05  
48:8D05  
33FF  
48:8942 50  
C744 24 48 16001800  
48:8044 24 70  
48:8944 24 68  
48:8BF1  
C744 24 60 00000001  
41:BE 00010000  
66:897C 24 70  
4D:85C0  
74 28  
8B14 25 3003FE7F  
8D4F 40  
8BC2  
83E0 3F  
2BC8

mov qword ptr ss:[rsp+50],  
mov dword ptr ss:[rsp+48],  
lea rax,qword ptr ss:[rsp+68],  
mov rsi,rcx  
mov dword ptr ss:[rsp+60],  
mov r14d,100  
mov word ptr ss:[rsp+70],  
test r8,r8  
je ntdll.7FF81EC009FB  
mov edx,dword ptr ds:[7FF81EC009FB],  
lea ecx,qword ptr ds:[rdi+100],  
mov eax,edx  
and eax,3F  
sub ecx,eax

RAX 0000000000000000  
RBX 00007FF81EC55A10  
RCX 00007FF81EBCD564  
RDX 0000000000000000  
RBP 0000000000000000

Hide FPU

LastError 00000002 (ERROR\_FILE\_NOT\_FOUND)  
LastStatus C0000034 (STATUS\_OBJECT\_NAME\_NOT\_FOUND)

GS 0028 FS 0053  
ES 0028 DS 0028

Default (x64 fastcall) 5 Unlocked

1: rcx 00007FF81EBCD564 ntdll.00007FF81EBCD564  
2: rdx 0000000000000000 0000000000000000  
3: r8 00000096D13FEDB8 00000096D13FEDB8  
4: r9 0000000000000000 0000000000000000  
5: [rsp+28] 0000000000000040 0000000000000000

ntdll.00007FF81EC00953

.text:00007FF81EC00951 ntdll.dll:\$D0951 #CFD51

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5

Address	Hex	ASCII
00007FF81EB31000	CC CC CC CC	iiiiii
00007FF81EB31010	48 89 5C 24	10 48 89 74
00007FF81EB31020	81 EC 80 00	00 00 48 88
00007FF81EB31030	48 89 44 24	70 4D 8B F8
00007FF81EB31040	0F 84 E1 65	0A 0F 85 95
00007FF81EB31050	34 C9 45 33	D2 4C 8D 74
00007FF81EB31060	04 5B 12 00	45 85 C9 0F
00007FF81EB31070	33 D2 49 F7	0F 49 FF CE
00007FF81EB31080	0E 48 85 C0	75 EA 48 80
00007FF81EB31090	0F 88 BE 65	0A 0F 88 BE
00007FF81EB310A0	C6 49 8B DE	49 8B DE F8
00007FF81EB310B0	42 C6 04 3E	00 EB 02 33
00007FF81EB310C0	48 33 CC E8	78 82 08 0C
00007FF81EB310D0	49 8B 5B 28	49 8B 5B 28
00007FF81EB310E0	C3 CC CC CC	CC CC CC CC

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused System breakpoint reached!

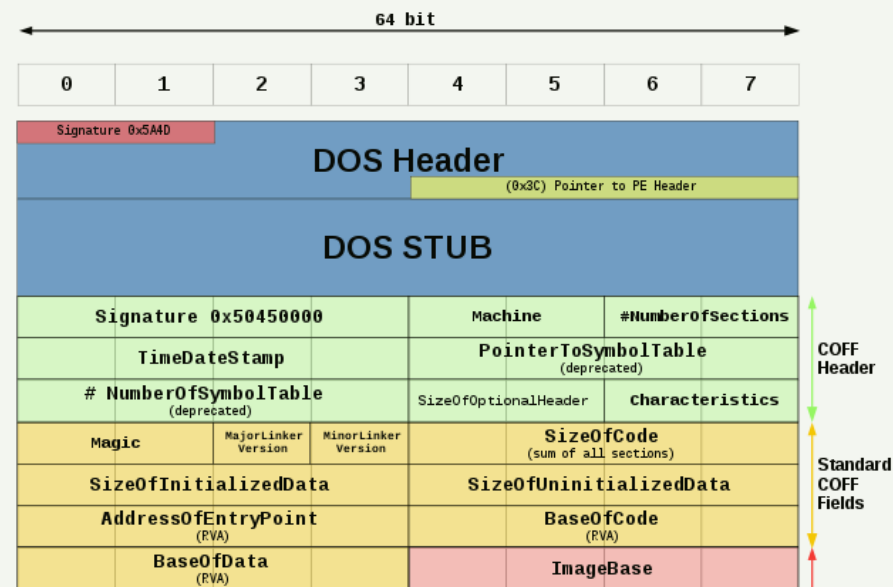
Time Wasted Debugging: 0:00:34:15

Este un punct de intrare temporar, gestionat de sistemul de operare (Windows loader), în modulul ntdll.dll

00007FF7A58C0000 este adresa virtuală la care fișierul este încărcat în memorie.

Address	Size	Party	Info	Content
00007FF568450000	00000000000001000	User		
00007FF568460000	00000000000001000	User		
00007FF568470000	00000000000023000	User		
00007FF7A58C0000	00000000000001000	User	minimal.cpp (messagebox).exe	
00007FF7A58C1000	00000000000001000	User	".text"	
00007FF7A58C2000	00000000000001000	User	".rdata"	
00007FF7A58C3000	00000000000001000	User	".pdata"	
00007FF7A58C4000	00000000000001000	User	".xdata"	
00007FF7A58C5000	00000000000001000	User	".idata"	
00007FF7A58C6000	00000000000001000	User	".bss"	
00007FF7A58C7000	00000000000001000	User	".comment"	
00007FF7A58C8000	00000000000001000	User	".reloc"	
00007FF7A58C9000	00000000000001000	User	".section names"	
00007FF7A58CA000	00000000000001000	User	".section names"	
00007FF7A58CB000	00000000000001000	User	".section names"	
00007FF7A58CC000	00000000000001000	User	".section names"	
00007FF7A58CD000	00000000000001000	User	".section names"	
00007FF7A58CE000	00000000000001000	User	".section names"	
00007FF7A58CF000	00000000000001000	User	".section names"	
00007FF7A58D0000	00000000000001000	User	".section names"	
00007FF7A58D1000	00000000000001000	User	".section names"	
00007FF7A58D2000	00000000000001000	User	".section names"	
00007FF7A58D3000	00000000000001000	User	".section names"	
00007FF7A58D4000	00000000000001000	User	".section names"	
00007FF7A58D5000	00000000000001000	User	".section names"	
00007FF7A58D6000	00000000000001000	User	".section names"	
00007FF7A58D7000	00000000000001000	User	".section names"	
00007FF7A58D8000	00000000000001000	User	".section names"	
00007FF7A58D9000	00000000000001000	User	".section names"	
00007FF7A58DA000	00000000000001000	User	".section names"	
00007FF7A58DB000	00000000000001000	User	".section names"	
00007FF7A58DC000	00000000000001000	User	".section names"	
00007FF7A58DD000	00000000000001000	User	".section names"	
00007FF7A58DE000	00000000000001000	User	".section names"	
00007FF7A58DF000	00000000000001000	User	".section names"	
00007FF7A58E0000	00000000000001000	User	".section names"	
00007FF7A58E1000	00000000000001000	User	".section names"	
00007FF7A58E2000	00000000000001000	User	".section names"	
00007FF7A58E3000	00000000000001000	User	".section names"	
00007FF7A58E4000	00000000000001000	User	".section names"	
00007FF7A58E5000	00000000000001000	User	".section names"	
00007FF7A58E6000	00000000000001000	User	".section names"	
00007FF7A58E7000	00000000000001000	User	".section names"	
00007FF7A58E8000	00000000000001000	User	".section names"	
00007FF7A58E9000	00000000000001000	User	".section names"	
00007FF7A58EA000	00000000000001000	User	".section names"	
00007FF7A58EB000	00000000000001000	User	".section names"	
00007FF7A58EC000	00000000000001000	User	".section names"	
00007FF7A58ED000	00000000000001000	User	".section names"	
00007FF7A58EE000	00000000000001000	User	".section names"	
00007FF7A58EF000	00000000000001000	User	".section names"	
00007FF7A58F0000	00000000000001000	User	".section names"	
00007FF7A58F1000	00000000000001000	User	".section names"	
00007FF7A58F2000	00000000000001000	User	".section names"	
00007FF7A58F3000	00000000000001000	User	".section names"	
00007FF7A58F4000	00000000000001000	User	".section names"	
00007FF7A58F5000	00000000000001000	User	".section names"	
00007FF7A58F6000	00000000000001000	User	".section names"	
00007FF7A58F7000	00000000000001000	User	".section names"	
00007FF7A58F8000	00000000000001000	User	".section names"	
00007FF7A58F9000	00000000000001000	User	".section names"	
00007FF7A58FA000	00000000000001000	User	".section names"	
00007FF7A58FB000	00000000000001000	User	".section names"	
00007FF7A58FC000	00000000000001000	User	".section names"	
00007FF7A58FD000	00000000000001000	User	".section names"	
00007FF7A58FE000	00000000000001000	User	".section names"	
00007FF7A58FF000	00000000000001000	User	".section names"	

- (1) Executarea codului în debugger la o adresă virtuală din modulul ntdll.dll
- (2) Selectarea fișierului messageboxa.exe din Memory Map
- (3) Imagine completă a structurii antetului DOS + PE + Optional Header pentru arhitectură 64-bit



Atentie: Dacă alegeți .text → Follow in Dump, ajungeti la începutul secțiunii .text, nu la adresa exactă a Entry Point-ului.

The screenshot shows the Immunity Debugger interface with the memory map of `minimal.cpp (MessageBoxA).exe`. The memory map lists various memory segments, including `KUSER_SHARED_DATA`, `User`, `Reserved`, `PEB, TEB`, `Stack`, and `Heap`. Two callouts labeled `imageBase (baza imaginii)` point to the `imageBase` field in the memory map. A red arrow labeled `1` points to the `minimal.cpp (messageboxa).exe` entry. Another red arrow labeled `2` points to the `Copy` button in the context menu.

The screenshot shows the Immunity Debugger interface with the CPU window active. The CPU window displays a list of memory addresses, sizes, parties, and contents. A red arrow points to the address 00007F7A58C0000, which is highlighted. A right-click context menu is open over the selected address, showing various options. The 'Follow in Dump' option is highlighted, and a red arrow points to it.

Address	Size	Party	Info	Content
00007FF568450000	00000000000001000	User		
00007FF568460000	00000000000001000	User		
00007FF568470000	00000000000023000	User		
00007FF7A58C0000	00000000000001000	User	minimal_cpp (messagebo...	
00007FF7A58C1000	00000000000001000	User	".text"	
00007FF7A58C2000	00000000000001000	User	".rdata"	
00007FF7A58C3000	00000000000001000	User	".pdata"	
00007FF7A58C4000	00000000000001000	User	".xdata"	
00007FF7A58C5000	00000000000001000	User	".idata"	
00007FF8193F0000	00000000000001000	System	apple1p.d11	
00007FF8193F1000	0000000000004F000	System	".text"	
00007FF819440000	00000000000022000	System	".rdata"	
00007FF819462000	00000000000003000	System	".pdata"	
00007FF819465000	00000000000004000	System	".rsrc"	
00007FF819469000	00000000000017000	System	".reloc"	
00007FF819480000	00000000000001000	System	ucrtbase.d11	
00007FF81C220000	00000000000001000	System	".text"	
00007FF81C221000	00000000000084000	System		

Context Menu Options:

- Follow in Disassembler
- Follow in Dump
- Follow in Symbols
- Dump Memory to File
- Comment
- Find Pattern...
- Region view
- Find references to region

Dump 1				Dump 2				Dump 3				Dump 4				Dump 5				Watch 1			
Address		Hex		Hex		Hex		Hex		Hex		Hex		Hex		Hex		ASCII					
00007FF7A58C0000		4D 5A 90 00 03 00 00 00		04 00 00 00 FF FF 00 00		MZ.....yy																	
00007FF7A58C0010		38 00 00 00 00 00 00 00		40 00 00 00 00 00 00 00		.....@																	
00007FF7A58C0020		00 00 00 00 00 00 00 00		00 00 00 00 00 00 00 00		.....																	
00007FF7A58C0030		00 00 00 00 00 00 00 00		00 00 00 00 80 00 00 00		.....																	
00007FF7A58C0040		0E 1F BA 0E 00 B4 09 CD		21 B8 01 4C CD 21 54 68		...°.i!.Li!Th																	
00007FF7A58C0050		39 73 20 70 72 6F 67 72		61 6D 20 63 61 6E 6E 6F		is program canno																	
00007FF7A58C0060		4 20 62 65 20 72 75 6E		20 69 6E 20 44 4F 53 20		t be run in DOS																	
00007FF7A58C0070		6D 6F 64 65 2E 0D 0D 0A		24 00 00 00 00 00 00 00		mode....\$. ....																	
00007FF7A58C0080		50 45 00 00 64 86 05 00		49 41 F3 65 00 00 00 00		PE. d...IAoe. ....																	
00007FF7A58C0090		00 00 00 00 F0 00 2E 02		0B 02 02 24 00 02 00 00		.δ.....\$. ....																	
00007FF7A58C00A0		00 08 00 00 00 00 00 00		00 10 00 00 00 10 00 00		..... ..																	
00007FF7A58C00B0		00 00 8C A5 F7 7F 00 00		00 10 00 00 00 02 00 00		...¥÷.....																	
00007FF7A58C00C0		04 00 00 00 00 00 00 00		05 00 02 00 00 00 00 00		.....																	
00007FF7A58C00D0		00 60 00 00 00 04 00 00		86 06 01 00 02 00 60 01		.....																	
00007FF7A58C00E0		00 00 20 00 00 00 00 00		00 10 00 00 00 00 00 00		.....																	
00007FF7A58C00F0		00 00 10 00 00 00 00 00		00 10 00 00 00 00 00 00		.....																	

Rezultatul adăugării 00007FF7A58C0000 (baza imaginii) la pointerul 3C în hexazecimal este 7FF7A58C003C.  
De ce “3C”?

Offset-ul 0x3C este locația standard din antetul unui fișier PE unde se găsește pointerul către antetul PE (PE Header)

Rezultatul adăugării 00007FF7A58C0000 (baza imaginii) la pointerul 3C în hexazecimal este 7FF7A58C003C.  
De ce “3C”?

Offset-ul 0x3C este locația standard din antetul unui fișier PE unde se găsește pointerul către antetul PE (PE Header)

imageBase = 00007FF7A58C0000 → adresa de încărcare în memorie a fișierului executabil (EXE sau DLL) de către sistemul de operare Windows.

# Pe scurt ... structura de început a unui fișier PE

- 0x00 - începe cu semnătura MZ = 4D 5A (2 bytes)
- Urmează câteva câmpuri specifice antetului DOS (nefolosite de Windows în general)
- 0x3C - la acest offset fix (60 zecimal), se află un DWORD (4 bytes) care indică offsetul unde începe antetul PE (adică locul unde se găsește semnătura PE\0\0)

## Deci fluxul este:

Nota: Antetul DOS (MS-DOS Header) începe cu 4D 5A, adică semnătura "MZ" (inițialele lui Mark Zbikowski, unul dintre arhitecții MS-DOS)

Windows încarcă fișierul în RAM:

- Citește primii 64 de bytes (antetul DOS).
- Se duce la offset 0x3C, citește un DWORD → peHeaderOffset.
- Sare la imageBase + peHeaderOffset.
- Acolo începe analiza structurii PE (importuri, secțiuni, etc.).



## Calculul adresei antetului PE pe baza valorii din offset-ul 0x3C

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1
Address	Hex	Hex	Hex	Hex	ASCII
00007FF7A58C0000	4D 5A 90 00	03 00 00 00	04 00 00 00	FF FF 00 00	MZ.....yy..
00007FF7A58C0010	00 00 00 00	00 00 00 00	40 00 00 00	00 00 00 00	.....@.....
00007FF7A58C0020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
00007FF7A58C0030	00 00 00 00	00 00 00 00	00 00 00 00	80 00 00 00	.....
00007FF7A58C0040	21 B8 01 4C	CD 21 54 68	..	..	..!..Li!Th
00007FF7A58C0050	61 6D 20 63	61 6E 6E 6F	is	program	canno
00007FF7A58C0060	20 69 6E 20	44 4F 53 20	t	be run in	DOS
00007FF7A58C0070	24 00 00 00	00 00 00 00	mode....\$.		
00007FF7A58C0080	49 41 F3 65	00 00 00 00	PE..d..IAoe		
00007FF7A58C0090	0B 02 02 24	00 02 00 00	..d....\$		
00007FF7A58C00A0	00 10 00 00	00 10 00 00	..		
00007FF7A58C00B0	00 10 00 00	00 02 00 00	..		
00007FF7A58C00C0	05 00 02 00	00 00 00 00	..		
00007FF7A58C00D0	86 06 01 00	02 00 60 01	..		
00007FF7A58C00E0	00 10 00 00	00 00 00 00	..		
00007FF7A58C00F0	00 10 00 00	00 00 00 00	..		

imageBase  
(baza imaginii)

Calculator

Expression: 00007FF7A58C0000 + 3C

Hexadecimal: 7FF7A58C003C

Signed: 140701611065404

Unsigned: 140701611065404

Octal: 3777364543000074

Bytes: 3C008CASF77F0000

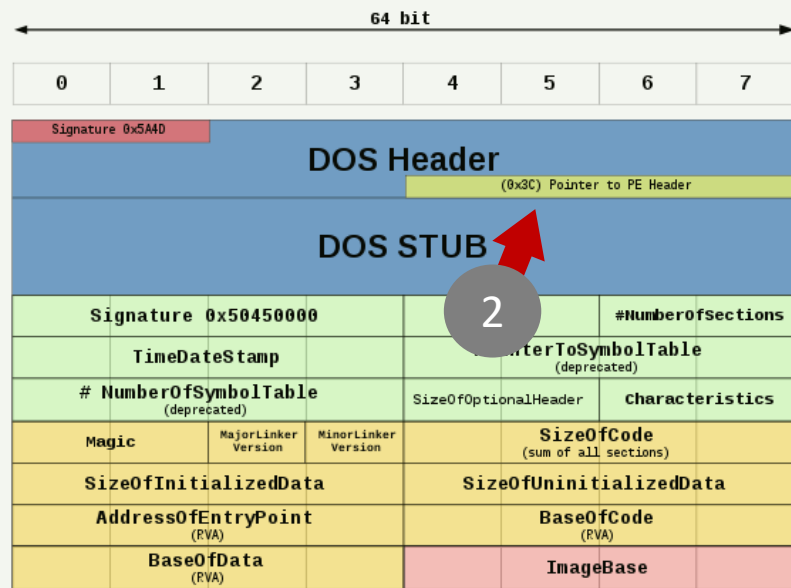
Binary: 00 0000 0111 1111 1111 01 1000 1100 0000 0000 0011 1100

ASCII: ?

Unicode: ?

Follow in Disassembler Follow in Dump Follow in Memory Map Close

```
# Hexadecimal
hex_value = "00007FF7A58C0000"
hex_add = "3C"
# Convertire la decimal
dec_value = int(hex_value, 16)
dec_add = int(hex_add, 16)
# sumare
result_dec = dec_value + dec_add
# reconverite la hexadecimal
result_hex = hex(result_dec)
```



Se adăugă 00007FF7A58C0000 (baza imaginii) la pointerul 3C în hexazecimal (imageBase + 3C): 7FF7A58C003C.

Se adăuga 00007FF7A58C0000 la valoarea 80 în hexazecimal si obtinem 7FF7A58C0080.

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1
Address	Hex	Hex	Hex	Hex	ASCII
00007FF7A58C003C	80 00 00 00	0E 1F BA 0E	00 B4 09 CD	21 B8 01 4C	.....f!..L
00007FF7A58C004C	CD 21 54 68	69 73 20 70	72 6F 67 72	61 6D 20 63	!This program c
00007FF7A58C005C	61 6E 6E 6F	74 20 62 65	20 72 75 6E	20 69 6E 20	annot be run in
00007FF7A58C006C	44 4F 53 20	6D 6F 64 65	2E 0D 0D 0A	24 00 00 00	DOS mode....\$.
00007FF7A58C007C	00 00 00 00	50 45 00 00	64 86 05 00	49 41 F3 65	..PE..d..IAoe
00007FF7A58C008C	00 02 00 00	00 00 00 00	F0 00 2E 02	0B 02 02 24	..d....\$
00007FF7A58C009C	00 10 00 00	00 08 00 00	00 00 00 00	00 10 00 00	..
00007FF7A58C00AC	00 02 00 00	00 04 00 00	00 00 00 00	00 00 00 00	..
00007FF7A58C00BC	00 00 00 00	00 60 00 00	00 04 00 00	00 00 00 00	..
00007FF7A58C00CC	00 60 01 00	00 20 00 00	00 00 00 00	00 10 00 00	..
00007FF7A58C00DC	00 00 00 00	00 10 00 00	00 00 00 00	00 10 00 00	..
00007FF7A58C00EC	00 00 00 00	00 00 00 00	00 10 00 00	00 00 00 00	..
00007FF7A58C00FC	00 00 00 00	00 50 00 00	00 68 00 00	00 00 00 00	..P..h
00007FF7A58C010C	00 00 00 00	00 30 00 00	00 0C 00 00	00 00 00 00	..0..

imageBase + 3C  
(7FF7A58C003C)

PE\0\0

Se folosește valoarea de la acest offset (0x80) pentru a localiza efectiv antetul PE în memorie.

se extrage valoarea-pointer (0x80) care duce la semnătura PE\0\0

# X64DBG: LITTLE-ENDIAN

La adresa 7FF7A58C0080 ne așteptăm să găsim „PE 00 00 ..”  
La 7FF7A58C0080 + 28, adică la 00007FF7A58C00A8 ne așteptăm la o valoare de 4 octeți și anume 00 01 00 00. Ce facem cu această valoare?

**Calculator**

Expression: 00007FF7A58C0080 + 28

Hexadecimal: 7FF7A58C00A8

Signed: 14070167065512

Unsigned: 14070167065512

Octal: 37773000250

Bytes: 8008CA3F77F0000

Binary: 00 0000 0111 1111 1111 1111 1010 0101 1000 1100 0000 0000 1000 1000

ASCII: ?

Unicode: ?

Follow in Disassembler Follow in Dump Follow in Memory Map Close

**Dump 1** **Dump 2** **Dump 3** **Dump 4** **Dump 5** **Watch 1**

Address	Hex	ASCII
00007FF7A58C00A8	00 10 00 00	...
00007FF7A58C00B8	00 10 00 00	...
00007FF7A58C00C8	05 02 00 00	...
00007FF7A58C00D8	86 01 00 00	...
00007FF7A58C00E8	00 00 00 00	...
00007FF7A58C00F8	00 00 00 00	...
00007FF7A58C0108	00 00 00 00	...
00007FF7A58C0118	00 00 00 00	...
00007FF7A58C0128	00 00 00 00	...
00007FF7A58C0138	00 00 00 00	...
00007FF7A58C0148	00 00 00 00	...
00007FF7A58C0158	00 00 00 00	...
00007FF7A58C0168	38 50 00 00	8P...
00007FF7A58C0178	00 00 00 00	...
00007FF7A58C0188	2E 74 65 78	.text
00007FF7A58C0198	00 02 00 00	...

**64 bit**

0	1	2	3	4	5	6	7
Signature 0x5A4D							
DOS Header							
(0x3C) Pointer to PE Header							
DOS STUB							
Signature 0x50450000				#NumberOfSections			
TimeDateStamp				PointerToSymbolTable (deprecated)			
# NumberOfSymbolTable (deprecated)				SizeOfOptionalHeader characteristics			
Magic		MajorLinker Version		MinorLinker Version		SizeOfCode (sum of all sections)	
SizeOfInitializedData				SizeOfUninitializedData			
AddressOfEntryPoint (RVA)				BaseOfCode (RVA)			
BaseOfData (RVA)				ImageBase			

**COFF Header**

**Standard COFF Fields**

Byte Index: 0 1

Big-Endian: 13 D4

Little-Endian: D4 13

Little-Endian

**X32DBG:**  
PUNCTUL DE INTRARE

$\text{imageBase} + 0x80$  (PE Header offset) +  $0x28$  (offset pentru AddressOfEntryPoint)

00 01 00 00 → se interpretează ca 0x00000100

**X32DBG:**  
PUNCTUL DE INTRARE

$\text{imageBase} + 0x80$  (PE Header offset) +  $0x28$  (offset pentru AddressOfEntryPoint)

00 01 00 00 → se interpretează ca 0x00000100

Calculator

Expression: 00007FF7A58C0000 + 00001000

Hexadecimal: 7FF7A58C1000

Signed: 1407016969440

Unsigned: 1407016969440

Octal: 377010000

Bytes: 00108CA5F77F0000

Binary: 0111 1111 1111 0111 1010 0101 1000 1100 0001 0000 0000 0000

ASCII:

Unicode:

Follow in Disassembler Follow in Dump Follow in Memory Map Close

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1

Address	Hex	ASCII
00007FF7A58C00A8	00 10 00 00	...
00007FF7A58C00B8	00 10 00 00	...
00007FF7A58C00C8	05 00 02 00	...
00007FF7A58C00D8	86 00 01 00	...
00007FF7A58C00E8	00 00 00 00	...
00007FF7A58C00F8	00 00 00 00	...
00007FF7A58C0108	00 00 00 00	...
00007FF7A58C0118	00 00 00 00	...
00007FF7A58C0128	00 00 00 00	...
00007FF7A58C0138	00 00 00 00	...
00007FF7A58C0148	00 00 00 00	...
00007FF7A58C0158	00 00 00 00	...
00007FF7A58C0168	38 50 00 00	8P...
00007FF7A58C0178	00 00 00 00	...
00007FF7A58C0188	2E 74 65 78	.text...
00007FF7A58C0198	00 02 00 00	...

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols

Address	Disassembly	Comment
00007FF7A58C1000	sub rsp,28	OptionalHeader.AddressOfEntryPoint
00007FF7A58C1004	xor r9d,r9d	
00007FF7A58C1007	lea r8,qword ptr ds:[7FF7A58C2000]	00007FF7A58C2000:"Minimal C++ Applic
00007FF7A58C100E	xor ecx,ecx	
00007FF7A58C1010	lea rdx,qword ptr ds:[7FF7A58C2018]	00007FF7A58C2018:"Hello, world!"
00007FF7A58C1017	call qword ptr ds:[<MessageBoxA>]	
00007FF7A58C101D	xor eax,eax	
00007FF7A58C101F	add rsp,28	
00007FF7A58C1023	ret	
00007FF7A58C1024	nop	
00007FF7A58C1025	nop	
00007FF7A58C1026	nop	
00007FF7A58C1027	nop	
00007FF7A58C1028	nop	
00007FF7A58C1029	nop	
00007FF7A58C102A	nop	
00007FF7A58C102B	nop	
00007FF7A58C102C	nop	
00007FF7A58C102D	nop	
00007FF7A58C102E	nop	
00007FF7A58C102F	nop	
00007FF7A58C1030	jmp qword ptr ds:[<MessageBoxA>]	JMP.<MessageBoxA>

Adresa punctului de intrare (Entry Point)

imageBase + AddressOfEntryPoint  
= 00007FF7A58C0000 + 0x100  
= 00007FF7A58C0100

Adresa punctului de intrare (Entry Point)

```
imageBase + AddressOfEntryPoint
= 00007FF7A58C0000 + 0x100
= 00007FF7A58C0100
```

```
def gaseste_punct_intrare_pe(calegere_fisier_pe):
    with open(calegere_fisier_pe, 'rb') as f:
        # Citim offsetul la header-ul PE din header-ul MZ
        f.seek(0x3C)
        offset_pe = int.from_bytes(f.read(4), 'little')

        # Ne mutăm la începutul header-ului PE folosind offsetul găsit
        f.seek(offset_pe)

        # Verificăm dacă avem semnătura PE corectă
        if f.read(4) != b'PE\0\0':
            raise ValueError("Fișierul nu este un executabil PE valid.")

        # Sărim peste COFF File Header pentru a ajunge la Optional Header
        f.seek(20, 1) # Header-ul COFF are mereu 20 de bytes

        # Citim adresa relativă a punctului de intrare din Optional Header
        # Această adresă este la offsetul 16 în Optional Header pentru
        # executabilele PE32 și la offsetul 24 pentru PE32+
        # Aici presupunem că este un executabil PE32 standard
        f.seek(16, 1)
        adresa_rva_punct_intrare = int.from_bytes(f.read(4), 'little')
        return adresa_rva_punct_intrare

cale_fisier_pe = 'C:\\Users\\Elitebook\\Desktop\\LaboratorATMCript.exe'

try:
    punct_intrare = gaseste_punct_intrare_pe(cale_fisier_pe)
    print(f"Punctul de intrare se găsește la adresa: {punct_intrare}")

    # Determinăm numărul de bytes necesari pentru a reprezenta adresa;
    # 4 bytes (32 de biți) sunt suficienți pentru majoritatea adreselor RVA
    bytes_little_endian = punct_intrare.to_bytes(4, byteorder='little')
    bytes_big_endian = punct_intrare.to_bytes(4, byteorder='big')

    print(f"Little-endian: {bytes_little_endian.hex(' ')}")
    print(f"Big-endian : {bytes_big_endian.hex(' ')}")
except Exception as e:
    print(e)
```

## Cum calculăm codul punctului de intrare?

Punctul de intrare se găsește la adresa: **4648**

Little-endian: **28 12 00 00**

Big-endian : **00 00 12 28**

- Într-un fișier executabil în formatul Portable Executable (PE) folosit pe sistemele Windows, header-ul COFF (Common Object File Format) este urmat imediat de header-ul PE Optional. Acesta din urmă conține o serie de câmpuri care sunt esențiale pentru încărcarea și execuția executabilului, inclusiv adresa punctului de intrare.
- După ce citești header-ul PE și verifici semnătura acestuia ("PE\0\0"), următorul pas este să sari peste COFF File Header pentru a ajunge la Optional Header. COFF File Header are o lungime fixă de 20 de bytes și conține informații despre structura de bază a fișierului, cum ar fi numărul de secțiuni, timestamp-urile, simbolurile etc.
- Pentru a ajunge la Optional Header, trebuie să te deplasezi peste acești 20 de bytes imediat după verificarea semnăturii PE. Acesta este motivul pentru care în codul anterior se face `f.seek(20, 1)` după ce se citește și se verifică semnătura PE. Această operație mută cursorul fișierului cu 20 de bytes înainte, peste COFF File Header, poziționându-l la începutul Optional Header.
- Optional Header începe imediat după COFF File Header și conține informații suplimentare necesare pentru executabil, cum ar fi versiunea de linker, dimensiunea secțiunilor de cod și date, adresele tabelului de import și export, și, cel mai important pentru această discuție, adresa punctului de intrare al executabilului. Adresa punctului de intrare este relativă la baza imaginii în memorie și este utilizată de sistemul de operare pentru a ști de unde să înceapă execuția codului atunci când fișierul este încărcat.
- Practic, pentru a citi adresa punctului de intrare din Optional Header, te deplasezi cu cursorul la poziția corespunzătoare (de exemplu, la offsetul 16 în Optional Header pentru executabile PE32) și citești valoarea respectivă, așa cum se face în secvența de cod din întrebarea ta.
- RVA se referă la o adresă care este calculată în raport cu o bază de încărcare atunci când un program este încărcat în memorie. Când un fișier executabil este încărcat în memorie de către sistemul de operare, acesta primește o adresă de bază la care va fi încărcat. Această adresă de bază poate varia de la o rulare la alta, în funcție de diverse aspecte, cum ar fi ASLR (Address Space Layout Randomization), care este o tehnică de securitate menită să prevină anumite tipuri de atacuri. O Adresă Virtuală Relativă (RVA) este deci o adresă specificată relativ la adresa de bază a modulului care este încărcat în memorie. De exemplu, dacă un modul este încărcat la adresa de bază `0x10000000` și o funcție din cadrul acestui modul se află la RVA `0x00001000`, adresa absolută a funcției în spațiul de memorie al procesului va fi `0x10001000`.



```
def gaseste_si_arata_bytes_punct_intrare(calegere_fisier_pe):
    with open(calegere_fisier_pe, 'rb') as f:
        # Citim offsetul la header-ul PE din header-ul MZ
        f.seek(0x3C)
        offset_pe = int.from_bytes(f.read(4), 'little')

        # Ne mutăm la începutul header-ului PE folosind offsetul găsit
        f.seek(offset_pe)

        # Verificăm dacă avem semnătura PE corectă
        if f.read(4) != b'PE\0\0':
            raise ValueError("Fișierul nu este un executabil PE valid.")

        # Sărim peste COFF File Header pentru a ajunge la Optional Header
        f.seek(20, 1) # Header-ul COFF are mereu 20 de bytes

        # Citim adresa relativă a punctului de intrare din Optional Header
        f.seek(16, 1)
        adresa_rva_punct_intrare = int.from_bytes(f.read(4), 'little')

        # Întrucât acest exemplu nu mapează RVA la offset-ul din secțiunea specifică,
        # vom citi direct folosind RVA ca un offset simplificat pentru a arăta primii
        # 10 bytes. Acest lucru poate să nu fie precis pentru toate fișierele PE
        f.seek(adresa_rva_punct_intrare)
        primii_10_bytes = f.read(10)

        return adresa_rva_punct_intrare, primii_10_bytes.hex(' ')

cale_fisier_pe = 'C:\\Users\\Elitebook\\Desktop\\LaboratorATMCript.exe'

try:
    punct_intrare, bytes_intrare = gaseste_si_arata_bytes_punct_intrare(cale_fisier_pe)
    print(f"Punctul de intrare: {punct_intrare} (decimal), {hex(punct_intrare)} (hex)")
    print(f"Primii 10 bytes de la punctul de intrare: {bytes_intrare}")
except Exception as e:
    print(e)
```

## Primii 10 bytes de la punctul de intrare!

Punctul de intrare se găsește la adresa: **4648** (decimal), **0x1228** (hex)  
 Primii 10 bytes de la punctul de intrare: **68 1c 58 46 00 e8 ee ff ff ff**

- Pentru a afișa primii 10 bytes de la punctul de intrare, trebuie să modificăm codul pentru a include citirea și afișarea acestor bytes. Acest pas necesită determinarea adresei absolute a punctului de intrare în fișier, care implică o conversie din adresa relativă la cea virtuală (RVA) în offset-ul fizic din fișier. În simplificarea noastră, vom folosi RVA pentru a citi direct din fișier, presupunând că executabilul nu este prea complex și că RVA corespunde cu offset-ul fizic.
- În acest cod, după ce determinăm adresa RVA a punctului de intrare, folosim acea adresă pentru a sări direct la locația corespunzătoare în fișier și citim primii 10 bytes de la acea adresă. Apoi, convertim acei bytes într-un șir hexadecimale pentru afișare. Reține că acest cod presupune că RVA poate fi folosit direct ca offset în fișier, ceea ce este o simplificare și nu ar putea fi precis pentru toate fișierele PE, deoarece unele adrese RVA pot necesita conversie bazată pe layout-ul secțiunilor din fișierul PE.

# Biblioteca *pefile*

```
import pefile

def arata_bytes_punct_intrare_pe(cale_fisier_pe):
    try:
        pe = pefile.PE(cale_fisier_pe)
        adresa_ep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
        offset_ep = pe.get_offset_from_rva(adresa_ep)

        with open(cale_fisier_pe, 'rb') as f:
            f.seek(offset_ep)
            primii_10_bytes = f.read(10).hex(' ')

        print(f"Punctul de intrare: {adresa_ep} (decimal), {hex(adresa_ep)} (hex)")
        print(f"Primii 10 bytes de la punctul de intrare: {primii_10_bytes}")
    except Exception as e:
        print(e)

cale_fisier_pe = 'C:\\\\Users\\Elitebook\\Desktop\\LaboratorATMCript.exe'
arata_bytes_punct_intrare_pe(cale_fisier_pe)
```

Punctul de intrare se găsește la adresa: **4648** (decimal), **0x1228** (hex)  
Primii 10 bytes de la punctul de intrare: **68 1c 58 46 00 e8 ee ff ff ff**

- Folosind biblioteca *pefile* pentru a analiza fișierele PE (Portable Executable), putem obține adresa de intrare (entry point) și citi primii 10 bytes din acea adresă într-un mod mult mai direct și simplu. *pefile* este o bibliotecă Python care facilitează lucrul cu fișierele PE, oferind acces ușor la structurile și informațiile acestora.

Acest cod face următoarele:

- Încarcă fișierul PE folosind *pefile*.
- Obține adresa de intrare (entry point) din header-ul opțional al fișierului PE.
- Calculează offset-ul fizic în fișier pentru adresa de intrare folosind funcția **get\_offset\_from\_rva** a obiectului **pe**.
- Deschide fișierul în modul binar, sare la offset-ul calculat și citește primii 10 bytes.
- Afișează adresa de intrare și primii 10 bytes în format hexadecimale.
- Folosind *pefile*, codul devine mai concis și direct, evitând necesitatea de a naviga manual prin structura fișierului PE sau de a calcula offset-urile manual. Biblioteca gestionează toate aspectele legate de interpretarea structurii fișierului PE, permițându-ne să ne concentrăm pe informațiile specifice de care avem nevoie.

C.6.5

# REGIMUL DE COMPILARE ÎN DIFERITE LIMBAJE DE PROGRAMARE



# DE REȚINUT !

TOATE LIMBAJELE DE PROGRAMARE  
DE NIVEL ÎNALT SUNT LA FEL:

- C++
- FASM
- PYTHON
- VB6



# PACHETUL C++ CARE NE AJUTĂ SĂ COMPILĂM IMEDIAT!

## Download a TDM-GCC installer:

tdm-gcc-webdl.exe

Minimal online installer. Select the components you want, and it downloads and unpacks them. Either edition, latest release only. (GCC 10.3.0)

tdm64-gcc-10.3.0-2.exe

64+32-bit MinGW-w64 edition. Includes GCC C/C++, GNU binutils, mingw32-make, GDB (64-bit), the MinGW-w64 runtime libraries and tools, and the windows-default-manifest package.

tdm-gcc-10.3.0.exe

32-bit-only MinGW.org edition. Includes GCC C/C++, GNU binutils, mingw32-make, GDB (32-bit), the MinGW.org mingwrt and w32api packages, and the windows-default-manifest package.

1

Instalam: tdm64-gcc-10.3.0-2.exe



# REGIM DE COMPILARE OPULENT

## COMPILAREA FIȘIERULUI .PY ÎN .EXE

```
pip install pyinstaller  
pyinstaller --onefile p.py
```

```
a = 5  
b = 3  
print(a+b)
```

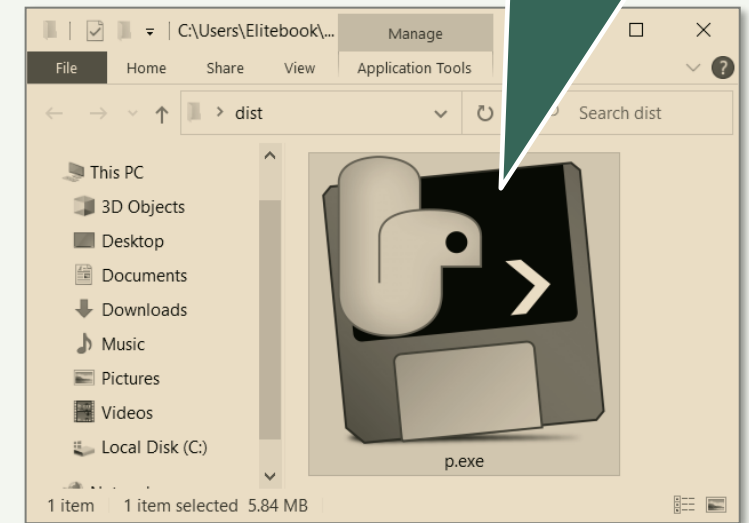
p.exe (6Mb)

```
Command Prompt  
Microsoft Windows [Version 10.0.19043.2364]  
(c) Microsoft Corporation. All rights reserved.  
C:\Users\Elitebook>cd desktop
```

```
Select Command Prompt  
C:\Users\Elitebook>cd desktop  
C:\Users\Elitebook\Desktop>pyinstaller --onefile p.py
```

```
Command Prompt  
C:\Users\Elitebook\Desktop>pyinstaller --onefile p.py  
1224 INFO: PyInstaller: 6.5.0, contrib hooks: 2024.3  
1225 INFO: Python: 3.12.2 (conda)  
1258 INFO: Platform: Windows-10-10.0.19043-SP0  
1260 INFO: wrote C:\Users\Elitebook\Desktop\p.spec  
1266 INFO: Extending PYTHONPATH with paths  
['C:\\Users\\Elitebook\\Desktop']  
1873 INFO: checking Analysis  
1894 INFO: checking PYZ  
1920 INFO: checking PKG  
1933 INFO: Bootloader C:\Users\Elitebook\miniconda3\Lib\site-packages\PyInstaller\bootloader\Windows-64bit-intel\run.exe  
1933 INFO: checking EXE  
1937 INFO: Rebuilding EXE-00.toc because p.exe missing  
1937 INFO: Building EXE from EXE-00.toc  
1937 INFO: Copying bootloader EXE to C:\Users\Elitebook\Desktop\dist\p.exe  
2031 INFO: Copying icon to EXE  
2110 INFO: Copying 0 resources to EXE  
2110 INFO: Embedding manifest in EXE  
2182 INFO: Appending PKG archive to EXE  
2198 INFO: Fixing EXE headers  
4182 INFO: Building EXE from EXE-00.toc completed successfully.  
C:\Users\Elitebook\Desktop>
```

- Deschide un terminal sau prompt de comandă.
- Navighează la directorul unde se află p.py.
- Rulează pyinstaller --onefile p.py.
- După finalizarea procesului, navighează la directorul dist care a fost creat în locația ta curentă. (Comanda cd dist)
- În dist, vei găsi p.exe.



```
Command Prompt  
4182 INFO: Building EXE from EXE-00.toc completed successfully.  
C:\Users\Elitebook\Desktop>cd dist  
C:\Users\Elitebook\Desktop\dist>p.exe  
8  
C:\Users\Elitebook\Desktop\dist>
```

1



# REGIM DE COMPILARE AUTO-SUFFICIENT

## COMPILARE (FMARE.CPP)

mare.exe (2.74Mb)

### Unde punem sursa C++?

- Cod sursă C++ minim compilat prin:
- `g++ -o fmare.exe fmare.cpp`
- Acest tip de compilare generează executabile mari care sunt destinate să funcționeze între versiunile de sistem de operare.
- Include toate bibliotecile de care un cod ar putea avea nevoie, indiferent dacă sunt folosite sau nu (devin balast).

### Deschidem CMD

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

C:\Users\Elitebook>
```

C:\Users\Elitebook\Desktop\cpp

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

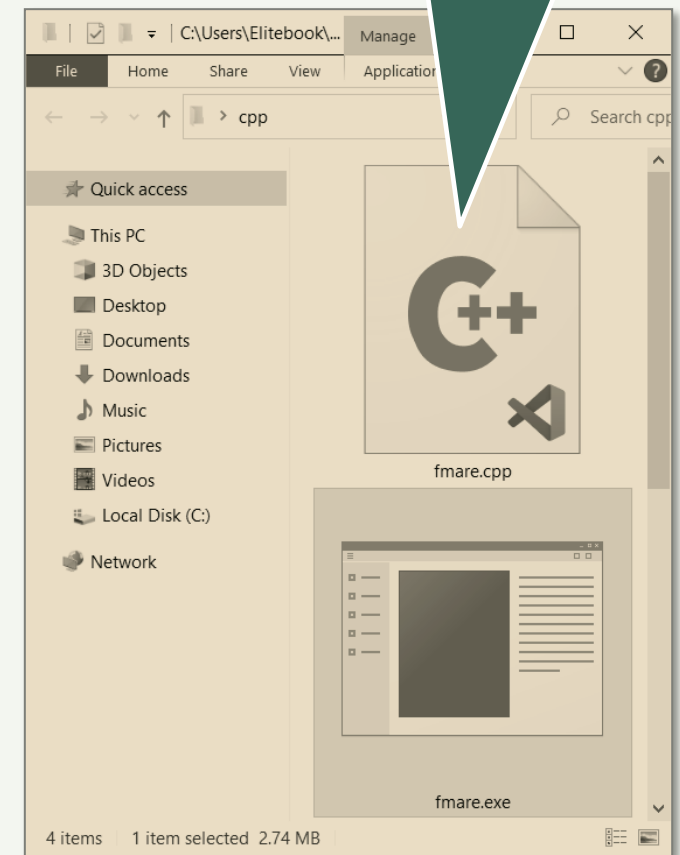
C:\Users\Elitebook>cd C:\Users\Elitebook\Desktop\cpp
C:\Users\Elitebook\Desktop\cpp>
```

```
Command Prompt

C:\Users\Elitebook\Desktop\cpp>g++ -o fmare.exe fmare.cpp
C:\Users\Elitebook\Desktop\cpp>
```

```
#include <iostream>

int main(){
    std::cout<<"Inginerie Inversa";
    return 0;
}
```



# REGIM DE COMPILARE SEMI-SUFFICIENT

## COMPILARE (MARE.CPP)

■ `g++ -Os -s -static -o mare.exe mare.cpp`

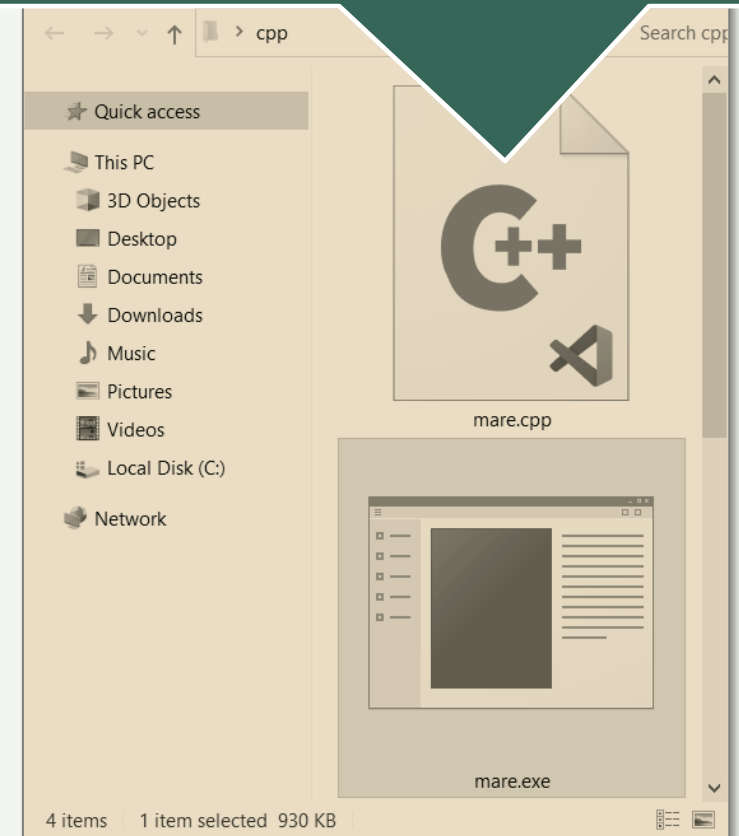
```
#include <windows.h>
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {  
    MessageBox(NULL, "Inginerie Inversa", "Cod malware", MB_OK);  
    return 0;  
}
```

mare.exe (930Kb)

```
Command Prompt  
Microsoft Windows [Version 10.0.19043.2364]  
(c) Microsoft Corporation. All rights reserved.  
  
C:\Users\Elitebook>cd C:\Users\Elitebook\Desktop\cpp  
  
C:\Users\Elitebook\Desktop\cpp>g++ -Os -s -static -o mare.exe mare.cpp  
  
C:\Users\Elitebook\Desktop\cpp>
```

```
Command Prompt  
  
C:\Users\Elitebook\Desktop\cpp>dir  
Volume in drive C has no label.  
Volume Serial Number is A06A-836B  
  
Directory of C:\Users\Elitebook\Desktop\cpp  
  
03/14/2024  08:19 PM    <DIR>          .  
03/14/2024  08:19 PM    <DIR>          ..  
03/14/2024  05:46 PM             726 mare.cpp  
03/14/2024  08:19 PM        952,320 mare.exe
```



# REGIM DE COMPILARE SEMI-DISTRIBUIT

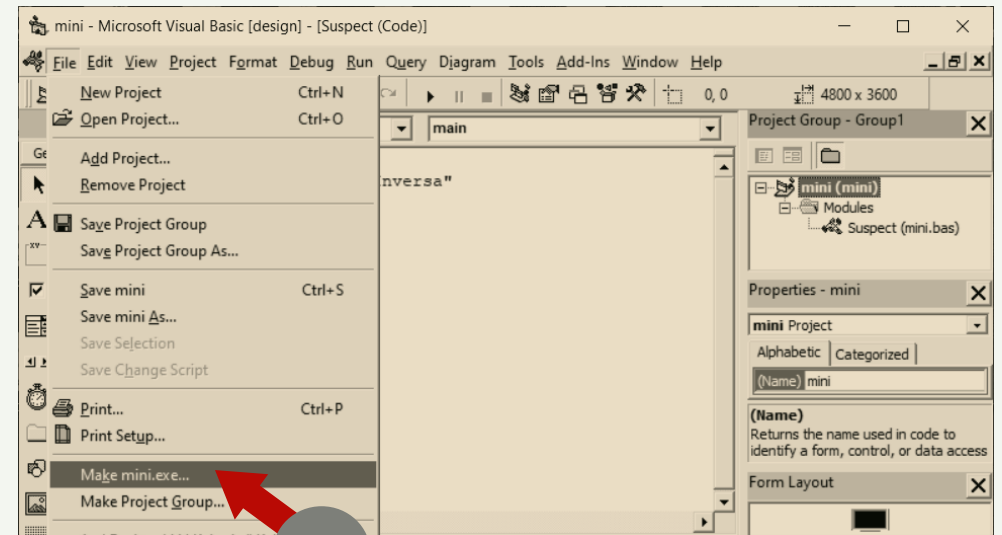
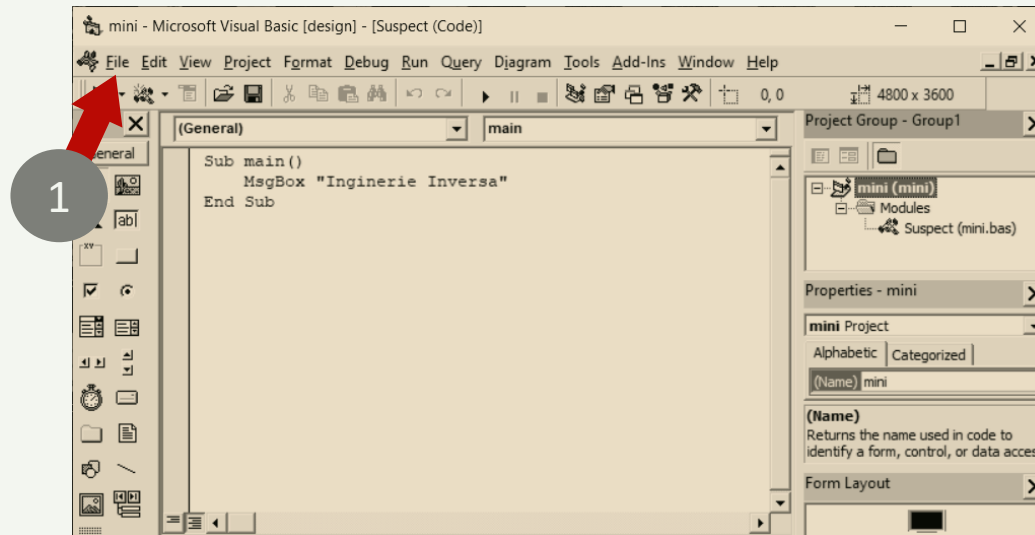
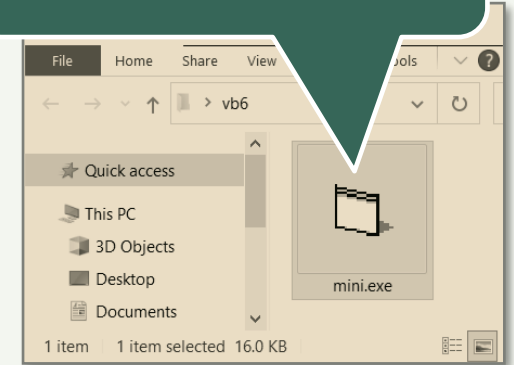
## COMPILARE (MINI.BAS)

mini.exe (16Kb)

- Cod sursă VB6 compilat în mod normal:
- Meniu [File] - [Make].
- Mic executabil de 16Kb.
- Acest tip de compilare generează executabile mici dependente de biblioteca dinamică numită msvbvm60.dll.



```
Sub main()  
  MsgBox "Inginerie Inversa"  
End Sub
```



# REGIM DE COMPILARE SUFFICIENT

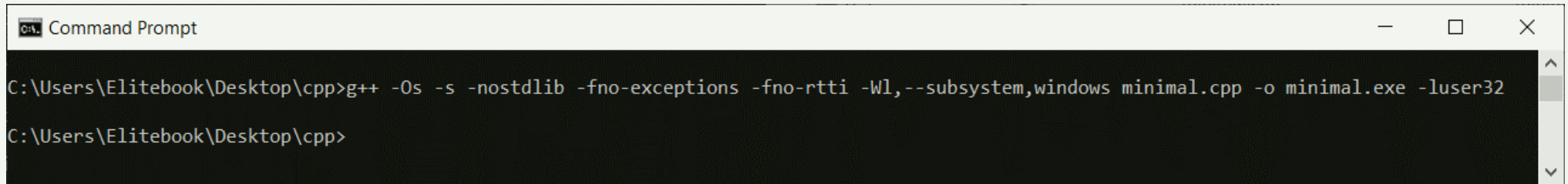
## COMPILARE (MINIMAL.CPP)

```
#include <windows.h>
```

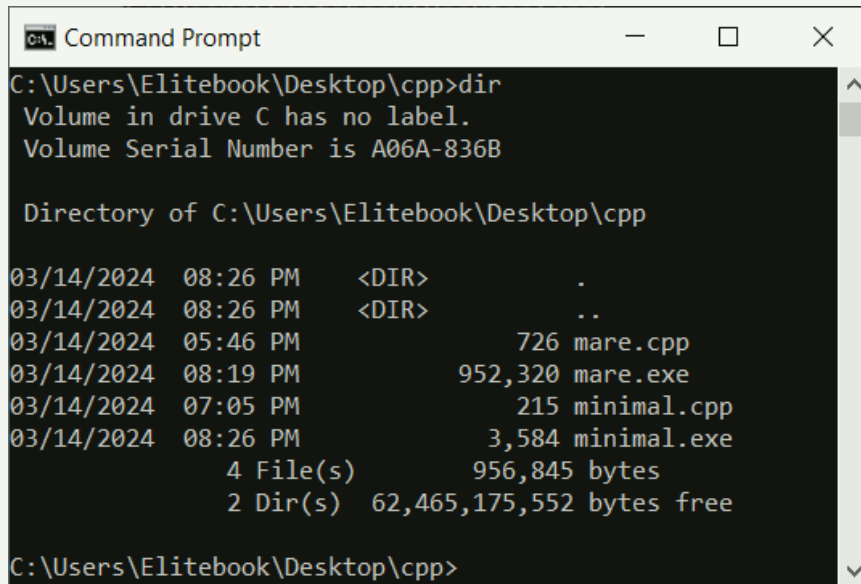
```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int  
nCmdShow) {  
    MessageBox(NULL, "Inginerie Inversa", "Cod malware", MB_OK);  
    return 0;  
}
```

minimal.exe (3.5kb)

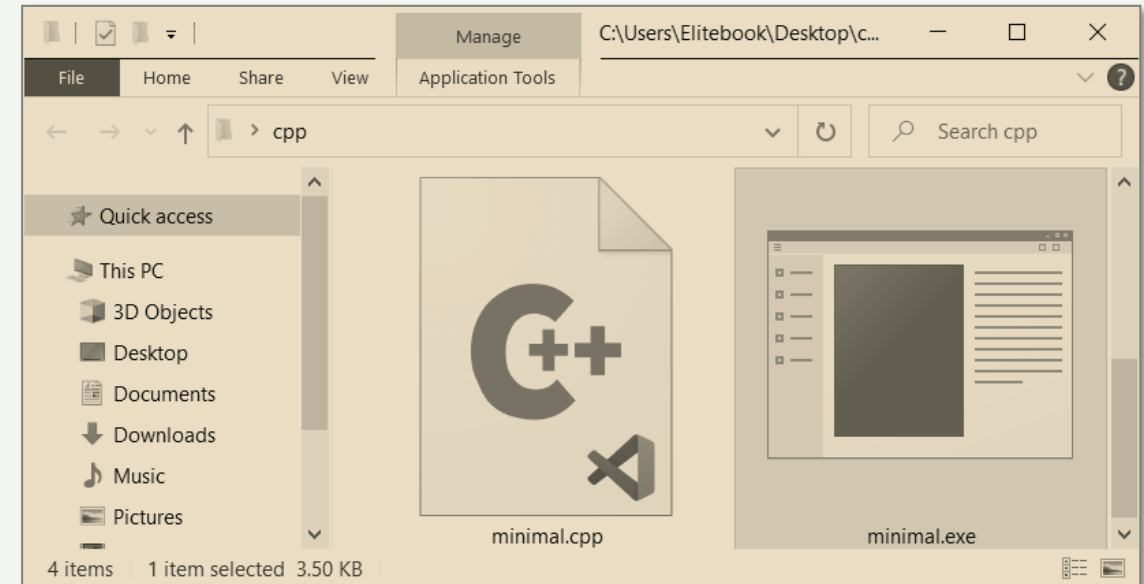
- `g++ -Os -s -nostdlib -fno-exceptions -fno-rtti -Wl,--subsystem,windows minimal.cpp -o minimal.exe -luser32`



```
Command Prompt  
C:\Users\Elitebook\Desktop\cpp>g++ -Os -s -nostdlib -fno-exceptions -fno-rtti -Wl,--subsystem,windows minimal.cpp -o minimal.exe -luser32  
C:\Users\Elitebook\Desktop\cpp>
```



```
Command Prompt  
C:\Users\Elitebook\Desktop\cpp>dir  
Volume in drive C has no label.  
Volume Serial Number is A06A-836B  
  
Directory of C:\Users\Elitebook\Desktop\cpp  
  
03/14/2024  08:26 PM    <DIR>          .  
03/14/2024  08:26 PM    <DIR>          ..  
03/14/2024  05:46 PM             726 mare.cpp  
03/14/2024  08:19 PM          952,320 mare.exe  
03/14/2024  07:05 PM             215 minimal.cpp  
03/14/2024  08:26 PM          3,584 minimal.exe  
               4 File(s)          956,845 bytes  
               2 Dir(s)  62,465,175,552 bytes free  
C:\Users\Elitebook\Desktop\cpp>
```



# REGIM DE COMPILARE OPTIM

## 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

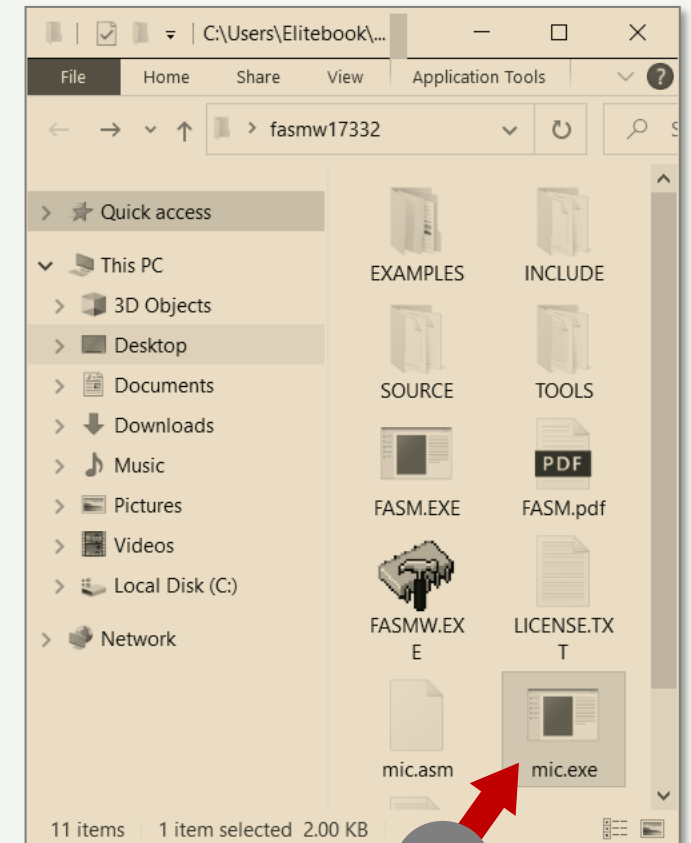
C:\Users\Elitebook\Desktop\fasmw17332>
```

```
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>
```





# Compilare in: 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>



# ARGUMENTELE PE STIVĂ

Fiecare **push** pune un argument pe stivă în ordinea corectă astfel încât, când **MessageBoxA** este apelată, aceasta să poată extrage și utiliza aceste argumente în mod corespunzător.

```
start:
    push 0           ; uType = MB_OK (deoarece 0 corespunde acestui stil)
    push title       ; lpCaption = pointer către titlul cutiei de mesaj
    push message     ; lpText = pointer către textul cutiei de mesaj
    push 0           ; hWnd = NULL (cutia de mesaj nu are fereastră părinte)
    call [MessageBoxA] ; Apelează funcția MessageBoxA cu argumentele de pe stivă
```



Instrucțiunile **push** pe care le vezi în cod sunt folosite pentru a pune argumentele pe stivă pentru funcția **MessageBoxA**. În Windows, majoritatea apelurilor de funcții către API-ul Win32 se fac prin intermediul stivei. Astfel, argumentele pentru **MessageBoxA** sunt împinse pe stivă în ordinea inversă a parametrilor, deoarece stiva este o structură de tip LIFO (Last In, First Out).

Funcția **MessageBoxA** are următorii patru parametri, listati aici în ordinea în care trebuie să fie împinși pe stivă:

1. **uType** - Stilul cutiei de mesaj (de exemplu, butoanele care vor apărea).
2. **lpCaption** - Un pointer către un șir de caractere null-terminated care reprezintă titlul cutiei de mesaj.
3. **lpText** - Un pointer către un șir de caractere null-terminated care reprezintă textul care va fi afișat în cutia de mesaj.
4. **hWnd** - Un handle către fereastra părinte a cutiei de mesaj. Dacă acesta este NULL, cutia de mesaj nu are fereastră părinte.

Valoare	Nume simbolic	Descriere
0	MB_OK	<input checked="" type="checkbox"/> un singur buton OK
1	MB_OKCANCEL	OK + Cancel
2	MB_ABORTRETRYIGNORE	Abort + Retry + Ignore
3	MB_YESNOCANCEL	Yes + No + Cancel
4	MB_YESNO	Yes + No
5	MB_RETRYCANCEL	Retry + Cancel

# DIFERENȚELE DINTRE

LIMBAJELE DE PROGRAMARE  
MODERNE ȘI CLASICE

- [Python] p.exe (6Mb)
- [C++] fmare.exe (2.74Mb)
- [C++] mare.exe (930Kb)
- [VB6.0] mini.exe (16Kb)
- [C++] minimal.exe (3.5Kb)
- [FASM] mic.exe (2Kb)



C.6.6

# ARHITECTURA CPU ȘI INSTRUCȚIUNILE DE BAZĂ



# SECȚIUNE / ROL

## CONVENTIE

- Secțiunea **.text** este pentru codul de mașină real pe care îl execută CPU.
- Secțiunea **.data** conține variabile globale sau statice care sunt inițializate de programator.
- Secțiunea **.rdata** (în PE) sau **.rodata** (în ELF) este similară cu **.data**, dar este doar pentru citire și de obicei conține constante.
- Secțiunea **.bss** deține variabile care încep neinițializate și vor fi inițializate la zero de către sistem la pornirea programului.
- Adnotările scrise de mână sugerează înțelegerea faptului că variabilele declarate în secțiunea **.bss** sunt destinate a fi atribuite mai târziu în execuția programului.
- Variabila din **.data** este inițializată imediat, iar variabila din **.rdata/.rodata** este atât inițializată, cât și marcată ca finală, ceea ce sugerează că nu poate fi modificată după atribuirea inițială, fiind astfel plasată într-o secțiune doar pentru citire.

**.text:** Cod executabil (asamblare).

**.data:** Date inițiale (acces citire/scriere).

- `int x = 98`

**.rdata/.rodata:** Date inițializate (doar acces pentru citire).

- `final int x = 98`

**.bss:** date neinițializate (acces de citire/scriere).

- `int x` (cu o nota „alocați mai târziu”)

```
format PE GUI 4.0
```

```
entry start
```

```
include 'INCLUDE/win32a.inc'
```

```
section '.data' data readable writeable
    hFile dd ?
    bytesRead dd ?
    fileName db 'C:\\test.txt', 0
    fileBuffer db 256 dup(?) ; buffer pentru continut !
    messageTitle db 'File Content', 0
    errorTitle db 'Error', 0
    errorMsg db 'The file could not be opened.', 0
    formatMessageBuffer db 256 dup(?)
```

```
section '.blabla' code readable executable
```

1

```
start:
    invoke CreateFile, fileName, GENERIC_READ, FILE_SHARE_READ, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
    mov [hFile], eax
    cmp eax, INVALID_HANDLE_VALUE
    jne read_file
    jmp display_error
```

```
read_file:
    invoke ReadFile, [hFile], fileBuffer, 256, bytesRead, 0
    invoke CloseHandle, [hFile]
    cmp dword [bytesRead], 0
    jne display_message
    jmp display_error
```

```
display_message:
    invoke MessageBox, NULL, fileBuffer, messageTitle, MB_OK
    jmp finished
```

```
display_error:
    invoke GetLastError
    invoke FormatMessage, FORMAT_MESSAGE_FROM_SYSTEM, 0, eax, 0, formatMessageBuffer, 256, 0
    invoke MessageBox, NULL, formatMessageBuffer, errorTitle, MB_OK
```

```
finished:
    invoke ExitProcess, 0
```

```
section '.idata' import data readable writeable
```

```
library kernel32, 'KERNEL32.DLL', user32, 'USER32.DLL'
import kernel32, CreateFile, 'CreateFileA', ReadFile, 'ReadFile', CloseHandle, 'CloseHandle', \
    GetLastError, 'GetLastError', FormatMessage, 'FormatMessageA', ExitProcess, 'ExitProcess'

import user32, MessageBox, 'MessageBoxA'
```



## Definire sectiuni din .text in .orice

mict.exe - PID: 15924 - Module: mict.exe - Thread: Main Thread 22044 - x32dbg

File View Debug Tracing Plugins Favourites Options Help Feb 19 2024 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols

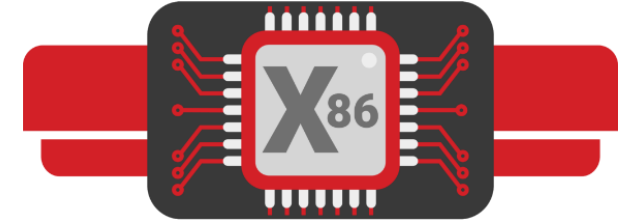
Address	Size	Party	Info	Content
00010000	00001000	User		
00020000	00001000	User		
00030000	00001000	User		
00040000	0001D000	User		
00060000	00035000	User	Reserved	
00095000	00008000	User		
000A0000	0003A000	User	Reserved	
000DA000	00006000	User	Stack (22044)	
000E0000	00004000	User		
000F0000	00002000	User		
00100000	00001000	User		
00110000	00001000	User		
00120000	00001000	User		
00130000	00010000	User		
00140000	00035000	User	Reserved	
00175000	00008000	User		
00180000	00035000	User	Reserved	
001B5000	00008000	User		
001C0000	00035000	User	Reserved	
001F5000	00008000	User		
00200000	001ED000	User	Reserved	
003ED000	0000E000	User	PEB, TEB (14036), wow64 TEB (14036), TEB (15940), wow64 TEB (15940),	
003FB000	00005000	User	Reserved (00200000)	
00400000	00001000	User	mict.exe	
00401000	00001000	User	".data"	
00402000	00001000	User	".blabla"	
00403000	00001000	User	".idata"	
00410000	000C9000	User	\Device\Harddisk0\Partition1\Windows\System32\locale.nls	
00520000	00007000	User	Heap (ID 0)	
00527000	00009000	User	Reserved (00520000)	
00600000	00007000	User		
00607000	00009000	User	Reserved (00600000)	
00610000	000FD000	User	Reserved	
0070D000	00003000	User	Stack (12012)	
00710000	000FD000	User	Reserved	
0080D000	00003000	User	Stack (15940)	
00810000	000FD000	User	Reserved	
0090D000	00003000	User	Stack (14036)	
75810000	00001000	System	win32u.dll	
75811000	00006000	System	".text"	
75817000	0000E000	System	".rdata"	
75825000	00001000	System	".data"	
75826000	00001000	System	".rsrc"	
75827000	00001000	System	".reloc"	
75D40000	00001000	System	kernelbase.dll	
75D41000	001DE000	System	".text"	
75F1F000	00004000	System	".data"	
75F23000	00006000	System	".idata"	
75F29000	00001000	System	".didat"	
75F2A000	00001000	System	".rsrc"	
75F2B000	00031000	System	".reloc"	
766E0000	00001000	System	msvcrt.dll	
766E1000	00006000	System	".text"	
7674F000	00003000	System	".data"	
76752000	00002000	System	".idata"	
76754000	00001000	System	".didat"	
76755000	00001000	System	".rsrc"	
76756000	00005000	System	".reloc"	
76760000	00001000	System	user32.dll	
76761000	000A5000	System	".text"	
76806000	00004000	System	".data"	
7680A000	00009000	System	".idata"	
76813000	00001000	System	".didat"	

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused System breakpoint reached! Time Wasted Debugging: 0:05:43:10

# TABELUL DE INSTRUCȚIUNI X86

## INSTRUCȚIUNILE OPCODE PENTRU ARHITECTURA X86



Note:

- Crearea unui tabel detaliat cu instrucțiunile x86, care să includă opcode-urile, modurile de adresare, și efectele asupra flagurilor, este o sarcină vastă datorită numărului mare de instrucțiuni și complexității arhitecturii x86. Totuși, voi oferi un exemplu de format pentru un astfel de tabel, care să acopere câteva instrucțiuni de bază. Acest exemplu poate servi ca punct de plecare în elaborarea unui material educațional mai cuprinzător pentru un curs de inginerie inversă.
- Acest tabel este doar un exemplu și acoperă o mică parte din setul de instrucțiuni x86. Pentru o listă completă și detalii despre fiecare instrucțiune, inclusiv modurile de adresare detaliate și efectele exacte asupra flagurilor, este recomandat să consultați documentația oficială a producătorului de procesor (de exemplu, manualele de la Intel sau AMD) sau alte resurse specializate în arhitectura x86. Aceste resurse vor oferi informațiile necesare pentru a înțelege în profunzime modul în care instrucțiunile x86 interacționează cu hardware-ul computerului.

•**Opcode:** Codul operației. Unele instrucțiuni au mai multe opcode-uri în funcție de modul de adresare sau de dimensiunea operandului.

•**Moduri de Adresare:** Indică cum sunt specificați operanzii (e.g., direct în instrucțiune, prin registre, prin adrese de memorie).

•**Efecte Asupra Flagurilor:** Indică care flaguri din registrul EFLAGS sunt modificate de execuția instrucțiunii.

•**Descriere:** O scurtă explicație a operației efectuate de instrucțiune.

Mnemonic	Opcode	Moduri de Adresare	Efecte Asupra Flagurilor	Descriere
MOV	Diverse	Registru, Memorie	Niciunul	Transferă date între registri sau între registru și memorie fără a modifica flagurile.
ADD	03	Registru, Memorie	CF, ZF, SF, OF, PF, AF	Adună doi operanzi și stochează rezultatul în operandul destinatar, actualizând flagurile de stare.
SUB	2D	Registru, Memorie	CF, ZF, SF, OF, PF, AF	Scade operandul sursă din operandul destinatar și stochează rezultatul în destinatar, actualizând flagurile de stare.
CMP	3D	Registru, Memorie	ZF, SF, OF, PF, CF, AF	Compară doi operanzi prin scădere și setează flagurile de stare în consecință fără a stochea rezultatul.
JMP	E9	Direct, Indirect	Niciunul	Efectuează un salt necondiționat la adresa specificată sau la cea calculată.
JE/JZ	74	Direct, Indirect	Utilizează ZF	Efectuează salt la adresa specificată dacă flagul ZF este setat (egalitate sau zero).
JNE/JNZ	75	Direct, Indirect	Utilizează ZF	Efectuează salt la adresa specificată dacă flagul ZF nu este setat (nu este egal sau nu este zero).
CALL	E8	Direct, Indirect	Niciunul	Apelază o subrutină la adresa specificată și pune adresa de retur pe stivă.
RET	C3	N/A	Niciunul	Întoarce execuția la adresa de pe vârful stivei.



# TABELE DE MODURI DE ADRESARE

MODURILE DE ADRESARE ÎN ARHITECTURA X86 DEFINESC MODUL ÎN CARE INSTRUCȚIUNILE ACCESEAZĂ DATELE DIN MEMORIE SAU REGISTRE

Instrucțiune	Ce face?
MOV	Copiază VALOAREA de la sursă în destinație
LEA	Încarcă ADRESA EFECTIVĂ (nu valoarea!) într-un registru

## 1. Adresare Directă (Immediate Addressing)

**Descriere:** Valoarea operandului este direct în instrucție.

**Exemplu:** **MOV AX, 1234h** - Valoarea 1234h este încărcată direct în registru.

## 2. Adresare Directă a Memoriei (Direct Memory Addressing)

**Descriere:** Adresa efectivă a operandului este specificată direct în instrucție.

**Exemplu:** **MOV AX, [1234h]** - Valoarea din memorie la adresa 1234h este încărcată în AX.

## 3. Adresare prin Registru (Register Addressing)

**Descriere:** Operandul este un registru.

**Exemplu:** **MOV AX, BX** - Conținutul lui BX este copiat în AX.

## 4. Adresare Indirectă prin Registru (Register Indirect Addressing)

**Descriere:** Adresa operandului este într-un registru.

**Exemplu:** **MOV AX, [BX]** - Valoarea din memorie la adresa conținută în BX este încărcată în AX.

Fiecare mod de adresare are aplicațiile sale, iar alegerea între ele depinde de contextul instrucțiunii și de datele pe care programul intenționează să le acceseze sau să le manipuleze. Înțelegerea acestor moduri de adresare este crucială pentru analiza și dezasamblarea eficientă a codului x86, permițând inginerilor reversi să urmărească fluxul de date prin program.

Registru	Nume complet	Funcție tradițională
AX	Accumulator	Operații aritmetice/logice
BX	Base	Adresare bazată pe offset-uri
CX	Counter	Bucă și repetiție (loop CX)
DX	Data	Operații I/O, multiplicări mari

Dimensiune	Registru	Ce înseamnă
8 biți	AL	Accumulator Low
16 biți	AX	Accumulator (clasic)
32 biți	EAX	Extended Accumulator
64 biți	RAX	Register Accumulator

Dimensiune	Nume registru	Exemplu
64-bit	RAX, RBX, RCX, RDX	← R = 64-bit register
32-bit	EAX, EBX, ECX, EDX	← E = extended
16-bit	AX, BX, CX, DX	← numele clasic x86
8-bit	AH, AL, etc.	← high/low byte

# TABELE DE MODURI DE ADRESARE

MODURILE DE ADRESARE ÎN ARHITECTURA X86 DEFINESC MODUL ÎN CARE INSTRUCȚIUNILE ACCESEAZĂ DATELE DIN MEMORIE SAU REGISTRE

## 5. Adresare cu Deplasament (Base plus Displacement Addressing)

**Descriere:** Adresa efectivă este suma dintre un registru (baza) și o valoare constantă (deplasament).

**Exemplu:** `MOV AX, [BX+10h]` - Valoarea din memorie la adresa rezultată din adunarea BX cu 10h este încărcată în AX.

## 6. Adresare Indexată (Indexed Addressing)

**Descriere:** Similar cu adresarea cu deplasament, dar folosește un registru de index pentru a calcula adresa.

**Exemplu:** `MOV AX, [SI+1234h]` - Valoarea din memorie la adresa calculată adunând SI și 1234h este încărcată în AX.

## 7. Adresare Bazată pe Index cu Deplasament (Based Indexed Addressing with Displacement)

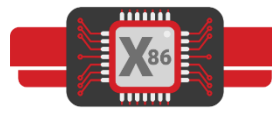
**Descriere:** Combinație a adresării indexate și a celei cu deplasament, folosind un registru de bază, unul de index și un deplasament.

**Exemplu:** `MOV AX, [BX+SI+10h]` - Adresa efectivă este calculată adunând BX, SI, și 10h.

## 8. Adresare Directă a Pointerului (Direct Addressing with Segment Override)

**Descriere:** Specifică un segment și o adresă directă; util pentru accesul la date în segmente diferite de cel implicit.

**Exemplu:** `MOV AX, ES:[1234h]` - Accesează date la adresa 1234h în segmentul specificat de ES.



# push/pop

În timp ce este adevărat că în multe convenții de apel (calling conventions) pentru diferite sisteme de operare și medii de rulare, instrucțiunile push sunt folosite pentru a plasa argumentele pe stivă înainte de un apel de funcție, nu toate instrucțiunile push pe care le vezi într-un disassembler vor fi pentru acest scop.

Instrucțiunea push este folosită pur și simplu pentru a pune o valoare pe stivă. Acest lucru poate fi făcut din mai multe motive:

- 1. Pentru Argumente de Funcții:** În multe cazuri, push este folosit pentru a trece argumente unei funcții, în special în convențiile de apel stdcall și cdecl care sunt comune în medii Windows și C.
- 2. Pentru Salvarea Regiștrilor:** Codul poate folosi push pentru a salva valoarea curentă a unui registru pe stivă înainte de a efectua operații care ar putea schimba acel registru, astfel încât valoarea originală poate fi recuperată (pop) mai târziu.
- 3. Pentru Alocarea Spațiului pe Stivă:** Uneori, un push poate fi utilizat pentru a ajusta stiva pentru alocarea de spațiu pentru variabile locale într-o funcție.
- 4. Pentru Instrucțiunile de Control al Fluxului:** Instrucțiunile push pot fi utilizate înainte de anumite instrucțiuni de control al fluxului, cum ar fi call (care, de asemenea, împinge adresa de întoarcere pe stivă).

Prin urmare, când citești cod disasamblat și vezi o instrucțiune push, trebuie să iei în considerare contextul în care este folosită. Urmărind fluxul de execuție și analizând cum sunt utilizate valorile împinse pe stivă, vei putea determina scopul acestora. Poate fi necesar să urmărești codul până la punctul în care valorile sunt efectiv folosite (de exemplu, un apel de funcție) pentru a înțelege de ce au fost plasate pe stivă.

Instrucțiunea pop în limbajul de asamblare este folosită pentru a elimina valoarea de pe vârful stivei și de obicei o stochează într-un registru sau într-o locație de memorie. Practic, face opusul instrucțiunii push – în timp ce push adaugă o valoare pe stivă, pop o scoate. Iată cum funcționează:

- 1. Extrage Valoarea de pe Vârful Stivei:** Când execuți o instrucțiune pop, procesorul va elimina valoarea care se află pe vârful stivei.
- 2. Stochează Valoarea:** Valoarea eliminată este apoi stocată în locația specificată de instrucțiunea pop, cum ar fi un registru.
- 3. Ajustează Pointerul de Stivă:** După ce valoarea este extrasă și stocată, pointerul de stivă (de exemplu, registrul ESP pe arhitecturile x86) este ajustat automat pentru a reflecta noua poziție a vârfului stivei, care este acum la elementul următor din stivă.

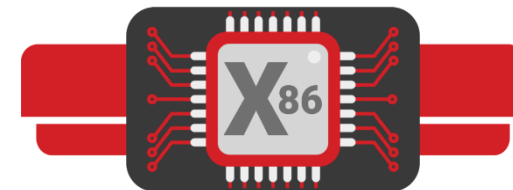
Instrucțiunea pop este adesea folosită în următoarele scenarii:

- **Pentru a Recupera Valori Salvate:** Pentru a recupera valori care au fost salvate anterior pe stivă cu push, adesea ca parte a conservării stării registrelor înainte de un apel de funcție.
- **Pentru a Obține Adresa de Întoarcere:** După ce o funcție a fost apelată cu instrucțiunea call, adresa de întoarcere este automat împinsă pe stivă de către CPU. La finalul funcției, se folosește pop pentru a obține această adresă și a reveni la punctul din cod unde a fost făcut apelul.
- **Pentru a Elibera Spațiu pe Stivă:** Dacă s-a folosit push pentru a alocă spațiu pe stivă (de exemplu, pentru variabile locale), pop poate fi folosit pentru a elibera acel spațiu după ce nu mai este nevoie de variabilele respective.

În rezumat, pop este o operație fundamentală în gestionarea stivei în limbajul de asamblare și este crucială pentru menținerea unui flux corect al programului și a stării procesorului.

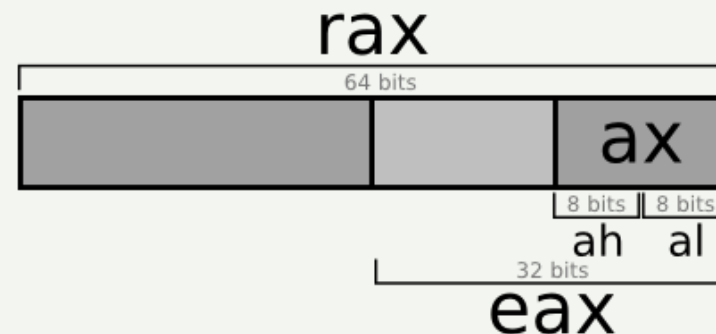
# Registre și steaguri CPU

AX devine EAX  
BX devine EBX  
SI devine ESI  
BP devine EBP etc.



## Registrele CPU în Arhitectura x86:

Categorie	Registre Principale	Descriere
Registre Generale	EAX, EBX, ECX, EDX	Utilizate pentru operații aritmetice, logice și de transfer de date
Registre de Index și Pointer	ESI, EDI, EBP, ESP	Utilizate pentru adresarea memoriei și manipularea stivei
Registre de Segment	CS, DS, ES, FS, GS, SS	Specifică segmentele de memorie pentru cod, date și stivă
Registre de Control și Stare	EFLAGS, EIP	Controlă execuția instrucțiunilor și indică starea procesorului



## Flaguri în Registrul EFLAGS:

Flag (Bit)	Nume	Descriere
0	CF (Carry Flag)	Indică un carry out sau borrow într-o operație aritmetică
2	PF (Parity Flag)	Indică paritatea numărului de biți setați la 1 în rezultat
4	AF (Adjust Flag)	Folosit în operații aritmetice BCD
6	ZF (Zero Flag)	Indică dacă rezultatul unei operații este zero
7	SF (Sign Flag)	Indică semnul (bitul cel mai semnificativ) al rezultatului
8	TF (Trap Flag)	Permite modul de depanare pas cu pas
9	IF (Interrupt Flag)	Controlează răspunsul la întreruperi
10	DF (Direction Flag)	Controlează direcția string operations
11	OF (Overflow Flag)	Indică dacă a avut loc un overflow într-o operație cu semn
12-13	IOPL (I/O Privilege Level)	Nivelul de privilegiu pentru operațiile de I/O
14	NT (Nested Task Flag)	Controlează înlănțuirea sarcinilor
16	RF (Resume Flag)	Controlează răspunsul la breakpoint-uri în modul de depanare
17	VM (Virtual-8086 Mode)	Indică dacă procesorul este în modul virtual 8086
18	AC (Alignment Check)	Controlează verificarea alinierii
19	VIF (Virtual Interrupt Flag)	Flag de întrerupere virtuală
20	VIP (Virtual Interrupt Pending)	Indică dacă o întrerupere virtuală este în așteptare
21	ID (ID Flag)	Suport pentru instrucțiunea CPUID

Arhitectura x86 include o serie de registre care sunt utilizate pentru a stoca date temporare necesare în timpul execuției programelor, precum și flaguri care indică diverse stări ale procesorului sau rezultate ale operațiilor.

Aceste registre și flaguri sunt fundamentale pentru programarea la nivel de sistem și dezvoltarea software-ului care necesită manipularea directă a hardware-ului. Registrele sunt folosite pentru a stoca temporar date și adrese în timpul execuției, în timp ce flagurile din registrul EFLAGS oferă informații despre starea curentă a procesorului și rezultatele operațiunilor executate.

# INSTRUCȚIUNILE DE TIP JUMP

SALT NECONDIȚIONAT / CONDIȚIONAT

Instrucțiunile de tip jump (săritură) în arhitectura x86 permit controlul fluxului execuției programului prin salturi la adrese specifice de cod. Salturile pot fi condiționate (efectuate doar dacă o anumită condiție este îndeplinită) sau necondiționate (efectuate întotdeauna). Iată un tabel simplificat care prezintă tipurile de instrucțiuni de salt în limbajul de asamblare x86, împreună cu o descriere scurtă pentru fiecare:

Mnemonic	Descriere	Tip de Salt
JMP	Salt necondiționat la o adresă specificată	Necondiționat
JE/JZ	Salt dacă egal/zero (Egalitate)	Condiționat
JNE/JNZ	Salt dacă nu este egal/nu este zero	Condiționat
JS	Salt dacă semnul este setat (negativ)	Condiționat
JNS	Salt dacă semnul nu este setat (pozitiv)	Condiționat
JP/JPE	Salt dacă paritatea este setată	Condiționat
JNP/JPO	Salt dacă paritatea nu este setată	Condiționat
JB/JNAE/JC	Salt dacă este mai mic decât (Fără semn)	Condiționat
JNB/JAE/JNC	Salt dacă nu este mai mic decât (Fără semn)	Condiționat
JL/JNGE	Salt dacă este mai mic decât (Cu semn)	Condiționat
JGE/JNL	Salt dacă este mai mare sau egal (Cu semn)	Condiționat
JG/JNLE	Salt dacă este mai mare decât (Cu semn)	Condiționat
JLE/JNG	Salt dacă este mai mic sau egal (Cu semn)	Condiționat
JA/JNBE	Salt dacă este mai mare decât (Fără semn)	Condiționat
JNA/JBE	Salt dacă este mai mic sau egal (Fără semn)	Condiționat
JO	Salt dacă overflow	Condiționat
JNO	Salt dacă nu este overflow	Condiționat

## Explicații Suplimentare

**Mnemonic.** Numele scurt al instrucțiunii, folosit în codul de asamblare.  
**Descriere.** O scurtă descriere a condiției sub care se efectuează saltul.  
**Tip de Salt.** Indică dacă saltul este efectuat necondiționat, indiferent de starea flag-urilor procesorului, sau condiționat, bazat pe evaluarea unei anumite condiții.

Instrucțiunile de salt condiționat testează starea flag-urilor în registrul de stat al procesorului (EFLAGS) pentru a determina dacă saltul trebuie să fie efectuat. De exemplu, instrucțiunea JE (Jump if Equal) sau JZ (Jump if Zero) efectuează saltul dacă flag-ul ZF (Zero Flag) este setat, indicând că ultima operație aritmetică sau logică a rezultat într-o valoare zero sau că două valori comparate sunt egale.

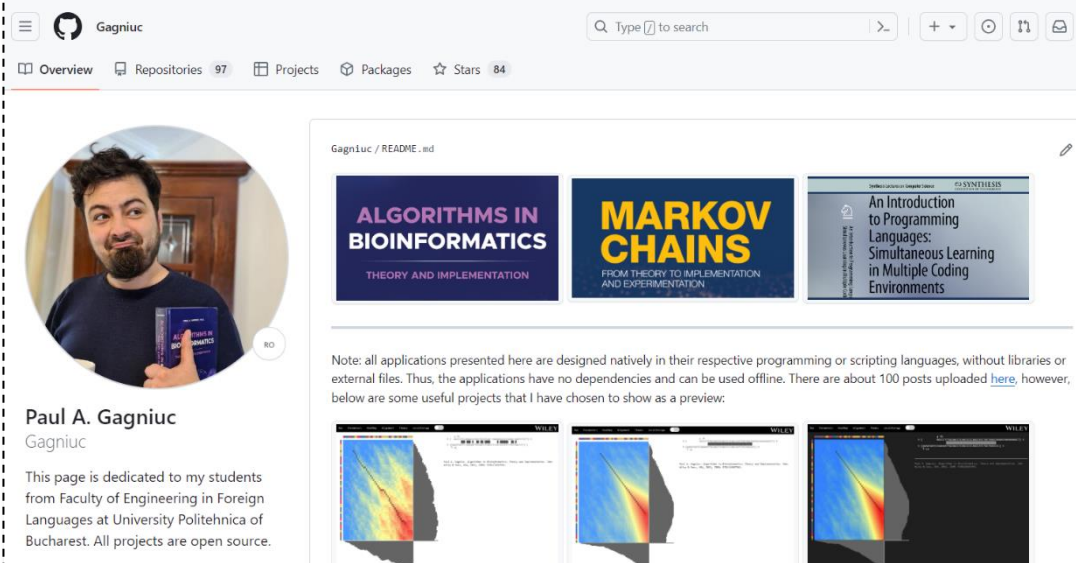
Fiecare din aceste instrucțiuni de salt are o utilizare specifică, permițând dezvoltatorilor să controleze fluxul execuției programelor în moduri complexe, de la bucle simple până la structuri de control condiționat avansate.



# 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>



Gagniuc

Overview Repositories 97 Projects Packages Stars 84

Gagniuc / README.md

**ALGORITHMS IN BIOINFORMATICS**  
THEORY AND IMPLEMENTATION

**MARKOV CHAINS**  
FROM THEORY TO IMPLEMENTATION AND EXPERIMENTATION

**An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments**

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.

Paul A. Gagniuc  
Gagniuc

This page is dedicated to my students from Faculty of Engineering in Foreign Languages at University Politehnica of Bucharest. All projects are open source.