

Securitate Software

II. Vulnerabilități legate de coruperea memoriei

Continut

- 1 Atacuri de tipul buffer overflow
- 2 Mecanisme de protecție
- 3 Exemple

Definiții

Buffer overflow apare atunci când date sunt scrise într-un buffer de lungime fixă, iar mărimea acestor date depășește capacitatea buffer-ului.

Mai general: orice acces (R / W) în afara zonei rezervate (sub / peste).

Cauze apariție: nevalidarea datelor introduse de utilizator.

Întâlnit în aplicații scrise în C/C++

- mai rar, *managed code* (.NET, Java)

Principalele efecte (riscuri):

- modificarea / coruperea datelor aplicației;
- blocarea aplicației \implies atac DoS;
- controlul fluxului de execuție al aplicației \implies alterarea comportamentului aplicației.

Context

- procese, organizarea memoriei:
 - ▶ *code*: codul programului și bibliotecile
 - ▶ *data*: variabile globale și statice, heap
 - ▶ *stack*: argumentele funcțiilor, variabile locale, date de control (adresa de retur)
- datele, informații de control și codul sunt amestecate
 - ▶ codul / informațiile de control pot fi suprascrise
 - ▶ sistemul incurcă datele cu codul

Exemplu clasic

```
char dst[5];  
char *src = "0123456789";  
strcpy(dst, src);
```

Stack overflow

- ce este stiva?
- structura și utilizarea în arhitectura Intel
 - ▶ convenții de apel, stack frames, etc.
- exploatare
 - ▶ posibil datorită prezenței pe stivă a datelor și a informațiilor de control
 - ▶ ex. variabile locale pentru funcții și
 - ★ saved stack frame pointer (EBP)
 - ★ adresa de retur

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```

Stiva II

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];

    /* Operations */
}
```


Stiva III

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];

    /* Operations */
}
```

Function
parameters

Stiva IV

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];

    /* Operations */
}
```

Return Address	Function parameters
-------------------	------------------------

Stiva V

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];

    /* Operations */
}
```

Saved Frame Pointer	Return Address	Function parameters
------------------------	-------------------	------------------------

Stiva VI

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionOne(int c)
{
    int LocalInt;
    char LocalBuffer[32];

    /* Operations */
}
```

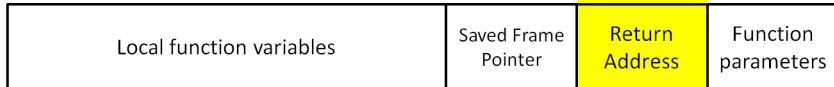
Local function variables	Saved Frame Pointer	Return Address	Function parameters
--------------------------	---------------------	----------------	---------------------

Stiva VII

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```



```
void FunctionTwo(void)
{
    /* Operations */
}
```



Structura stivei (stack frames)

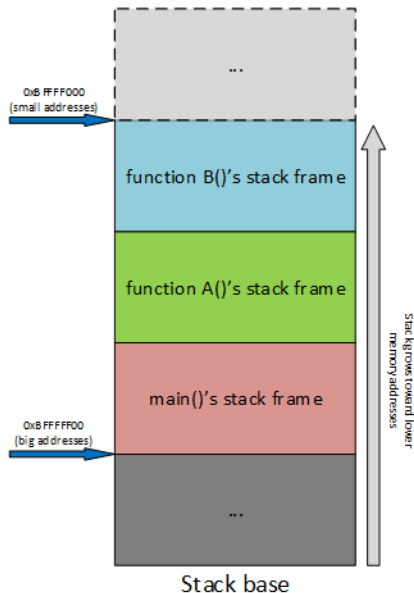
```
int function_B(int a, int b) {  
    int x, y; // variabile locale  
    x = a * a;  
    y = b * b;  
    return (x + y);  
}  
  
int function_A(int p, int q) {  
    int c; // variabile locale  
    c = p * q * function_B(p, p);  
    return c;  
}  
  
int main(int argc, char **argv, char **env) {  
    int res;  
    res = function_A(1, 2);  
    return res;  
}
```

Utile

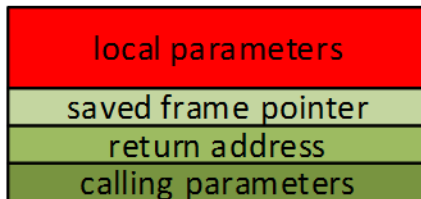
Pentru a vizualiza rapid trecerea din cod C/C++ în cod asamblare puteți folosi <http://gcc.godbolt.org/>

- folosiți compilatorul gcc,
- sintaxa Intel,
- opțiunea *-m32* pentru a compila programul pe 32 de biti

Stack frames II



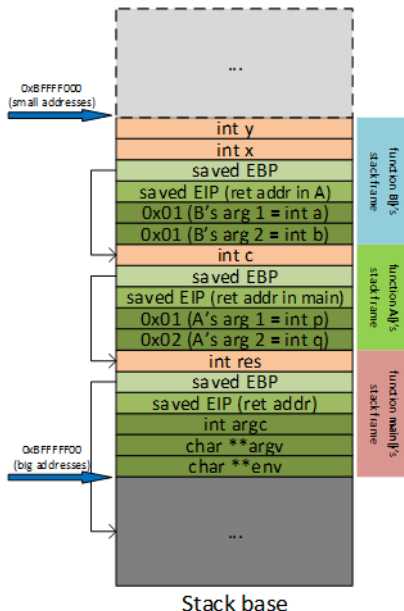
function A()'s stack frame



Stack frames III

```
int function_B(int a, int b)
push EBP
mov EBP, ESP
sub ESP, 48h
...
int function_A(int p, int q)
push EBP
mov EBP, ESP
sub ESP, 44h
...
push 1
push 1
call function_B
int main(int argc, char **argv, char **env)
push 2
push 1
call function_A
...
```

Stack frames IV



Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții

- argumentele unei funcții și variabilele locale sunt pe stivă:
 - ▶ partea nedorită din buffer poate suprascrie aceste valori
- specific aplicației
- depinde de modul în care compilatorul generează codul
 - ▶ convenții de apel, organizarea stivei

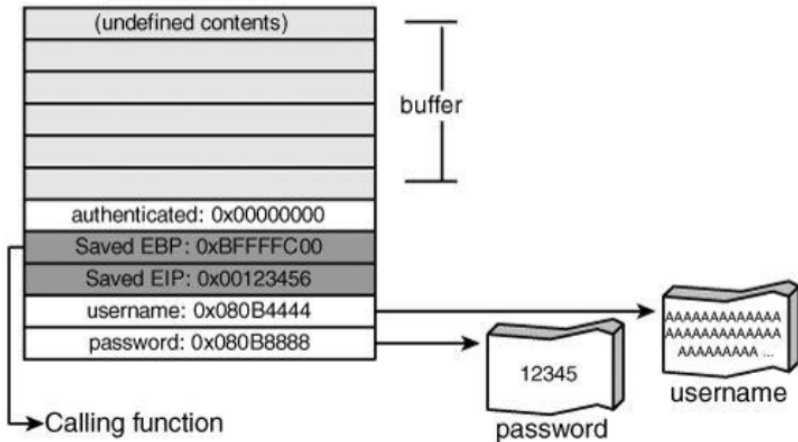
Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții II

ex. alterarea valorii variabilei locale *authenticated* \implies schimbarea funcționalității

```
int authenticate(char *username, char *password) {
    int authenticated;
    char buffer[1024];
    authenticated = verify_password(username, password);
    if (authenticated == 0) {
        sprintf(buffer, "incorrect password for user %s\n", username);
        log("%s", buffer);
    }
    return authenticated;
}
```

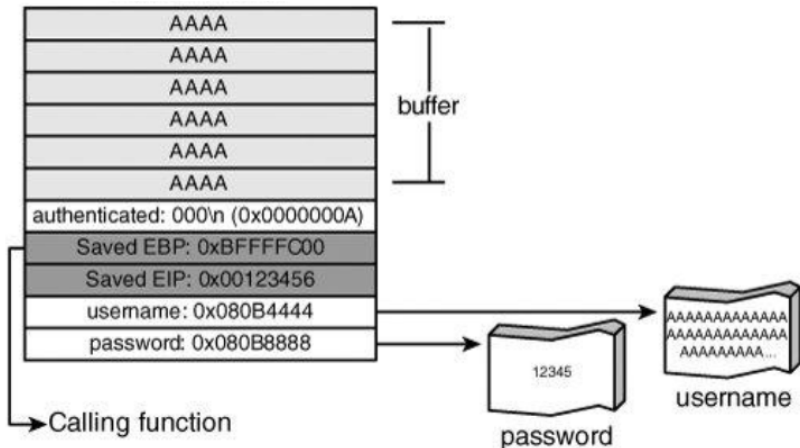
Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții III

Stack frame pentru funcția `authenticate()`



Exploatare: suprascrierea variabilelor locale sau a argumentelor unei funcții III

Stack frame pentru funcția `authenticate()` după exploatare = atacator autentificat



Exploatare: suprascrierea datelor de control

- ① suprascrie adresa de retur \implies deturna fluxul aplicației, execuția revine
 - ▶ într-o zonă de memorie ce conține date controlate de atacator
 - ★ ex. variabile globale, locație de pe stivă, buffer static ce conține codul atacatorului
 - ★ cod injectat de atacator: *shellcode* (încearcă stabilirea unei conexiuni cu mașina atacatorului)
 - ★ datorat confuziei între date și cod
 - ★ posibil dacă este permisă execuția codului injectat
 - ▶ undeva în codul aplicației sau într-o bibliotecă comună
 - ★ unde se găsește cod util pentru atacator
 - ★ ex. apelul unei funcții sistem dintr-o bibliotecă
 - ★ independent de codul atacatorului

Exploatare: suprascrierea datelor de control II

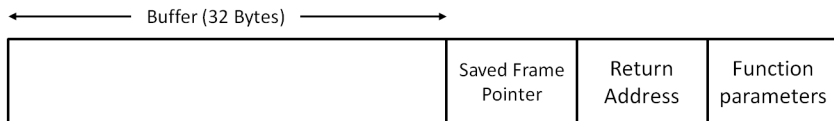
- ② suprascrie *stack frame pointer (EBP)*
 - ▶ modifică execuția funcției apelante
 - ▶ specific aplicației

Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```



Exemplu

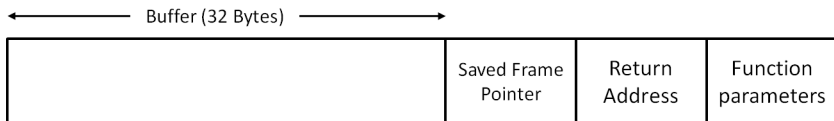
/* UNSAFE Function */

```
void UnsafeFunction(char * str)
{
    char Buffer[32];

    /* Copy str into Buffer */
    strcpy(Buffer,str);
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"
2. "A" * 40

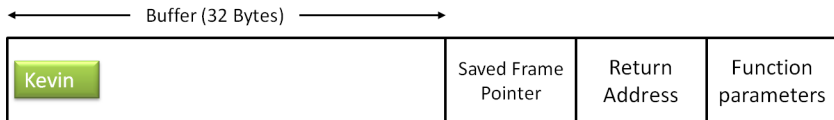


Exemplu

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    char Buffer[32];  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"
2. "A" * 40



Exemplu

/* UNSAFE Function */

```
void UnsafeFunction(char * str)
```

```
{
```

```
    char Buffer[32];
```

```
    /* Copy str into Buffer */
```

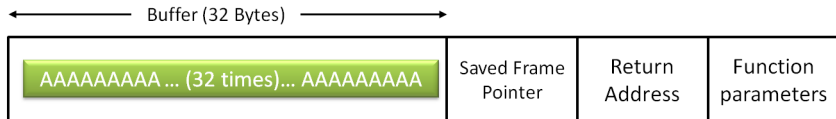
```
    strcpy(Buffer,str);
```

```
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"

2. "A" * 40



Exemplu

/* UNSAFE Function */

```
void UnsafeFunction(char * str)
{
    char Buffer[32];

    /* Copy str into Buffer */
    strcpy(Buffer, str);
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"

2. "A" * 40



Exemplu

/* UNSAFE Function */

```
void UnsafeFunction(char * str)
```

```
{
```

```
    char Buffer[32];
```

```
    /* Copy str into Buffer */
```

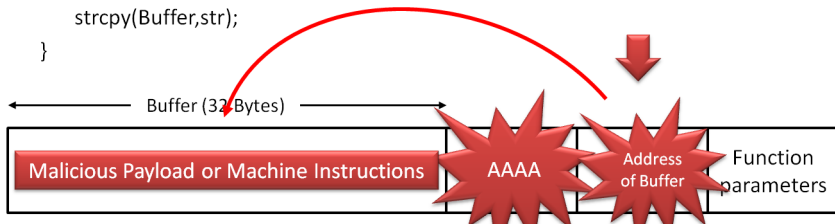
```
    strcpy(Buffer,str);
```

```
}
```

SAMPLE INPUTS (STR VALUES):

1. "Kevin"

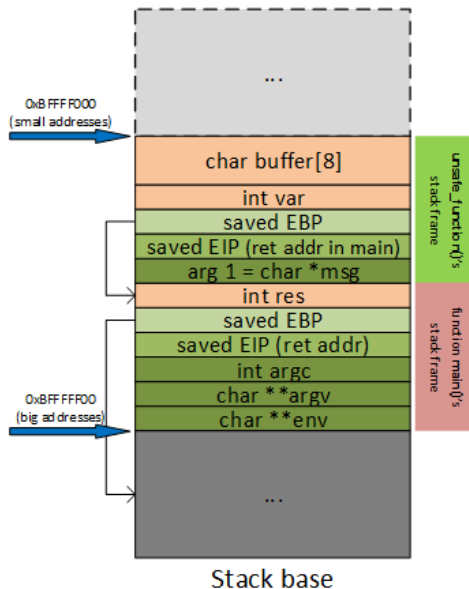
2. "A" * 40



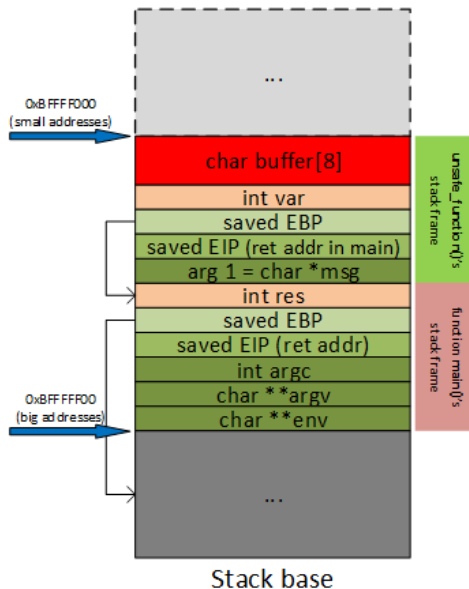
Exemplu: suprascrierea adresei de retur

```
int unsafe_function(char *msg) {  
    int var; // variabila locala  
    char buffer[8];  
    var = 10;  
    strcpy(buffer, msg);  
    return var;  
}  
  
int main(int argc, char **argv, char **env) {  
    int res;  
    /* Buffer overflow pentru "strlen(argv[1]) >= 8" */  
    res = unsafe_function(argv[1]);  
    return res;  
}
```

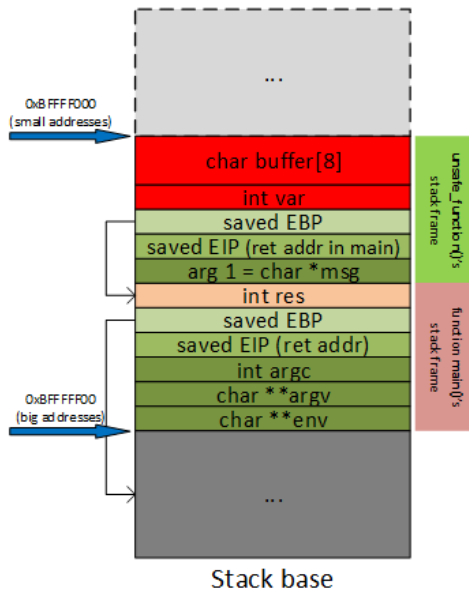

Vizualizare stiva



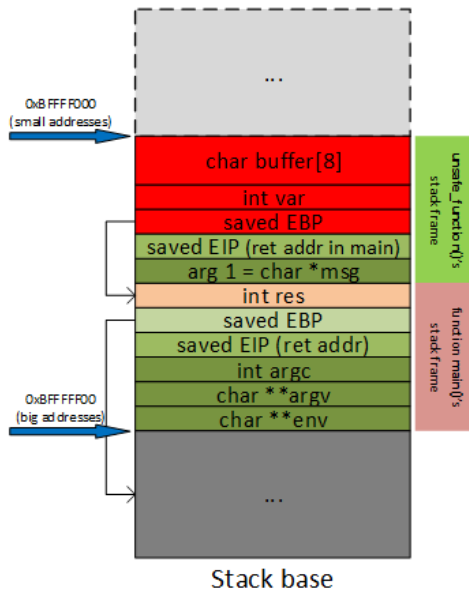
Vizualizare stiva



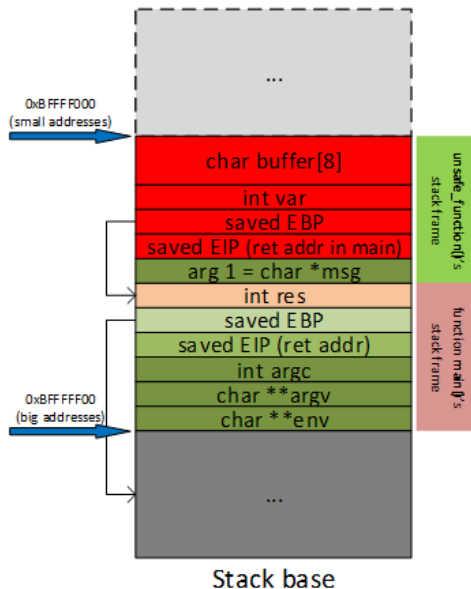
Vizualizare stiva



Vizualizare stiva



Vizualizare stiva



Exemplu de depășire cu 1 (suprascrisere EBP)

```
int unsafe_function(char *msg) {
    char buffer[512]; // variabila locala
    // !!limita intervalului verificata gresit
    if (strlen(msg) <= 512)
        strcpy(buffer, msg);
}

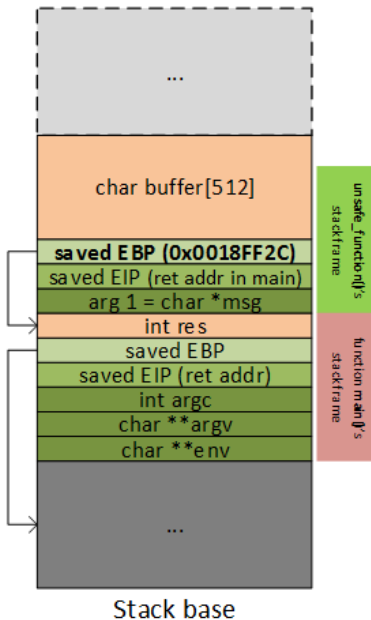
int main(int argc, char **argv, char **env) {
    int res;
    /* Buffer overflow pentru "strlen(argv[1]) >= 512" */
    res = unsafe_function(argv[1]);
    return res;
}
```

Exemplu de depășire cu 1 (suprascrisere EBP) (exemplu II)

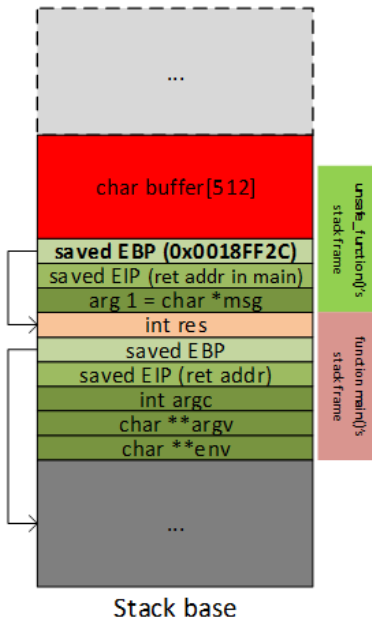
- Exemplu 2

```
    ...  
void process_string(char *src) {  
    char dest[32];  
    for (i = 0; src[i] && (i <= sizeof(dest)); i++)  
        dest[i] = src[i];  
    ...  
}
```

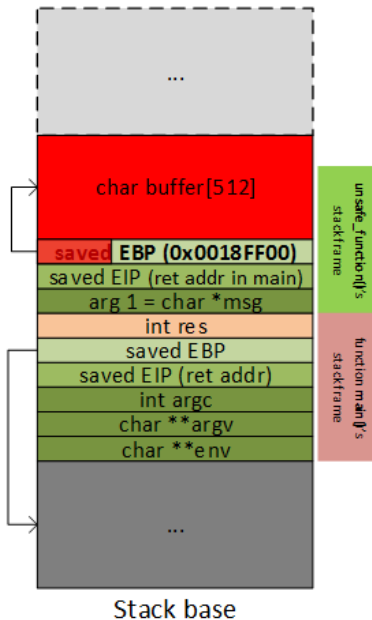
Vizualizare stiva



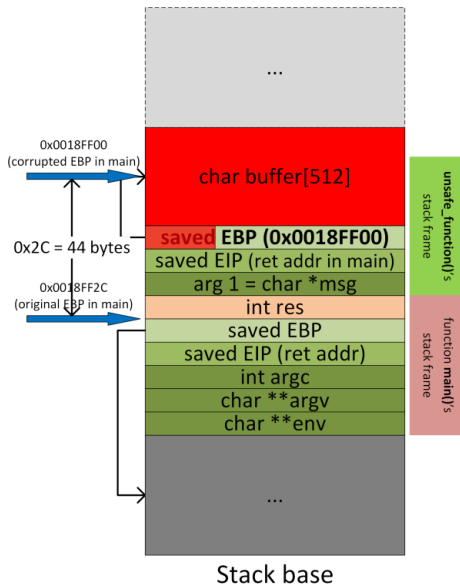
Vizualizare stiva



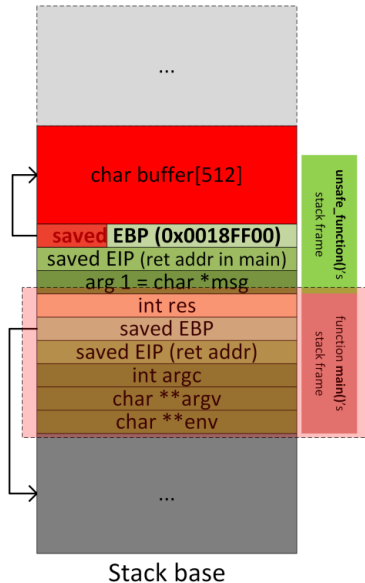
Vizualizare stiva



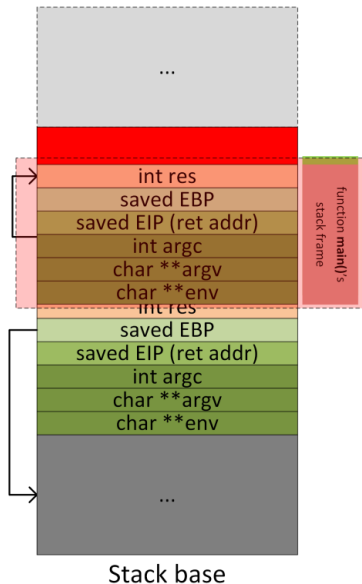
Vizualizare stiva



Vizualizare stiva



Vizualizare stiva



Atac *depasire cu 1* II - suprascrierea unei variabile locale

- depinde de modul în care se folosește variabila adiacentă după depășire
- dacă variabila suprascrisă este un întreg ce memorează o dimensiune
 - ▶ valoarea este trunchiată
 - ▶ \implies programul va face calcule greșite pe baza noii valori
- dacă variabila reprezintă un identificator de utilizator (*user ID*)
 - ▶ \implies ar putea permite programului curent să primească drepturi nepermise într-un mod normal de funcționare

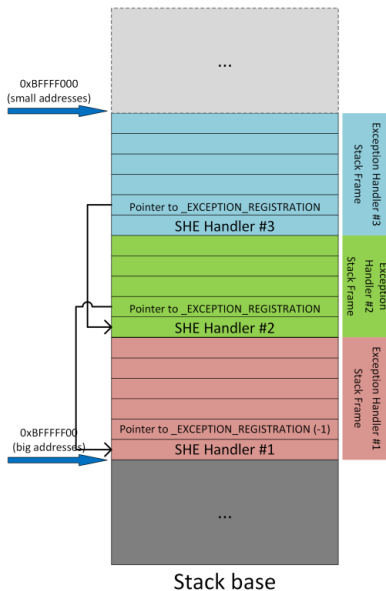
Atacuri SEH - Windows

- SEH = structured exception handlers
- *smashing SEH*
- specific Windows
 - ▶ programele ar putea înregistra *handlers* pentru a acționa asupra greșelilor
 - ▶ tratarea excepțiilor lansate de programe în timpul rulării
- stiva conține structurile ce permit înregistrarea *exception handlers*
 - ▶ adresa rutinei de tratare a excepțiilor
 - ▶ pointer catre *handler* parinte
- lanțul de tratare a excepțiilor este traversat de la cel mai nou *handler* până la primul *handler* înregistrat
 - ▶ se identifică rutina corespunzătoare erorii prin execuția fiecărei rutine

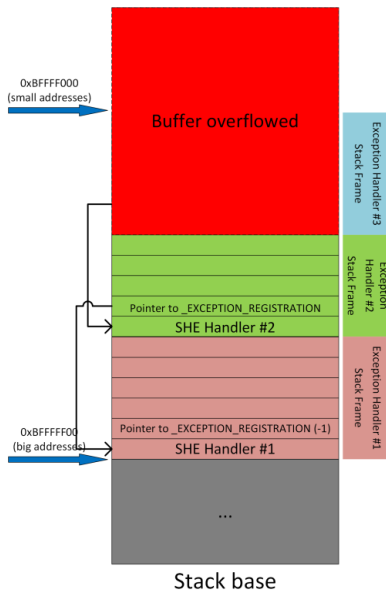
Atacuri SEH - Windows (II)

- dacă un atacator poate efectua un atac *stack overflow*
 - ▶ suprascrie structura de tratare a excepțiilor (adresa *handler*)
 - ▶ genera o excepție
 - ▶ deturna fluxul execuției (programul executa codul atacatorului)

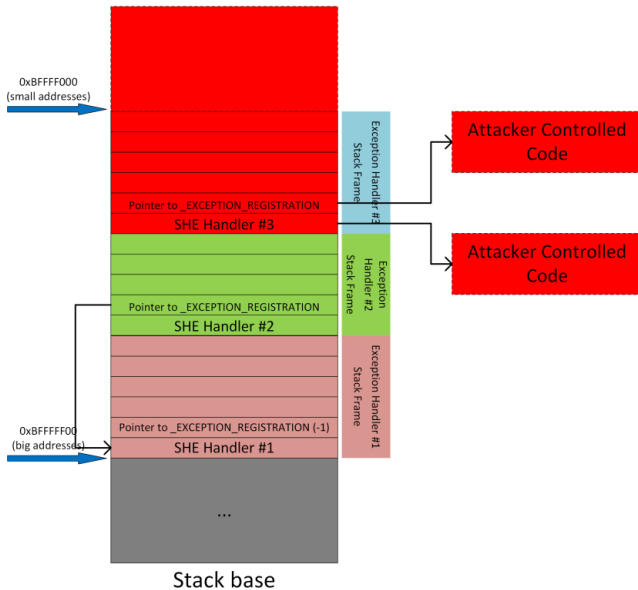
Vizualizarea unui atac SEH



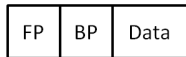
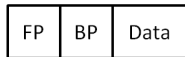
Vizualizarea unui atac SEH



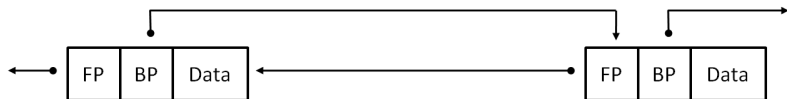
Vizualizarea unui atac SEH



Heap Overflows, application heaps - review



Heap Overflows, application heaps - review

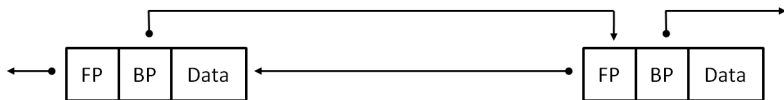


Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocate space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```

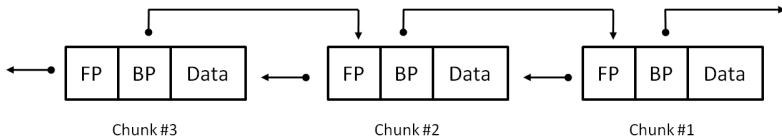


Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocate space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```



Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocates space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```

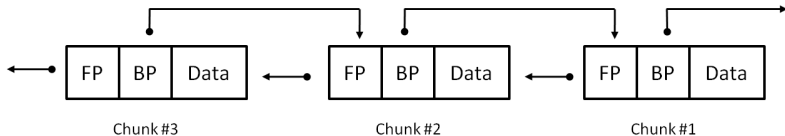
Pseudo-code For Chunk Freeing:

NextChunk = Current->FP

PreviousChunk = Current->BP

NextChunk->BP = PreviousChunk

PreviousChunk->FP = NextChunk



Heap Overflows, application heaps - review

```
void SampleFunction(void)
{
    /* Allocate space on heap */
    char * ptr = (char *)malloc(32);

    /* Operations */

    /* Free allocated heap space */
    free(ptr);
}
```

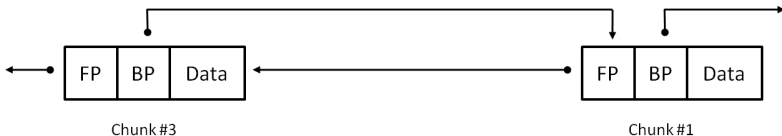
Pseudo-code For Chunk Freeing:

NextChunk = Current->FP

PreviousChunk = Current->BP

NextChunk->BP = PreviousChunk

PreviousChunk->FP = NextChunk



Heap overflows

- heap management

- ▶ *malloc* și *free*
- ▶ in-line metadata
 - ★ blocurile alocate sunt precedate de un antet ce ține informații despre bloc și vecinii săi
 - ★ câmpuri: dimensiunea blocului, dimensiunea blocului precedent, state, indicatori suplimentari
- ▶ blocurile libere sunt organizate într-o listă înlănțuită
 - ★ pointeri către elementul precedent/urmator din lista

Heap overflows - exploatare

- idee: suprascrie pointerii blocurilor libere sau dimensiunea blocului alocat
- suprascrie *heap* = urmatoarele blocuri ca fiind libere \implies sunt scoase din listă
- *overflow buffer* suprascrie pointerii listei astfel încât aceștia vor adresa locații de memorie utile atacatorului
- posibilitatea de a scrie 4 octeți oriunde în memorie (adrese de retur, pointeri, etc.)

Heap overflows - exploatare (II)

- ținte:

- ▶ global offset table (GOT)/process linkage table (PLT): folosite de executabilele ELF pentru apelul funcțiilor din biblioteci (adresa)
- ▶ exit handlers
- ▶ lock pointers
- ▶ exception handler routines
- ▶ function pointers

Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

Heap overflow - vizualizare

/* UNSAFE Function */

void UnsafeFunction(char * str)

{

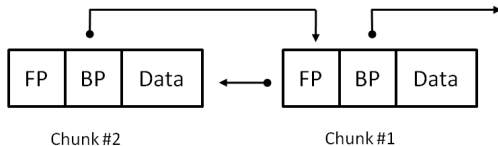
/* Allocate 32 bytes heap space */

char * Buffer = (char *)malloc(32);

/* Copy str into Buffer */

strcpy(Buffer,str);

}



Heap overflow - vizualizare

/* UNSAFE Function */

void UnsafeFunction(char * str)

{

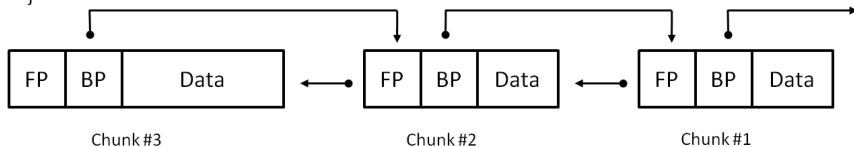
/* Allocate 32 bytes heap space */

char * Buffer = (char *)malloc(32);

/* Copy str into Buffer */

strcpy(Buffer, str);

}



Heap overflow - vizualizare

```
/* UNSAFE Function */
```

```
void UnsafeFunction(char * str)
```

```
{
```

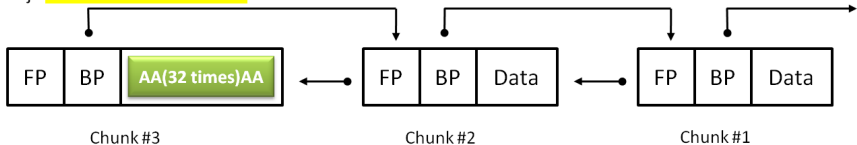
```
    /* Allocate 32 bytes heap space */
```

```
    char * Buffer = (char *)malloc(32);
```

```
    /* Copy str into Buffer */
```

```
    strcpy(Buffer,str);
```

```
}
```



Heap overflow - vizualizare

/* UNSAFE Function */

```
void UnsafeFunction(char * str)
```

```
{
```

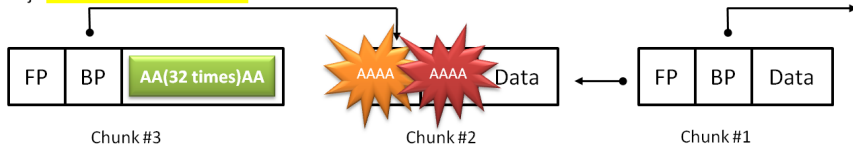
```
    /* Allocate 32 bytes heap space */
```

```
    char * Buffer = (char *)malloc(32);
```

```
    /* Copy str into Buffer */
```

```
    strcpy(Buffer,str);
```

```
}
```



Heap overflow - vizualizare

/* UNSAFE Function */

```
void UnsafeFunction(char * str)
```

```
{
```

```
    /* Allocate 32 bytes heap space */
```

```
    char * Buffer = (char *)malloc(32);
```

```
    /* Copy str into Buffer */
```

```
    strcpy(Buffer,str);
```

```
}
```

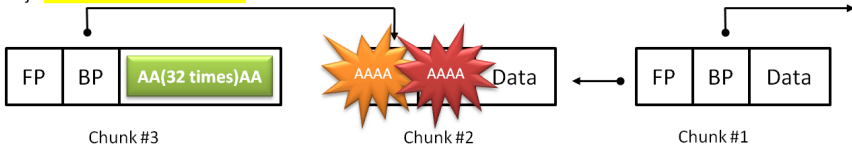
Pseudo-code For Chunk Freeing:

NextChunk = Current->FP

PreviousChunk = Current->BP

NextChunk->BP = PreviousChunk

PreviousChunk->FP = NextChunk



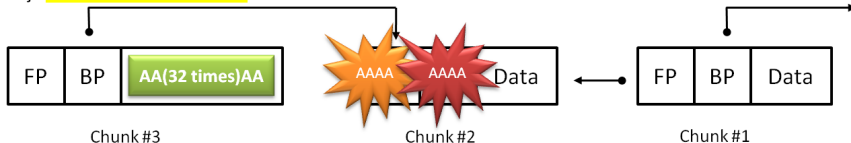
Heap overflow - vizualizare

```
/* UNSAFE Function */  
void UnsafeFunction(char * str)  
{  
    /* Allocate 32 bytes heap space */  
    char * Buffer = (char *)malloc(32);  
  
    /* Copy str into Buffer */  
    strcpy(Buffer,str);  
}
```

Pseudo-code For Chunk Freeing:

NextChunk = AAAA
PreviousChunk = AAAA

NextChunk->BP = PreviousChunk
PreviousChunk->FP = NextChunk



Heap overflow - vizualizare

/* UNSAFE Function */

```
void UnsafeFunction(char * str)
{
    /* Allocate 32 bytes heap space */
    char * Buffer = (char *)malloc(32);

    /* Copy str into Buffer */
    strcpy(Buffer,str);
}
```

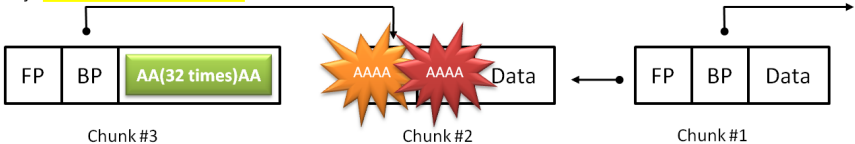
Pseudo-code For Chunk Freeing:

NextChunk = **AAAA**

PreviousChunk = **AAAA**

AAAA ->BP = **AAAA**

PreviousChunk->FP = NextChunk



Shell code

- cod "controlat" de atacator sau care poate fi folosit de atacator în scopuri specifice
- de obicei, constă în fragmente mici de cod mașină concepute pentru a executa comenzi folosind interpretorul de comenzi, pentru a crea o conexiune către atacator, etc.
- cod independent (*position independent code*) ce folosește API sistem
- apeluri sistem pot fi efectuate direct (merge de obicei sub Linux) sau prin intermediul funcțiilor din bibliotecile sistemului de operare (sub Windows)
- independent de programul care-l lansează în execuție

Example

- varianta Linux de apel a funcției sistem `execve`

```
xorl %eax, %eax ; zero out EAX
movl %eax, %edx ; EDX = envp = NULL
movl $address_of_shell_string, %ebx ; EBX = path parameter
movl $address_of_argv, %ecx ; ECX = argv
movl %$0x0B, %al ; syscall number for execve()
int $0x80 ; invoke the system call
```

- position independent code, calcul adresa argumentelor
 - ▶ adresa sirului este adresa de retur încărcată pe stivă de instrucțiunea *call*

```
jmp end
code:
... shellcode ...
end:
call code
.string "/bin/sh"
```

Exemple (II)

Codul rezultat:

```
jmp end
code:
popl %ebx ; EBX = pathname argument
xorl %eax, %eax ; zero out EAX
movl %eax, %edx ; EDX = envp
pushl %eax ; put NULL in argv array
pushl %ebx ; put "/bin/sh" in argv array
movl %esp, %ecx ; ECX = argv
movl $0x0B, %al ; 0x0B = execv() syscall
int $0x80 ; system call
end:
call code
.string "/bin/sh"
```

Eliminarea vulnerabilităților

- identificarea greșelilor
- corectarea lor
- **folosiți funcții pe șiruri sigure**, când este posibil

- ▶ *strcpy* → *strncpy* → *strncpy_s*
- ▶ *strcpy* → *strcpy_s*
- ▶ mai multe

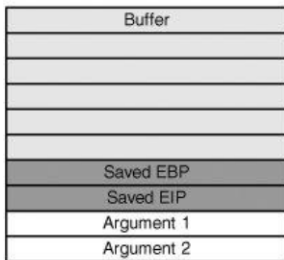
[https://msdn.microsoft.com/en-us/library/windows/hardware/ff565508\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff565508(v=vs.85).aspx)

- API sigure vs. API vechi / interzise

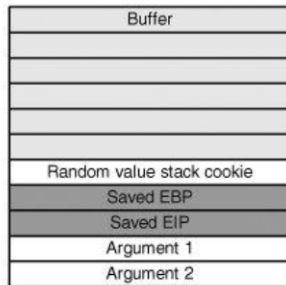
Stack Cookies (Canary values)

- detecția și prevenirea atacurilor de tipul *buffer overflow* pe stivă
- compile time prevention
- inserarea unei valori aleatoare pe stiva
 - ▶ imediat după adresa de retur si frame pointer
 - ▶ înainte de variabilele locale din cadrele stivei
- la sfârșitul funcției, înainte de retur, se generează cod pentru a verifica dacă valoarea aleatoare (*canary*) s-a modificat
 - ▶ în caz afirmativ se generează o excepție
 - ▶ altfel programul își continuă execuția

Stack Cookies (II)



Ordinary function
stack frame



Protected function
stack frame

Stack Cookies (Canary values) (III)

Limitări:

- nu protejează împotriva suprascrierii variabilelor locale
 - ▶ o soluție ar fi rearanjarea variabilelor pe stiva
- se pot pastra valorile aleatoare (*canary values*), când se poate accesa stiva
- suprascrierea parametrilor funcției
 - ▶ preluarea controlului aplicației înainte de returul funcției
 - ▶ dificil de realizat, compilatoare care salvează valoarea parametrilor și în regiștrii
- sub Windows, pointerii SEH pot fi suprascriși
- compiled-time

Protecție împotriva atacurilor heap overflow

- adaugarea de heap cookies în antetul blocurilor
- validarea operațiilor de ștergere (verificarea dacă elementul anterior și următor din listă referă elementul ce va fi șters)
- limitări:
 - ▶ protejează doar structura, nu și datele
 - ▶ algoritmi specifici de gestionare a memoriei construiți pe baza algoritmilor sistem

Non-Executable Stack and Heap Protection

- Data Execution Protection (DEP)
- nega (dacă procesorul permite) dreptul de execuție (non-execute / NX) în paginile de memorie ce conțin date (stiva sau heap)
- limitări:
 - ▶ nu protejează împotriva întoarcerii controlului în zone de cod utile
 - ▶ **deactivate NX**: funcția reîntoarce controlul în funcții existente ce permit execuția în anumite zone de memorie (controlate de atacator)
 - ▶ **return-into-libc**: reîntoarce controlul în funcții existente ce rulează codul existent în avantajul atacatorului
 - ▶ **return-oriented-programming (ROP)**: încarcă mai multe adrese de retur pe stiva (*gadgets*) care înlănțuite execută cod în avantajul atacatorului

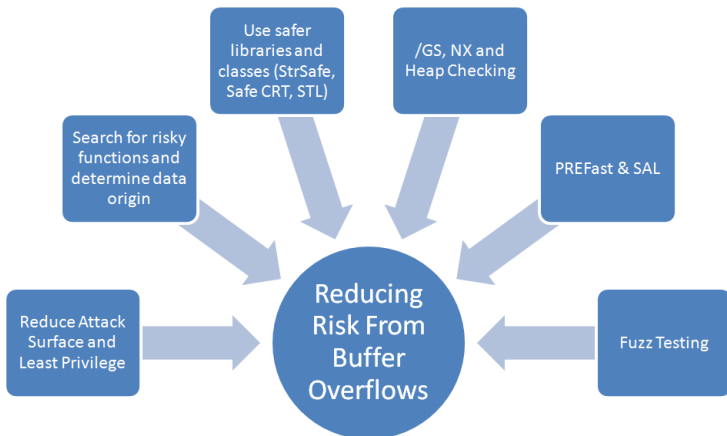
Address-Space Layout Randomization (ASLR)

- majoritatea atacurilor de tipul *buffer overflow* se bazează pe adrese de memorie cunoscute
- funcționează împotriva atacatorilor care folosesc adrese de memorie predeterminate
 - ▶ adrese folosite pentru a găsi *gadgets* în atacuri ROP
- aleatorizare unde codul și datele aplicației (inclusiv bibliotecile comune) sunt mapate la execuție în spațiul de adrese
- limitări:
 - ▶ date fixe în memorie ce nu pot fi realocate de mecanismul ASLR
 - ★ structuri de date specializate
 - ★ loader
 - ★ biblioteci non-relocabile
 - ▶ tehnici *brute force* pot fi folosite pentru a găsi adrese utile

SEH sigur sub Windows

Adresa handler de tratare a excepției este verificată înainte de a apela rutina

Microsoft SDL (Security Development Lifecycle)



SDL: Examinarea codului sursă

- Revizuirea codului sursă: inspectarea manuală a aplicației pentru vulnerabilități specifice, ex. *buffer overflow*
 - ▶ datele primite din rețea, fișiere, linia de comandă
 - ▶ transferul datelor către structurile interne
- **metoda generală**: urmăriți datele primite de la utilizator din punctul de intrare în aplicației prin toate apelurile de funcții

Run-time protection

- **compiler protection:** run-time checks

Instrumente pentru analiza codului sursa

- instrumente automate ce ajută în identificarea vulnerabilităților cunoscute

Fuzz testing

- metoda de testare ce ajută în identificarea problemelor de securitate ce se manifestă datorită nevalidării datelor introduse de utilizator

Example

```
void process_string(char *src)
{
    char dest[32];

    for (i = 0; src[i] && (i <= sizeof(dest)); i++)
        dest[i] = src[i];
}
```

Example

```
void process_string(char *src)
{
    char dest[32];

    for (i = 0; src[i] && (i < sizeof(dest)); i++)
        dest[i] = src[i];
}
```

Exemple

```
int setFilename(char *filename) {  
    char name[20];  
    sprintf(name, "%16s.dat", filename);  
    int success = saveFormattedFilenameToDB(name);  
    return success;  
}
```

Example

```
...  
char source[21] = "the character string";  
char dest[12];  
strncpy(dest, source, sizeof(source)-1);  
...
```

Example

```
...  
char source[21] = "the character string";  
char dest[12];  
strncpy(dest, source, sizeof(dest)-1);
```

Example

```
...  
char source[21] = "the character string";  
char dest[12];  
strncpy(dest, source, sizeof(dest)-1);  
dest[sizeof(dest)-1] = '\\0';  
...
```


Exemple

- `returnChunkSize()` returns “-1” on error
- the return value is not checked before the `memcpy` operation
- `memcpy()` assumes that the value is unsigned
- when “-1” is returned, it will be interpreted as MAXINT-1 (e.g. 0xFFFFFFFF)

```
int returnChunkSize(void *chunk) {  
    /* if chunk info is valid, return the size of usable memory,  
     * else, return -1 to indicate an error  
     */  
    ...  
}  
  
int main() {  
    ...  
    memcpy(destBuf, srcBuf,  
           (returnChunkSize(destBuf) - 1));  
    ...  
}
```

```
#include <string.h>  
void *memcpy(void *dest, const void *src, size_t n);
```

Exemple

```
int *id_sequence;  
  
/* Allocate space for an array of three ids. */  
  
id_sequence = (int*) malloc(3);  
if (id_sequence == NULL) exit(1);  
  
/* Populate the id array. */  
  
id_sequence[0] = 13579;  
id_sequence[1] = 24680;  
id_sequence[2] = 97531;
```

Exemple

```
int *id_sequence;

/* Allocate space for an array of three ids. */

id_sequence = (int*) malloc(3 * sizeof(int*));
if (id_sequence == NULL) exit(1);

/* Populate the id array. */

id_sequence[0] = 13579;
id_sequence[1] = 24680;
id_sequence[2] = 97531;
```

CWE Buffer-Overflow

- CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
- CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
 - ▶ rang 3 in top 25
- CWE-121: Stack-based Buffer Overflow
- CWE-122: Heap-based Buffer Overflow
- CWE-124: Buffer Underwrite ('Buffer Underflow')
- CWE-125: Out-of-bounds Read
- CWE-131: Incorrect Calculation of Buffer Size (!)
 - ▶ rang 20 in top 25
- CWE-170: Improper Null Termination

CWE Buffer-Overflow (II)

- CWE-190: Integer Overflow (!)
 - ▶ rang 24 in top 25
- CWE-193: Off-by-one Error
- CWE-805: Buffer Access with Incorrect Length Value

Example

- Internet worm: Morris finger worm (1988)
- Common Vulnerabilities and Exposures (<https://cve.mitre.org/find/index.html>)
 - ▶ aproximativ 26000 de rezultate pentru o căutare după cuvintele "buffer overflow"
- Vulnerability Notes Database (<https://www.kb.cert.org/vuls/>)
- CVE-2015-0235 - GHOST: glibc gethostbyname buffer overflow
- CVE-2014-0001 - Buffer overflow in *client/mysql.cc* in Oracle MySQL and MariaDB before 5.5.35
- CVE-2014-0182 - Heap-based buffer overflow in the *virtio_load* function in *hw/virtio/virtio.c* in QEMU before 1.7.2

Example (II)

- CVE-2014-0498 - Stack-based buffer overflow in Adobe Flash Player before 11.7.700.269
- CVE-2014-0513 - Stack-based buffer overflow in Adobe Illustrator CS6 before 16.0.5
- CVE-2014-8271 - Tianocore UEFI implementation reclaim function vulnerable to buffer overflow
- CVE-2013-0002 - Buffer overflow in the Windows Forms (aka WinForms) component in Microsoft .NET Framework
- CVE-2005-3267 - Integer overflow in Skype client leads to a resultant heap-based buffer overflow

Bibliografie

- “The Art of Software Security Assessments”, chapter 5, “Memory Corruption”, pp. 167 – 202
- Interactive: The Top Programming Languages 2015,
[http://spectrum.ieee.org/static/
interactive-the-top-programming-languages-2015](http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015)
- CWE-119 - Buffer Errors