



# Bitdefender®. Awake.

## Programare multi-modul

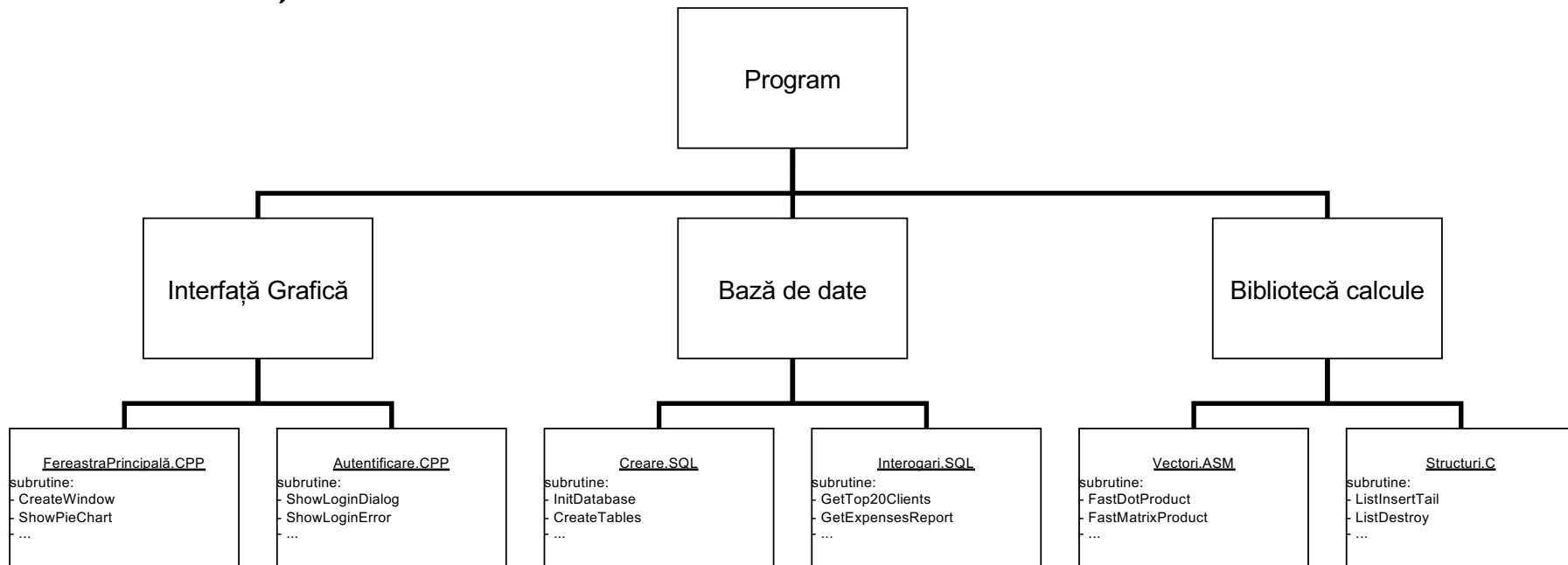
**Marius Vanța**

**[mvanta@bitdefender.com](mailto:mvanta@bitdefender.com)**

# 1. Arhitecturi modulare

# Programare modulară

- Cum împărțim problema în sub-probleme?
  - Modularizare:
    - program -> unități logice
    - cod (al unităților) -> fișiere distincte
    - Fișiere -> subrutine



# Programare modulară

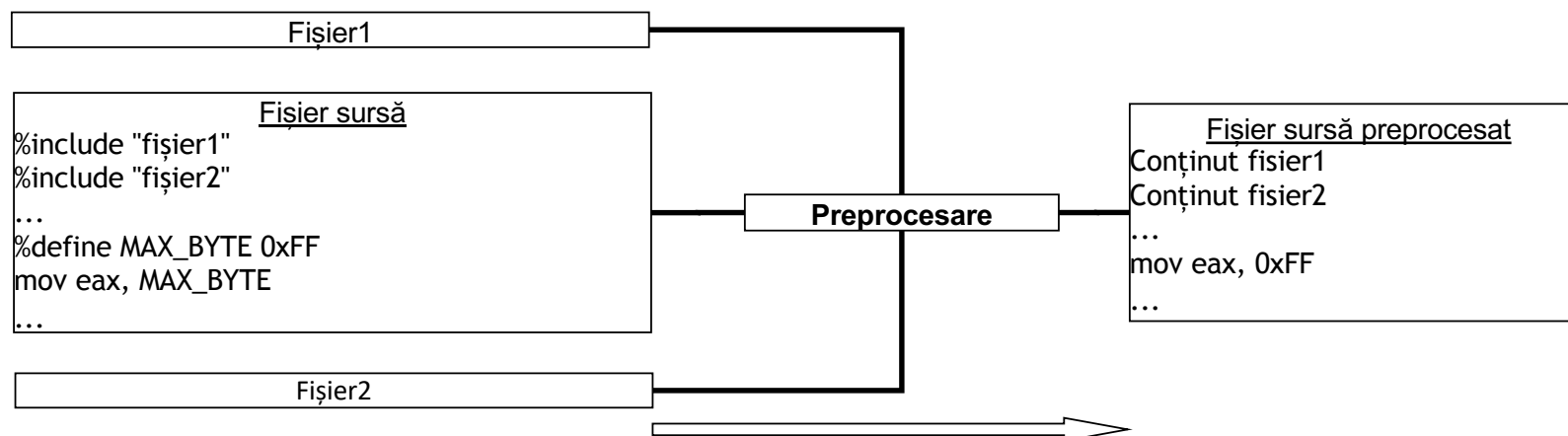


- Pentru care sub-probleme există deja rezolvări disponibile?
  - Reutilizare:
    - Fișiere sursă
      - *Refolosire cod și date din asamblare*
        - Directiva %include (NU este programare multimodul, deoarece la compilare ajunge DOAR 1 SINGUR MODUL obtinut prin concatenarea textuala a fișierelor incluse!!)
    - Fișiere binare
      - *Refolosire cod și date din asamblare*
      - *Cod și date din limbaje de nivel înalt*
      - *Biblioteci*
  - Existenta de fișiere binare separate implica COMPILARE SEPARATA !!!

## 2. Tehnici și instrumente

# Tehnici și instrumente

- Includerea **statică** la compilare/**asamblare**: directiva **%include**
  - Specifică limbajului (dar are echivalent și în alte limbaje)
  - Modularizare: permite doar divizarea codului scris în acel limbaj!
    - NU este programare multimodul! (aceasta necesita COMPILARE SEPARATA !!!)
  - Reutilizare: expune codul sursă!
  - Periculos și problematic:
    - Mecanism de preprocesor -> concatenare textuală a fișierelor
    - Expune cu vizibilitate globală toate denumirile -> conflicte (redefiniții/redeclarații)
    - Include fișierul în întregime - și ce se folosește și ce nu!



# Tehnici și instrumente



- Exemplu folosire **%include**

```
; fișierul constante.inc
```

```
; gardă dublă-includere
```

```
%ifndef    _CONSTANTE_INC_ ; la prima includere, _CONSTANTE_INC_ nu este definit
```

```
%define    _CONSTANTE_INC_ ; definim _CONSTANTE_INC_ -> condiție falsă la viitoare includeri
```

```
; recomandat ca astfel de fișiere (incluse de către altele) să conțină (doar) declarații!
```

```
MAX_BYTE    equ 0xFF
```

```
MAX_WORD    equ 0xFFFF
```

```
MAX_DWORD    equ 0xFFFFFFFF
```

```
MAX_QWORD    equ 0xFFFFFFFFFFFFFFFF
```

```
%endif ; _CONSTANTE_INC_
```

# Tehnici și instrumente



- Exemplu folosire **%include** – împachetare eax într-un BYTE/WORD/DWORD, conform magnitudinii valorii acestuia

```
; fișierul program.asm
#include "constante.inc"

    cmp     eax, MAX_BYTE
    ja      .nu_incape_in_octet      ; încapă valoarea din eax într-un byte?

.incape_in_octet:
    mov     [rezultat_octet], al      ; dacă da, salvăm AL în rezultat_octet
    jmp     .gata

.nu_incape_in_octet:
    cmp     eax, MAX_WORD
    ja      .nu_incape_in_cuvant      ; altfel verificăm dacă ajunge un WORD

.incape_in_cuvant:
    mov     [rezultat_word], ax       ; dacă da, salvăm AX în rezultat_word
    jmp     .gata

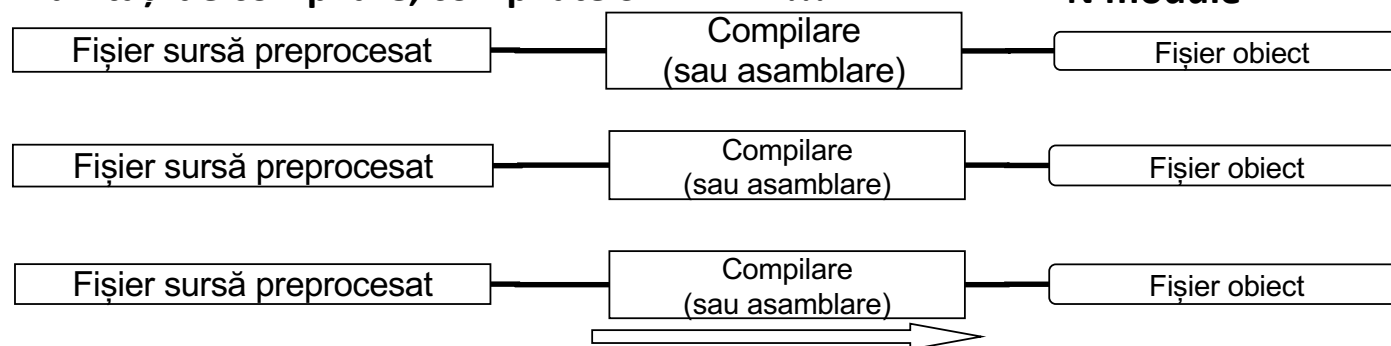
.nu_incape_in_cuvant:
    mov     [rezultat_dword], eax     ; dacă nu ajunge un WORD, salvăm întreg eax
.gata:
```



# Tehnici și instrumente

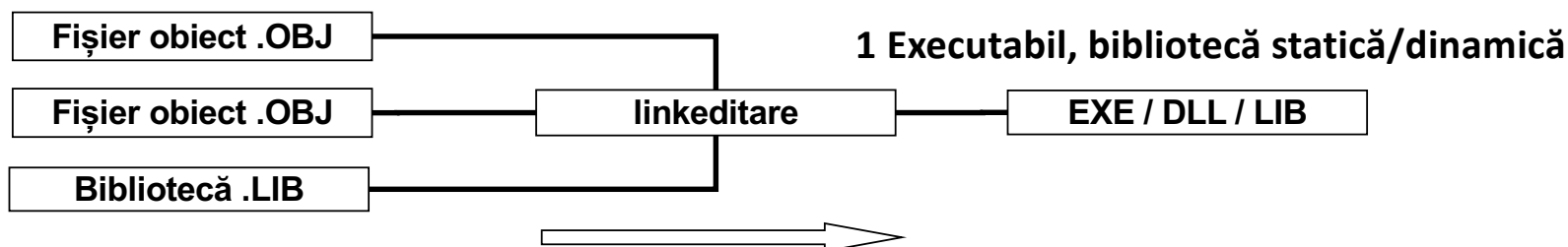
- Legarea **statică** la **linkeditare**
  - pas realizat de către un **linkeditor** după asamblare/compilare

**N unități de compilare, compilate SEPARAT !!!**



**N module**

**N module**



# Tehnici și instrumente



- Legarea **statică** la **linkeditare** – sumar responsabilități
  - Preprocesor: **text => text**
    - Efectuează prelucrări asupra *textului* sursă, rezultând un *text* sursă intermediar
    - Se poate imagina ca fiind o componentă a compilatorului sau asamblorului
    - Poate lipsi, multe limbaje nu au un preprocesor!
  - Asamblor: **instrucțiuni (text) => codificare binară (fișier obiect)**
    - Codifică instrucțiunile și datele (variabilele) din textul sursă preprocesat și construiește un fișier obiect ce conține cod mașină și valori de variabile alături de informații despre conținut (denumiri de variabile, subrutine, informații despre tipul și vizibilitatea acestora etc)
  - Compilator: **instrucțiuni (text) => codificare binară (fișier obiect)**
    - Identifică secvențe de instrucțiuni de procesor prin care se pot obține funcționalitățile descrise în textul sursă *iar apoi, precum un asamblor*, generează un fișier obiect ce conține codificarea binară a acestora și a variabilelor din program
    - Asamblarea este un caz special de compilare, unde instrucțiunile de procesor sunt gata oferite direct în textul programului și ca atare nu necesită să fie alese de către compilator!
  - Linkeditor: **fișiere obiect => bibliotecă sau program**
    - Construiește rezultatul final, adică un program (.exe) sau bibliotecă (.dll sau .lib) în care *leagă împreună* (include) codul și datele binare prezente în fișierele obiect.
    - Nu este interesat în ce compilatoare sau ce limbaje au fost folosite! Legarea necesită doar ca fișierele de intrare să respecte formatul standard al fișierelor obiect!

# Tehnici și instrumente



- **Legarea statică la linkeditare**

- Permite unirea mai multor **module binare** (fișiere obiect sau biblioteci statice) într-un singur fișier
  - Intrări: oricâte fișiere obiect (**.OBJ**) și/sau biblioteci statice (**.LIB**)
    - *Atenție, nu toate fișierele .LIB sunt biblioteci statice!*
  - Ieșire: .EXE sau .LIB sau .DLL (Dynamic-Link Library)
- Multimodul: oricâte fișiere pot fi compilate separat și linkeditate împreună
  - Pas realizat de linkeditor *după* compilare/asamblare -> **nu depinde de limbaj!**
- Reutilizare:
  - În formă binară - nu expune codul sursă!
  - Permite inter-operabilitate între limbaje diferite!
- Alte avantaje și dezavantaje:
  - Editorul de legături *poate* identifica și elimina resurse neutilizate sau efectua alte optimizări
  - Dimensiune mare a programului: programul înglobează resursele externe reutilizate
  - Dimensiune mare a programelor: bibliotecile populare duplicate în multe programe
- NASM: directivele **global (mecanism export)** și **extern (mecanism import)**
  - `global nume` – oferirea posibilității de utilizare din exterior a acestei resurse date prin nume
  - `extern nume` – solicitare de acces la resursa specificată; necesită să fie publică!

# Tehnici și instrumente



- Legarea **statică** la **linkeditare** – cerințele nasm
  - Resursele sunt partajate de comun acord
  - *Export* prin **global** nume1, nume2, ...
    - Ofer disponibilitate oricărui fișier ar fi interesat
  - *Import* prin **extern** nume1, nume2, ...
    - Solicit acces, indiferent din ce fișier va fi oferită resursa
  - Solicitare fără disponibilitate = eroare!
    - Nu se pot importa decât resurse ce sunt exportate undeva
  - Însă disponibilitate fără solicitare este caz permis. De ce?
    - Răspuns: chiar dacă niciun modul din program nu solicită/folosește, poate se va utiliza într-o versiune viitoare sau de către un alt program.
  - Limbajele de programare de nivel mai înalt oferă și ele la rândul lor construcții sintactice cu rol echivalent!
    - Exemplu: în limbajul C
      - *Disponibilitatea este automată/implicită, putându-se însă opta pentru a bloca accesul prin folosirea cuvântului cheie **static***
      - *Solicitarea de acces se face (tot) prin intermediul cuvântului cheie **extern***

# Tehnici și instrumente



- Legarea **statică** la **linkeditare** – cerințele nasm
  - Folosirea în practică a directivelor global și extern

## ; FIȘIER1.ASM

global Var1, Subrutina2

extern Var3, Subrutina3

Subrutina1:

....

apel(Subrutina3)

....

operatii(Var3)

....

Subrutina2:

....

Var1 dd ...

Var2 db ...

## ; FIȘIER2.ASM

extern Var1, Subrutina2

global Subrutina3, Var3

Subrutina3:

....

apel(Subrutina2)

....

operatii(Var1)

....

Subrutina1:

....

Var2 db ...

Var3 dd ...

# Tehnici și instrumente

- Legarea **statică** la **linkeditare** – cerințele nasm
  - Folosirea în practică a directivelor global și extern

## ; FIȘIER1.ASM

global Var1, Subrutina2

extern Var3, Subrutina3

Subrutina1:

....

apel(Subrutina3)

....

operatii(Var3)

....

Subrutina2:

....

Var1 dd ...

Var2 db ...

## ; FIȘIER2.ASM

extern Var1, Subrutina2

global Subrutina3, Var3

Subrutina3:

....

apel(Subrutina2)

....

operatii(Var1)

....

Subrutina1:

....

Var2 db ...

Var3 dd ...

*Pot fi refolosite  
denumirile cât timp  
nu sunt globale!*

# Tehnici și instrumente

- Exemplu program multimodul nasm + nasm
  - Pașii necesari construirii programului executabil final
    - Se assemblează fișierul main.asm
      - *nasm.exe -fobj main.asm*
    - Se assemblează fișierul sub.asm
      - *nasm.exe -fobj sub.asm*
    - Se editează legăturile dintre cele două module
      - *alink.exe main.obj sub.obj -oPE -entry:start -subsys:console*
  - Observație: cele două module pot fi asamblate în orice ordine! Abia în timpul linkeditării este necesar ca simbolurile referite să aibă cu toate implementare disponibilă în unul dintre fișierele obiect oferite linkeditorului.
  - Linkeditarea, în mod evident, este posibilă doar după asamblare/compilare!

# Tehnici și instrumente



- Exemplu program multimodul nasm + nasm

; MODULUL MAIN.ASM

global SirFinal  
extern Concatenare

import printf msvcrt.dll  
import exit msvcrt.dll  
extern printf, exit  
global start

segment code use32 public code class='code'

start:

```
mov eax, Sir1
mov ebx, Sir2
call Concatenare
push dword SirFinal
call [printf]
add esp, 1*4
push dword 0
call [exit]
```

segment data use32

```
Sir1 db 'Buna ', 0
Sir2 db 'dimineata!', 0
SirFinal resb 1000 ; spatiu pentru rezultat
```

; MODULUL SUB.ASM

extern SirFinal  
global Concatenare

segment code use32 public code class='code'

; eax = adresa primului sir, ebx = adresa sirului secund

Concatenare:

```
mov edi, SirFinal ; destinatie = SirFinal
mov esi, eax       ; sursa = primul sir
```

.sir1Loop:

```
lodsb ; luam octetul urmator
test al, al ; este terminatorul de sir (=0)?
jz .sir2 ; daca da, trecem la sirul al doilea
stosb ; (altfel) copiem in destinatie
jmp .sir1Loop ; si continuam pana la nul
```

.sir2:

```
mov esi, ebx ; sursa = sirul al doilea
```

.sir2Loop:

```
lodsb ; acelasi proces pentru noul sir
```

```
test al, al
```

```
jz .gata
```

```
stosb
```

```
jmp .sir2Loop
```

.gata:

```
stosb
```

```
; adaugam terminatorul de sir din al
```

```
ret
```



# Tehnici și instrumente



- Legarea **statică** la **linkeditare**: nasm + limbaje de nivel înalt
  - Cerințe ale editorului de legături
    - Directiva global pentru permis acces din alt limbaj către etichetele noastre
    - Directiva extern pentru obținut acces în NASM către resursele implementate în alte limbaje
    - Declararea în limbajul de nivel înalt a variabilelor și subrutinelor definite în nasm
      - *Exemplu C: declaratorul extern!*
  - Intrarea în procedură
  - Nealterarea valorilor unor regiștri
  - Transmiterea și accesarea parametrilor
  - Alocarea de spațiu pentru datele locale (opțional)
  - Întoarcerea unui rezultat (opțional)
- Ultimele aspecte sunt discutate in detaliu la convenții de apel!
  - vezi interfațarea cu limbajele de nivel inalt: convenții de apel

# Tehnici și instrumente



- Exemplu program multimodul asm + C

```
//
// AFISARE.C
//

// solicita catre preprocesorul de C includerea fisierului stdio.h
// stdio.h declara antetul (tipul de rezultat si parametri) functiei C printf
#include <stdio.h>

// declaram functia din fisierul asm incat compilatorul C sa cunoasca tipul de parametri si rezultat
// linkeditorul se va ocupa de implementarea functiei, compilatorul necesita doar sa-i cunoasca antetul
void asm_start(void); //echivalent ca efect cu extern void asm_start(void) ! Orice functie declarata la nivelul cel mai
// exterior al unui modul C face parte din clasa de memorie extern

// functia afisare este apelata de catre codul asm
void afisare(int *vector, int numar_elemente) //orice functie definita la nivelul cel mai exterior al
{                                           //unui modul C este implicit "globala" - adica se exporta implicit
    int index;
    for (index = 0; index < numar_elemente; index++)
    {
        printf("%d", vector[index]);
    }
    printf("\n");
}

// programul principal, acesta apeleaza functia asm_start scrisa in asamblare
void main(void) // de aici incepe executia programului final
{
    asm_start(); // apelam functia din fisierul asm
}
```

# Tehnici și instrumente



- Exemplu program multimodul asm + C

```
;
; VECTOR.NASM
;

; informam asamblorul despre existenta functiei afisare
extern _afisare          ; atenție la adăugarea _ ca prefix al a numelor provenite din C!

; informam asamblorul ca dorim ca asm_start sa fie disponibil altor unitati de compilare
global _asm_start       ; atenție la adăugarea _ ca prefix al numelor referite de către C!

; codul asm este dispus intr-un segment public, posibil a fi partajat cu alt cod extern
segment code public code use32

_asm_start:
    push dword elemente ; parametru transmis prin valoare (se urcă în stivă valoarea 5)
    push dword vector   ; vectorul este transmis prin referință (adresa lui este copiată pe stivă)
    call _afisare        ; apelul funcției C, din nou cu prefix _
    add esp, 4*2         ; afisare este o functie C (cdecl) -> necesită ca NOI să eliberăm argumentele!
    ret                 ; revenire la codul C care ne-a apelat

; linkeditorul poate folosi segmentul public de date si pentru date din afara
segment data public data use32
    vector dd 1, 2, 3, 4, 5 ; vectorul ce-l vom afisa cu rutina C
    elemente equ ($ - vector) / 4 ; constantă egală cu 5 (numarul elementelor din vector)
```

# Tehnici și instrumente



- Exemplu program multimodul asm + C
  - De ce \_ ?
  - Construire executabil:
    1. Compilare/asamblare:
      - *afisare.c poate fi compilat cu orice compilator C (după preferințe) -> afisare.obj*
        - Visual C: `cl /c afisare.c`
        - `nasm.exe vector.asm -fwin32 -o vector.obj`
    2. Editarea legăturilor:
      - *Se apelează orice linkeditor compatibil C, solicitând:*
        - Intrări: afisare.obj și vector.obj
        - Ieșire: aplicație de consolă
      - `link vector.obj afisare.obj /OUT:afisare.exe /MACHINE:X86 /SUBSYSTEM:CONSOLE`
  - Alternativ, fișierele pot fi înglobate într-o "soluție" Visual Studio, instruind IDE-ul:
    1. Să asambleze fișierul asm: specificând de exemplu drept **Pre-Build Event** comanda de asamblare de mai sus (`nasm.exe vector.asm -fwin32 -o vector.obj`)
    2. Să includă afisare.obj drept intrare adițională la linkeditare
    3. Există *extensii* pentru Visual Studio care rezolvă automat și transparent problema!