Laborator 11: Gestiunea bufferelor. Buffer overflow

Acest laborator își propune prezentarea conceptului de buffere în C și limbaj de asamblare împreună cu operațiile specifice acestora, dar și vulnerabilitățile pe care acestea le au și cum pot ele să fie exploatate de un potențial atacator prin folosirea unui program cu scopul de a ataca un sistem sau de a obține informații la care în mod normal nu ar avea acces.

Obiective:

- Prezentarea conceptelor de buffer și buffer overflow
- Exemple de atacuri de tip buffer overflow
- Prezentarea unor modalități de securizare a programelor pentru evitarea atacurilor de tip buffer overflow

Buffer. Buffer overflow

Ce este un buffer?

Un buffer este o zonă de memorie definită printr-o adresă de start și o dimensiune. Fie N dimensiunea bufferului, adică numărul de elemente. Dimensiunea totală a bufferului este N x dimensiunea unui element. Un șir de caractere (*string*) este un caz particular de buffer.

Ce este un buffer overflow?

Un buffer overflow este un eveniment care se produce atunci când în parcurgerea unui buffer se depășește limita superioară, adică poziția ultimului element (v[N - 1]). Un buffer overflow este un caz particular de *index out of bounds*, în care vectorul poate fi accesat folosind și indecși negativi. Multe funcții din C nu verifică dimensiunea bufferelor cu care lucrează, acestea cauzând erori de tip buffer overflow atunci când sunt apelate. Câteva exemple de astfel de funcții sunt:

- memcpy [http://www.cplusplus.com/reference/cstring/memcpy/]
- strcpy [https://www.cplusplus.com/reference/cstring/strcpy/]
- fgets [http://www.cplusplus.com/reference/cstdio/fgets/]

Un exemplu clasic de buffer overflow este dat de următorul cod:

```
char buffer[32];
fgets(buffer, 64, stdin);
```

În urma execuției acestui cod vom obține un buffer overflow care ar putea conduce chiar la o eroare de tip *Segmentation Fault*, însă acest lucru nu este garantat. Totul depinde de poziția bufferului pe stivă și de ceea ce se va suprascrie prin cei 32 de octeți ce depășesc dimensiunea bufferului. Mai multe detalii despre ceea ce se va suprascrie, dar și modalitatea prin care se va face acest lucru veți descoperi în timpul rezolvării exercițiilor.

Atacuri de tip buffer overflow

Cum este folosit buffer overflow?

Buffer overflow poate fi exploatat de un potențial atacator pentru a suprascrie anumite date din cadrul unui program, afectând fluxul de execuție și oferind anumite beneficii atacatorului. Cel mai adesea, un atacator inițiază un atac de tip buffer overflow cu scopul de a obține acces la date confidențiale, la care, în mod normal, un utilizator obișnuit nu ar avea acces.

Atacurile de tip buffer overflow sunt folosite în general pe buffere statice, stocate la nivel de stivă. Acest lucru se datorează faptului că pe stivă, pe lângă datele programului, se stochează și adrese de retur în urma apelurilor de funcții (vezi laboratorul 7). Aceste adrese pot fi suprascrise printr-un atac de tip buffer overflow, caz în care poate fi alterat fluxul de execuție a programului. Prin suprascrierea adresei de retur, odată cu încheierea execuției funcției curente nu se va mai reveni la execuția funcției apelante, ci se va „sări” la o altă adresă din cadrul executabilului de unde se va continua execuția. Acest eveniment poate conduce la comportament nedefinit al programului (*undefined behaviour*) dacă adresa la care se „sare” nu a fost calculată corect.

Scopul unui atacator este acela de a prelua controlul unui sistem prin obținerea accesului la un shell din care să poată rula comenzi. Acest lucru se poate realiza prin suprascrierea adresei de retur, folosind un apel de sistem prin intermediul căruia se poate deschide un shell pe sistemul pe care executabilul rulează (mai multe detalii la cursul de SO).

Cum ne protejăm de atacuri de tip buffer overflow?

Există multe modalități de a proteja un executabil de acest tip de atacuri. Pe majoritatea le veți studia în amănunt la cursul de SO anul viitor. O bună practică împotriva acestui tip de atac este de a evita folosirea unor funcții nesigure, precum cele prezentate mai sus. Mai multe detalii despre bune practici împotriva atacurilor de tip buffer overflow puteți găsi aici [https://security.web.cern.ch/recommendations/en/codetools/c.shtml].

De multe ori, bunele practici se dovedesc a fi insuficiente în „lupta” împotriva atacatorilor, motiv pentru care au fost inventate mai multe mecanisme de protecție a executabilelor prin manipularea codului și a poziției acestuia în cadrul executabilului (*Position Independent Code* - PIC [https://en.wikipedia.org/wiki/Position-independent_code]), prin randomizarea adreselor (*Address Space Layout Randomization* - ASLR [https://en.wikipedia.org/wiki/Address_space_layout_randomization]) sau prin introducerea unor verificări suplimentare în cod pentru a detecta eventuale atacuri. Aceste verificări se realizează prin introducerea unor valori speciale, numite **canary** pe stivă, între buffer și adresa de retur a funcției. Aceste valori sunt generate și plasate în cadrul executabilului de către compilator și diferă la fiecare rulare a executabilului. În momentul în care un atacator vrea să suprascrie adresa de retur se va suprascrie și valoarea canary și înainte de a se părăsi apelul funcției curente se va verifica dacă acea valoare a fost modificată sau nu. Dacă a fost modificată înseamnă că a avut loc un buffer overflow și execuția programului va fi întreruptă. Acest mecanism se numește **Stack Smashing Protection** sau **Stack Guard**. Mai multe detalii despre Stack Guard, dar și despre atacuri de tip buffer overflow puteți găsi aici [https://en.wikipedia.org/wiki/Buffer_overflow].

Feedback

Pentru a îmbunătăți cursul de IOCLA, componentele sale și modul de desfășurare, ne sunt foarte utile opiniile voastre. Pentru aceasta, vă rugăm să accesați și completați formularul de feedback de pe site-ul curs.upb.ro [https://curs.upb.ro/]. Trebuie să fiți autentificați și înrolați în cadrul cursului.

Formularul este anonim și este activ în perioada 21 mai 2022 - 03 iunie 2022. Rezultatele vor fi vizibile în cadrul echipei cursului doar după încheierea sesiunii. Puteți accesa formularul de feedback începând cu 21 mai 2022. Este accesibil la link-ul "Formular feedback" a paginii principale a cursului de IOCLA al seriei voastre. Nu este în meta-cursul disponibil tuturor seriilor.

Vă invităm să evaluați activitatea echipei de IOCLA și să precizați punctele tari și punctele slabe și sugestiile voastre de îmbunătățire a disciplinei. Feedback-ul vostru ne ajută să creștem calitatea materiei în anii următori și să îmbunătățim disciplinele pe care le veți face în continuare.

Vom publica la începutul semestrului viitor analiza feedback-ului vostru.

Ne interesează în special:

1. Ce nu v-a plăcut și ce credeți că nu a mers bine?
2. De ce nu v-a plăcut și de ce credeți că nu a mers bine?
3. Ce ar trebui să facem ca lucrurile să fie plăcute și să meargă bine?

Pregătire infrastructură

În cadrul laboratoarelor vom folosi repository-ul de git al materiei IOCLA - https://github.com/systems-cs-pub-ro/iocla [https://github.com/systems-cs-pub-ro/iocla]. Repository-ul este clonat pe desktop-ul mașinii virtuale. Pentru a îl actualiza, folosiți comanda git pull origin master din interiorul directorului în care se află repository-ul (`~/Desktop/iocla`). Recomandarea este să îl actualizați cât mai frecvent, înainte să începeți lucrul, pentru a vă asigura că aveți versiunea cea mai recentă. Dacă doriți să descărcați repository-ul în altă locație, folosiți comanda `git clone [[https://github.com/systems-cs-pub-ro/iocla|https://github.com/systems-cs-pub-ro/iocla]] ${target}`. Pentru mai multe informații despre folosirea utilitarului `git`, urmați ghidul de la Git Immersion [https://gitimmersion.com/].

În acest semestru am avut parte de o participare redusă la cursurile PCLP2. Ne dorim să înțelegem care sunt motivele pentru această situație, așa că vă rugăm să completați acest formular de feedback [https://docs.google.com/forms/d/1kesynY60tv8pduYFdCzB3tfLhj1uYDbVIRHCCgsa9jE]. Formularul este anonim și necesită aproximativ 30 de secunde pentru completare. Rezultatele acestuia ne vor ajuta să vă înțelegem mai bine perspectiva și să ne adaptăm nevoilor voastre.

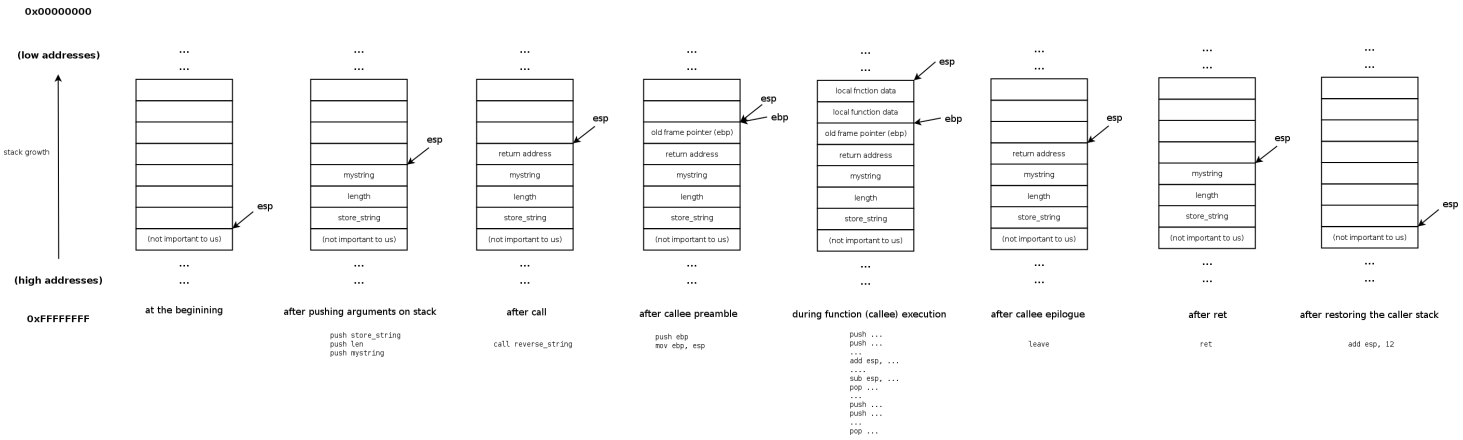
Pentru desfășurarea acestui laborator vom folosi interfața în linia de comandă.

Pe parcursul laboratorului, în linia de comandă, vom folosi:

- asamblorul `nasm`
- comanda `gcc` pe post de linker
- `objdump` și `ghidra` pentru dezasamblarea fișierelor obiect și executabile
- `gdb` pentru analiza dinamică, investigație și debugging

În general nu va fi nevoie să dați comenzi de compilare. Fiecare director cuprinde un Makefile pe care îl puteți rula pentru a compila în mod automat fișierele cod sursă limbaj de asamblare sau C.

Înainte de a începe laboratorul, alocați-vă 1-2 minute să reparcurgeți diagrama de mai jos ce arată structura stivei în cazul unui apel de funcție.



0. Stay a while and listen

Ați sărit direct la exerciții? Știm cum este. Totuși, vă rugăm să ne oferiți feedback și să ne ajutați să îmbunătățim materia. Mergeți la secțiunea feedback pentru detalii.

1. Tutorial: Folosirea unui buffer în zona de date

Accesați, în linia de comandă, directorul `1-data-buffer/` din arhiva de resurse a laboratorului și consultați fișierul `data_buffer.asm`. În acest fișier se găsește un program care populează un buffer cu informații și apoi le afișează.

Consultați cu atenție programul, apoi compilați-l folosind comanda:

```
make
```

Observați că în urma comenzii de compilare de mai sus au rezultat un fișier obiect și un fișier executabil, prin rularea comenzii:

```
ls
```

Rulați programul prin intermediul fișierului executabil, adică folosind comanda:

```
./data_buffer
```

Observați comportamentul programului în funcție de codul său.

2. Tutorial: Folosirea unui buffer pe stivă

Accesați directorul `2-3-4-stack-buffer/` din arhiva de resurse a laboratorului și consultați fișierul `stack_buffer.asm`. În acest fișier se găsește un program care populează un buffer cu informații și apoi le afișează. Este similar celui de mai sus doar că acum buffer-ul este alocat pe stivă.

Consultați cu atenție programul, apoi compilați-l folosind comanda:

```
make
```

apoi rulați-l folosind comanda:

```
./stack_buffer
```

Observați comportamentul programului în funcție de codul său.

Pe lângă buffer am mai alocat o variabilă locală pe 4 octeți, accesibilă la adresa `ebp-4`. Este inițializată la valoarea `0xCAFEBABE`. Această variabilă va fi importantă mai târziu. Ce este relevant acum este să știm că această variabilă este în memorie **imediat după buffer**: când se trece de limita buffer-ului se ajunge la această variabilă.

Care este diferenta intre cele 2 programe inspectate pana acum?

3. Citirea de date dincolo de dimensiunea buffer-ului

Acum că am văzut cum arată buffer-ul în memorie și unde este plasată variabila, actualizați programul `stack_buffer.asm` pentru ca secvența de afișare a buffer-ului (cea din jurul etichetei `print_byte`) să ducă și la afișarea octeților variabilei. Adică trebuie să citiți date dincolo de dimensiunea buffer-ului (și să le afișați). Este un caz de buffer overflow de citire, cu obiectiv de **information leak**: aflarea de informații din memorie.

Nu e ceva complicat, trebuie doar să "instruiți" secvența de afișare să folosească altă limită pentru afișare, nu limita curentă de 64 de octeți.

Afișați și alte informații dincolo chiar de variabila locală. Ce informație vine pe stivă după variabila locală (următorii 4 octeți)? Dar următorii 4 octeți după?

4. Scrierea de date dincolo de dimensiunea buffer-ului

Pe baza experienței de mai sus, realizați modificări pentru ca valoarea variabilei să fie `0xDEADBEEF` (în loc de `0xCAFEBABE` cum este la început) fără a modifica însă explicit valoarea variabilei. Folosiți-vă de modificarea buffer-ului și de registrul `ebx` în care am stocat adresa de început a buffer-ului.

Din nou, nu este ceva complicat. Trebuie să vă folosiți de valoarea `ebx` și un offset ca să scrieți valoarea `0xDEADBEEF` la acea adresă. Adică folosiți o construcție de forma:

```
mov byte [ebx+TODO], TODO
```

Realizați acest lucru după secvența de inițializare a buffer-ului (după instrucțiunea `jl fill_byte`).

La o rezolvare corectă a acestui exercițiu, programul va afișa valoarea `0xDEADBEEF` pentru variabila locală.

5. Tutorial: Citirea de date de la intrarea standard

Accesați directorul `5-6-read-stdin/` din arhiva de resurse a laboratorului și consultați fișierul `read_stdin.asm`. În acest fișier se găsește un program care folosește apelul `gets` ca să citească informații de la intrarea standard într-un buffer de pe stivă. La fel ca în cazul precedent am alocat o variabilă locală pe 4 octeți imediat după buffer-ul de pe stivă.

Consultați cu atenție programul, apoi compilați-l folosind comanda:

```
make
```

apoi rulați-l folosind comanda:

```
./read_stdin
```

Observați comportamentul programului funcție de input-ul primit.

6. Buffer overflow cu date de la intrarea standard

Funcția gets [https://man7.org/linux/man-pages/man3/gets.3.html] este o funcție care este practic interzisă în programele C din cauza vulnerabilității mari a acesteia: nu verifică limitele buffer-ului în care se face citirea, putând fi ușor folosită pentru buffer overflow.

Pentru aceasta transmiteți șirul de intrare corespunzător pentru ca valoarea afișată pentru variabila locală să nu mai fie `0xCAFEBABE`, ci să fie `0x574F4C46` (valorile ASCII în hexazecimal pentru `FLOW`).

Nu modificați codul în limbaj de asamblare. Transmiteți șirul de intrare în format corespunzător la intrarea standard pentru a genera un buffer overflow și pentru a obține rezultatul cerut.

Nu scrieți șirul `"574F4C46"`. Acesta e un șir care ocupă 8 octeți. Trebuie să scrieți reprezentarea ASCII a numărului `0x574F4C46` adică `FLOW`: `0x57` este `W`, `0x4F` este `O`, `0x4C` este `L` iar `0x46` este `F`.

x86 este o arhitectură little endian. Adică șirul `"FLOW"`, având corespondența caracter-cod ASCII `F`: `0x46`, `L`: `0x4C`, `O`: `0x4F`, `W`: `0x57` va fi stocat în memorie pe 4 octeți ca `0x574F4C46`. **Ce trebuie să faceți**: Va trebui ca, în cadrul buffer overflow-ului, să obțineți valoarea în memorie `0x574F4C46`. Obțineți șirul ASCII corespondent valorii în memorie `0x574F4C46` pe care trebuie să îl furnizați la intrarea standard a programului vulnerabil.

Ca să transmiteți șirul de intrare, e recomandat să-l scrieți într-un fișier și apoi să redirectați acel fișier către comanda aferentă programului. Puteți folosi un editor precum `gedit` sau `vim` pentru editarea fișierului. Avantajul acestora este că vă afișează și coloana pe care vă aflați și puteți să știți câte caractere ați scris în fișier. Alternativ, puteți folosi python pentru a vă genera mai ușor payload-ul. De exemplu, pentru a genera un payload care să suprascrie o valoare în cod cu valoarea `0xDEADBEEF`, puteți executa următoarea comandă:

```
python -c 'print "A"*32 + "\xEF\xBE\xAD\xDE"' > payload
```

E recomandat să numiți fișierul `payload`. Redirectarea fișierului `payload` către program se face folosind o comandă precum:

```
./read_stdin < payload
```

7. Buffer overflow cu date de la intrarea standard și fgets()

Așa cum am precizat mai sus, funcția `gets` este interzisă în programele curente. În locul acesteia se poate folosi funcția fgets [https://man7.org/linux/man-pages/man3/fgets.3.html]. Creați o copie a fișierului cod sursă `read_stdin.asm` din subdirectorul `5-6-read-stdin/` într-un fișier cod sursă `read_stdin_fgets.asm` în subdirectorul `7-read-stdin-fgets/`. În fișierul cod sursă `read_stdin_fgets.asm` schimbați apelul funcției `gets()` cu apelul funcției `fgets`.

Pentru apelul `fgets()` citiți de la intrarea standard. Ca argument pentru al treilea parametru al `fgets()` (de tipul `FILE *`) veți folosi intrarea standard. Pentru a specifica intrarea standard folosiți stream-ul stdin [https://linux.die.net/man/3/stdin]. Va trebui să îl marcați ca extern folosind, la începutul fișierului în limbaj de asamblare, construcția:

```
extern stdin
```

`stdin` este o adresă; pentru a apela `fgets()` cu intrarea standard, este suficient să transmitem pe stivă valoarea de la adresa `stdin`, adică folosind construcția:

```
push dword [stdin]
```

Urmăriți pagina de manual a funcției fgets [https://man7.org/linux/man-pages/man3/fgets.3.html] pentru a afla ce parametri primește.

Pentru apelul funcției `fgets()` folosiți construcția:

```
call fgets
```

De asemenea, marcați simbolul ca fiind extern folosind construcția:

```
extern fgets
```

Întrucât funcția `fgets()` are 3 parametri (care ocupă 3×4=12 octeți) va trebui ca după apelul funcției, în restaurarea stivei, să folosiți `add esp, 12` (în loc de `add esp, 4` ca în cazul programul de mai sus care folosea `gets()`).

Să păstrați posibilitatea unui buffer overflow și să demonstrați acest lucru prin afișarea valorii `0x574F4C46` pentru variabila locală. Adică să folosiți ca al doilea argument pentru `fgets()` (dimensiunea) o valoare suficient de mare cât să permită realizarea unui buffer overflow.

La fel ca mai sus, ca să transmiteți șirul de intrare pentru program, e recomandat să-l scrieți într-un fișier și apoi să redirectați acel fișier către comanda aferentă programului. Redirectarea fișierului `payload` către program se face folosind o comandă precum:

```
./read_stdin_fgets < payload
```

Nu modificați codul în limbaj de asamblare. Transmiteți șirul de intrare în format corespunzător la intrarea standard pentru a genera un buffer overflow și pentru a obține rezultatul cerut.

8. Buffer overflow pentru program scris în cod C

De cele mai multe ori vom identifica vulnerabilități de tip buffer overflow în programe scrise în C. Acolo trebuie să vedem ce buffere sunt și care este distanța de la buffer la variabila dorită pentru a putea face suprascrierea.

Este important de avut în vedere că distanța între un buffer și o altă variabilă în C poate nu corespunde cu cea "din teren"; compilatorul poate face actualizări, reordonări, poate lăsa spații libere între variabile etc.

Pentru exercițiul curent, accesați directorul `8-c-buffer-overflow/` din arhiva de resurse a laboratorului și observați codul sursă aferent în C. Pentru cazul în care doriți să nu mai compilați voi codul aveți în arhivă și fișierul limbaj de asamblare echivalent și fișierul în cod obiect și fișierul executabil.

Descoperiți diferența între adresa buffer-ului și adresa variabilei, creați un fișier de intrare (numit și `payload`) cu care să declanșați overflow-ul și faceți în așa fel încât să fie afișat mesajul *Full of win!*.

Ca să vedeți realitatea "din teren", adică să aflați care este diferența dintre buffer și variabila pe care dorim să o suprascriem, consultați fișierul în limbaj de asamblare echivalent (`do_overflow.asm`), obținut prin asamblarea codului C. În acest fișier puteți afla adresa relativă a buffer-ului față de `ebp` și a variabilei față de `ebp`; urmăriți secvența cuprinsă între liniile 32 și 41; aveți o mapare între numele variabilei și offset-ul relativ față de `ebp`. Cu aceste informații puteți crea șirul pe care să îl transmiteți ca payload către intrarea standard a programului.

Dacă doriți să recompilați fișierele rulați:

```
make clean && make
```

La fel ca mai sus, ca să transmiteți șirul de intrare pentru program, e recomandat să-l scrieți într-un fișier și apoi să redirectați acel fișier către comanda aferentă > programului. Redirectarea fișierului payload către program se face folosind o comandă precum:

```
./do_overflow < payload
```

9. Bonus: Stack canary

Pornind de la resursele exercițiului anterior din directorul `8-9-c-buffer-overflow` inspectați fișierul `Makefile`.

```
cat Makefile
```

Analizați atent opțiunile de compilare. Ce observați?

Așa cum ați observat în cadrul exercițiului anterior, deși am depășit dimensiunea buffer-ului și am suprascris o altă variabilă din program, acesta și-a încheiat execuția în mod normal. Acest lucru este nedorit atunci când lucrăm cu buffere, deoarece sunt o sursă de la care poate porni foarte ușor un atac. Folosind `objdump` inspectați funcția `main` a executabilului.

Pentru a inspecta sursa, folosiți următoarea comandă:

```
objdump -M intel -d do_overflow
```

Acum intrați în fișierul `Makefile` și modificați parametrii `CFLAGS` înlocuind `-fno-stack-protector` cu `-fstack-protector`. Recompilați programul și rulați-l. Ce observați?

Prin opțiunea sau flag-ul `-fstack-protector` i-am cerut compilatorului să activeze opțiunea de *Stack Smashing Protection* pentru executabilul nostru. Astfel, orice atac de tip buffer overflow va fi detectat în cod și execuția programului se va încheia cu eroare.

Inspectați din nou executabilul recompilat cu noul flag folosind `objdump`. Ce s-a schimbat?

Compilatorul a introdus pe stivă o valoare generată aleator, numită **canary**, pe care o verifică înainte de a părăsi execuția funcției curente. Prin buffer overflow, aceasta a fost suprascrisă odată cu depășirea dimensiunii buffer-ului, ceea ce a determinat o neconcordanță între valoarea inițială a canary-ului și cea de la finalul execuției funcției.

10. Bonus: Buffer overflow pentru binar

De multe ori nu avem șansa accesului la codul sursă și vrem să descoperim vulnerabilități în fișiere executabile. În directorul `10-overflow-in-binary` din arhiva de resurse a laboratorului, găsiți un fișier

executabil. Folosind `ghidra` sau `gdb` pentru investigație descoperiți cum puteți exploata vulnerabilitatea de tip buffer overflow, pentru ca programul să afișeze mesajul *Great success!*.

Pentru a rula `ghidra` pe fișierul executabil `overflow_in_binary` trebuie să vă creați un proiect nou în care să importați fișierul executabil. Ghidra va detecta automat formatul fișierului. Rulați analiza executabilului după care căutați în Symbol Tree după funcția main.

Identificați în codul dezasamblat cum se transmite intrarea către program. Identificați unde este buffer overflow-ul. Identificați condiția de comparație pe care doriți să o declanșați. Apoi construiți payload-ul corespunzător și transmiteți-l în forma adecvată programului.

Soluții

Soluțiile pentru exerciții sunt disponibile aici [https://elf.cs.pub.ro/asm/res/laboratoare/lab-11-sol.zip].

Resurse

Dacă laboratorul v-a impresionat într-un mod plăcut, puteți afla mai multe despre acest tip de atac, dar și despre securitate cibernetică în general, de pe acest canal [https://www.youtube.com/c/LiveOverflow].