În laboratorul de astăzi vom vedea cum putem executa cod propriu exploatând o vulnerabilitate numită buffer overflow. Atunci când găsim un program vulnerabil, acesta poate fi folosit pentru a executa cod malitios. Deoarece acest atac este destul de vechi, de-a lungul timpului au aparut mai multe metode pentru a împiedica executarea atacului. În acest laborator ne vom lovi de unele dintre aceste metode.

Avem următorul cod vulnerabil:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void func(char *string)
{
    char buffer[128];
    strcpy(buffer, string);
}
int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

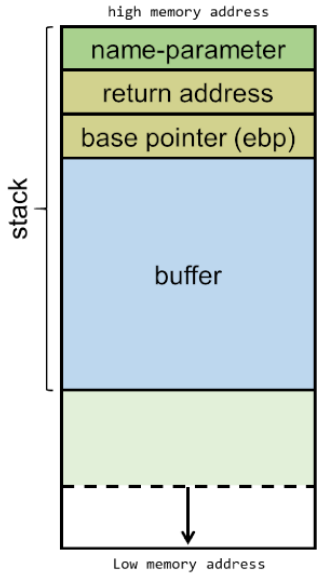
Îl puteți compila cu alinierea memoriei specifică sistemelor pe 32 de biți cu comanda

gcc -g -m32 -o overflow overflow.c

Pentru a avea optiunea -m32 trebuie să instalam două dependințe cu comanda

sudo apt-get install libc6-dev-i386 gcc-multilib

Dacă vom analiza programul, vom vedea că programul nu face altceva decât să apeleze o funcție cu argumentul dat de noi din consolă. Funcția copiază string-ul dat ca argument într-o variabilă locală având o dimensiune de 128 de caractere. Din cauza modului în care string-ul dat de noi este copiat (fără să se verifice dimensiunea) și a convenției apelurilor de funcții, apare un buffer overflow dacă îi dăm ca argument funcției *func* un string mai mare de 128 de caractere. Deoarece variabila în care copiem este una locală și este alocată pe stivă, vom reaminti cum arată stiva la apelul unei funcții:



Pentru a rula programul cu un input din consolă vom printa caractere cu funcția de printare din python (este necesară instalarea python cu comanda *sudo apt-get install python*). Astfel putem rula comanda *./overflow $(python -c 'print 128* "A"')* și să punem 128 de caractere „A” în buffer.

Practic: Scopul nostru final este să rescriem adresa de întoarcere a funcției către codul nostru malițios. Ajutându-vă și de imaginea de mai sus încercați să gasiți un input pentru care se rescrie adresa de întoarcere. Vă puteți ajuta de gdb. Atunci când porniți gdb, puneți un breakpoint la linia unde se află funcția strcpy (în exemplul nostru: linia 8 cu comanda *b 8*), rulați programul cu comanda *run*, iar cu comenzile p &buffer și p $ebp observați cât spațiu ocupă de fapt variabilele locale pe stivă. Pentru calculul va poate fi de folos următorul link:

https://www.calculator.net/hex-calculator.html

Explicați cum ați găsit input-ul. Ce observați când rulați programul cu acest input ?

Avem de-a face cu primul mecanism de protecție de astazi numit Stack Canary. Mecansimul presupune punerea unei valori aleatoare pe stivă înaintea adresei de întoarcere a funcției. Când noi încercăm să rescriem adresa de întoarcere, rescriem mai întâi această valoare aleatoare și cauzăm detectarea atacului și intrarea într-o rutină de eroare. Pentru a opri acest mecanism, trebuie să îl dezactivăm la compilare cu comanda *gcc -g -m32 -fno-stack-protector -o overflow overflow.c*

Practic: Încercați să rulați în terminal programul cu input-ul găsit anterior. Rulați programul în gdb și analizați comportamentul acestuia.

Va trebui să rescriem adresa de întoarcere a funcției astfel încât să aibă sens și să fie o adresă validă. Putem deci să încercăm un atac care să lanseze un shell spre exeplu. Pentru asta, va trebui ca în input să punem o succesiune de bytes numită shellcode care să execute un shell și să rescriem adresa de întoarcere către acest shellcode.

Vom avea nevoie de adresa la care este depus buffer-ul pe stivă. Pentru ca adresa să fie corectă și atacul să meargă va trebui să deazctivăm mecansimul de protecție ASLR care presupune încărcarea programului și a stivei la adrese de memorie diferite la fiecare rulare. Putem dezactiva ASLR cu comanda

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

Acum putem intra in gdb. Punem un breakpoint la functia *func* cu *b func* și rulăm programul cu *run*. Trebuie să rulăm cu un input care rescrie adresa de întoarcere. Execuția se va opri la breakpoint-ul din *func* și cu ajutorul comenzii *p &buffer* putem afla adresa unde se află buffer-ul pe stivă. Pentru a efectua atacul vom folosi următorul shellcode:

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80

Care reprezintă codul pentru executarea shell-ului scris sub formă de caractere (bytes transformați în caractere) și care poate fi dat ca input programului vulnerabil. Se observă că shellcode-ul ocupă 25 de bytes.

Este timpul să ne gândim cum vom construi input-ul pentru atac. Să o luăm în ordine inversă. Știm de la sarcinile precedente că sunt necesare 144 (în funcție de sistemul de operare - valoarea poate diferi) de caractere date ca input pentru a rescrie adresa de întoarcere care reprezintă 4 caractere (4 bytes). Mai rămân 140, dintre care 4 caractere (bytes) pentru frame pointer. Mai avem nevoie de 136 caractere. Numărul de caractere pentru care se rescrie adresa de întoarcere pot fi diferit, dar principiul este același.

Ideal, ar trebui ca adresa de întoarcere să fie rescrisă cu adresa de pe stivă unde este shellcode-ul dat de noi ca input. Dar noi nu știm unde începe shellcode-ul, știm doar adresa la care începe buffer-ul (găsită anterior).

Așa ca vom folosi tehnica **NOP sled** care presupune punerea în input a unui șir de instrucțiuni *nop* care nu au niciun efect. Această instrucțiune face ca procesorul să execute următoarea instrucțiune. Astfel, vom pune ca adresă de întoarcere o adresă la care se găsește o instrucțiune *nop* din șirul de instrucțiuni *nop* dat în input. Instrucțiunea *nop* poate fi pusă în input cu ajutorul caracterului **\x90**

Să ne întoarcem la construcția input-ului și să pornim din capătul opus. Să spunem că dorim să punem 40 de *nop-uri* adică 40 de bytes. Prima dată vom pune în input *nop-urile*, urmate de shellcode (toți cei 25 de bytes). În total 65 de bytes. Diferența de 75 de bytes (140-65) o vom în input ca 71 de caractere „A” și 4 caractere „B” pentru a vedea mai bine când se rescrie frame pointer-ul.

Input-ul nostru va arăta așa:

[40 - nop] + [25 - shellcode] + [71 - „A”] + [4 - „B”] + [addr_nop]

Pentru a scurta timpul de execuție vom pune ca adresa de întoarcere să sară în mijlocul secvenței de *nop*, adică addr_nop să fie adresa buffer-ului + 20 (unde 20 trebuie convertit în hexazecimal). Adresele se pot scrie în interiorul print-ului ca un string, în ordine inversă și fiecare 2 cifre din adresă se prefixează cu \x. Exemplu: adresa 0xffffd25c va fi scrisă ca „\x5c\xd2\xff\xff”. Deoarece memoria este gestionată altfel în gdb, atunci când construim input-ul va trebui să adăugam un offset între 20 și 100 la adresa găsită („ghicim” adresa la care vrem să sărim – de obicei acest lucru se face cu o buclă), deoarece adresa de început a buffer-ului este aceeași pentru că am dezactivat ASLR. Ca ultimă alternativă, puteți printa adresa reala a buffer-ului din program cu comanda *printf(„%p”, buffer);* pusă în *func* înainte de apelul la *strcpy*.

String-urile se pot concatena folosind operatorul +. Pentru calculul adreselor, va poate fi de folos următorul link:

https://www.calculator.net/hex-calculator.html

Practic: Încercați să montați atacul, pentru început în gdb. Ce observați ?

Aici întâmpinăm altă problemă, ca un mecanism de protecție. Memoria unde este alocată stiva este marcată ca fiind neexecutabilă. Ceea ce înseamnă că nu putem executa shellcode-ul nostru. Pentru a dezactiva acest mecansim compilăm programul cu comanda

gcc -g -m32 -fno-stack-protector -z execstack -o overflow overflow.c

Încercând diverse offset-uri pentru adresa de întoarcere astfel încât să săriți în mijlocul șirului de NOP-uri, montați atacul în terminal.

Practic: Activați ASLR cu comanda

echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

și explicați de ce atacul nu mai funcționează.