SOFTWARE (IN)SECURITY

Laboratorio del corso di sicurezza II

Valerio Costamagna - @vaioco

November 21, 2015

Universitá degli studi di Torino

Headline del corso

- Architettura IA32
- Tipologie di bugs
- Focus on: Stack and Heap overflow
- Modern mitigation and protection

Preparazione ambiente

- Linux
- apt-get install binutils build-essentials ia32-libs
- hopperapp disassembler [1]
- checksec.sh [2]
- GDB: Gnu Debugger

[1] http://www.hopperapp.com/
[2] http://www.trapkit.de/tools/checksec.html

Lezione 1

1. richiami architettura IA32/x86
2. principali istruzioni assembler
3. stack
4. tools and code!

IA32

Architettura IA32/x86

- Intel architecture: versione 32 bit del x86 Instruction Set Architecture (ISA)
- arch CISC: istruzioni di lunghezza variabile, diverse istruzioni per accedere/modificare la memoria
- 8 registri 32 bit *general purpose*: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- 6 registri 16 bit *segment register*: CS, DS, ES, FS, GS, SS
- diversi registri speciali
- little endian

Table: Registri principali

Nome	Descrizione
EBP	stack pointer
ESP	frame pointer
EIP	instruction pointer
EFLAGS	*bit flags*
ESI	source string/mem operations
EDI	dest string/mem operations
ECX	counter in loops
EAX	valore di return di una funzione

Nomenclatura

ABI

Application Binary Interface: descrive l'interfaccia tra il SO e i binari su una particolare architettura. Determina la *calling convention* (come vengono chiamate le funzioni), come vengono gestite le syscall, etc...

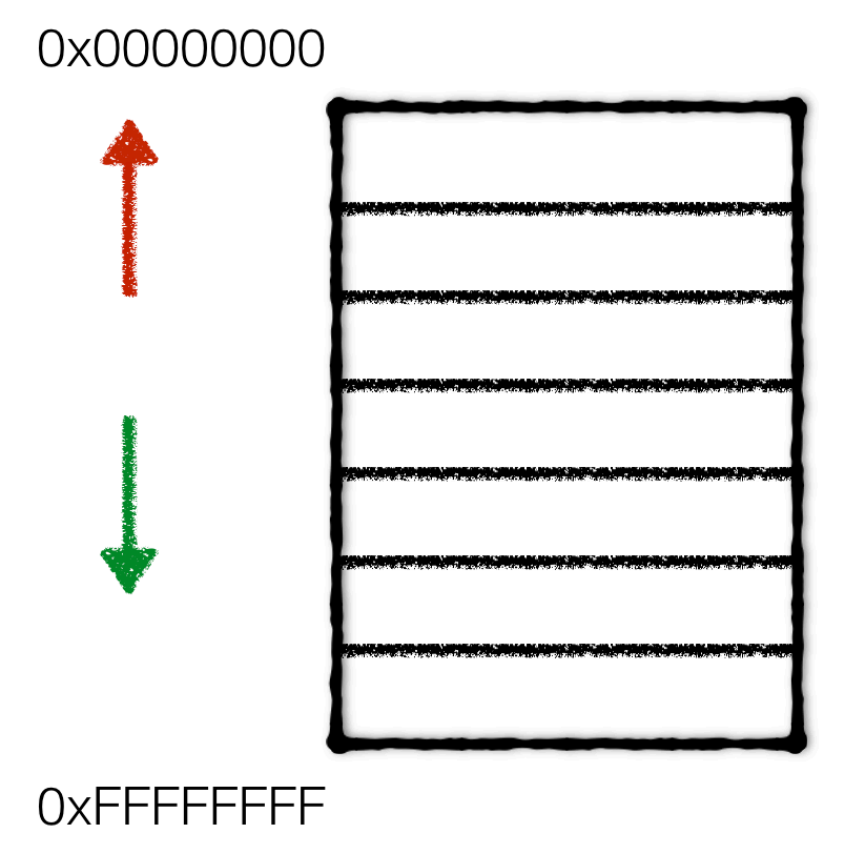
ILP32

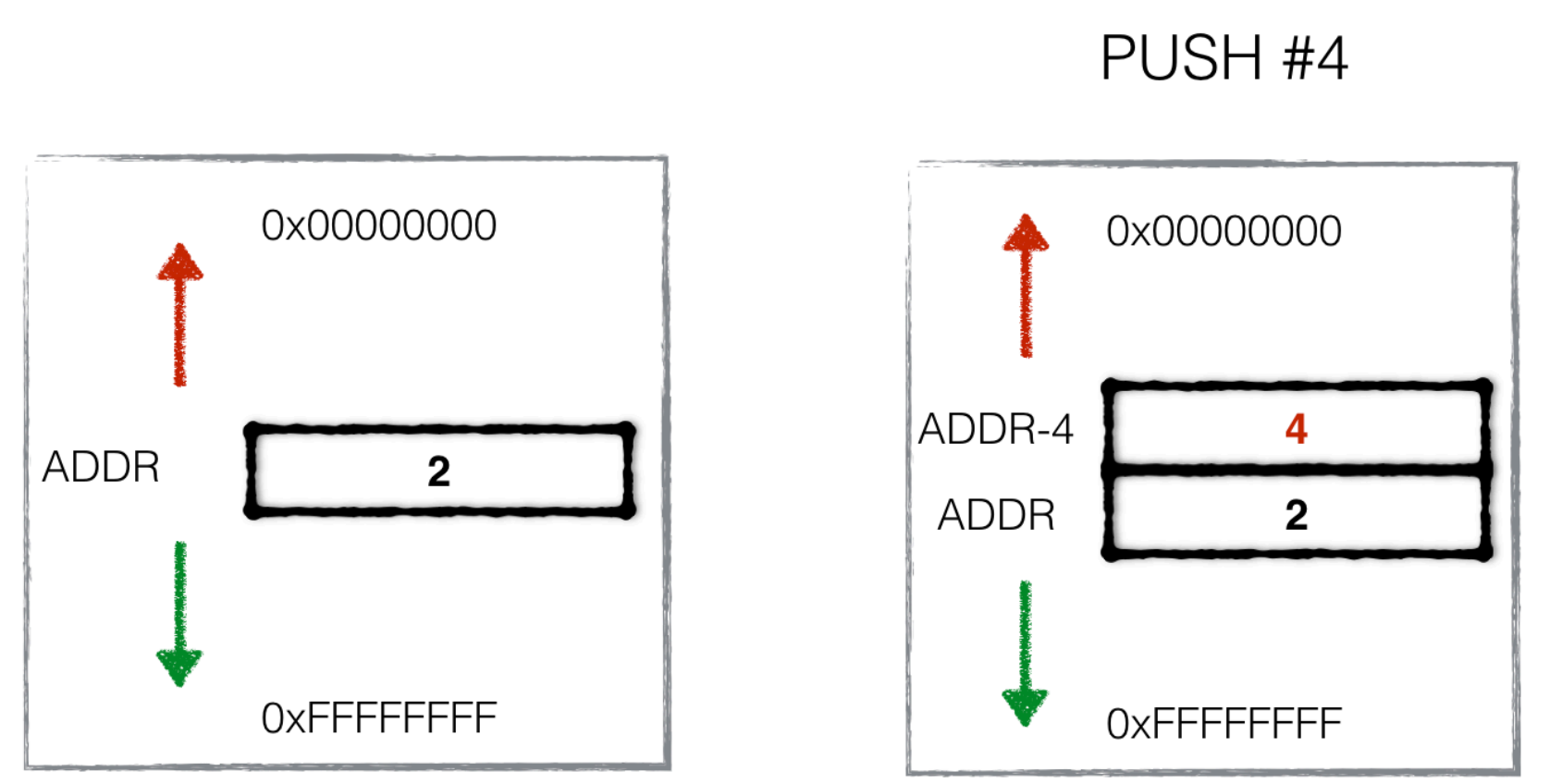
Data model in cui i tipi *Int = long = pointer* occupano 32 bit di memoria

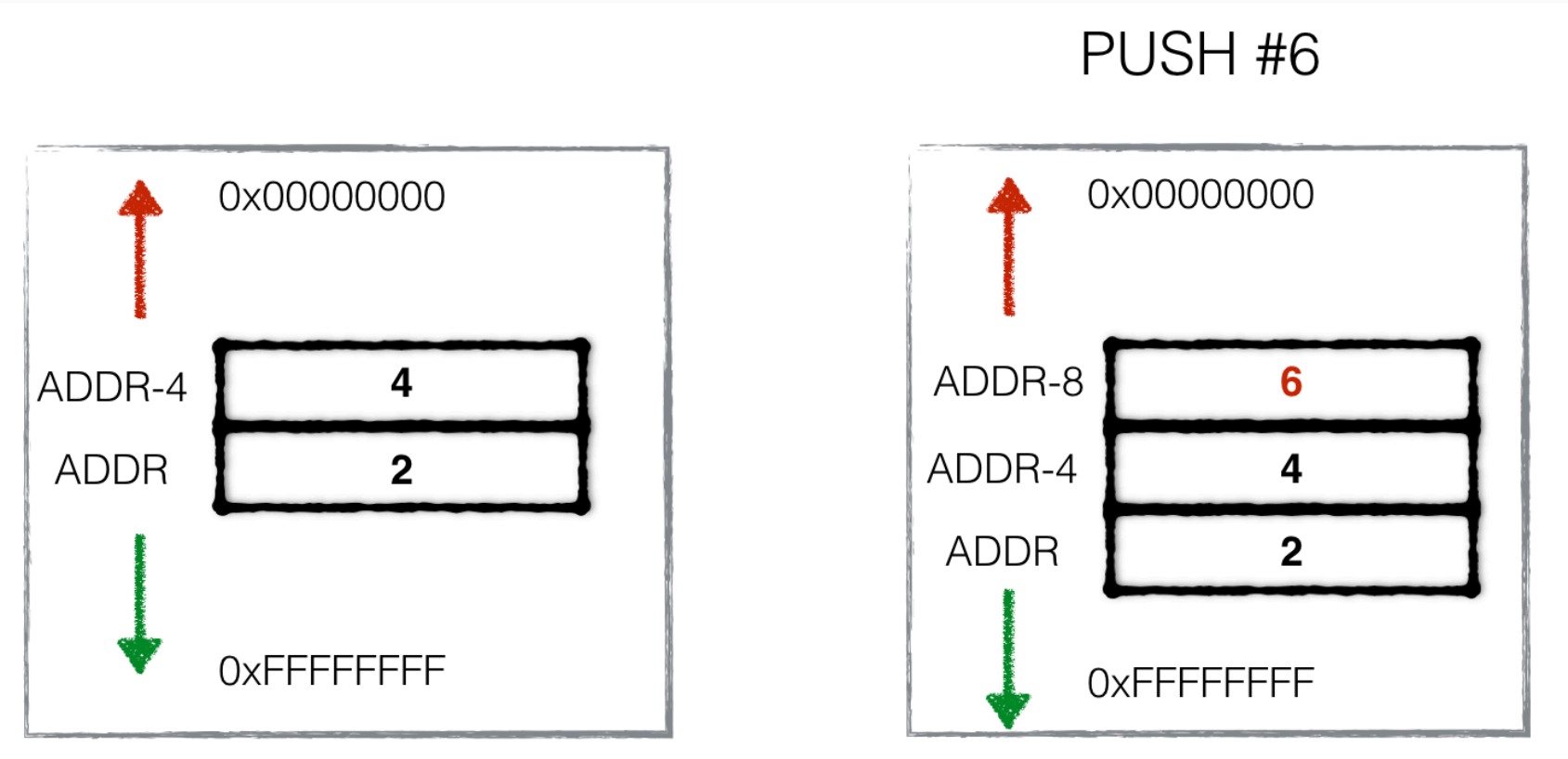
STACK

Stack

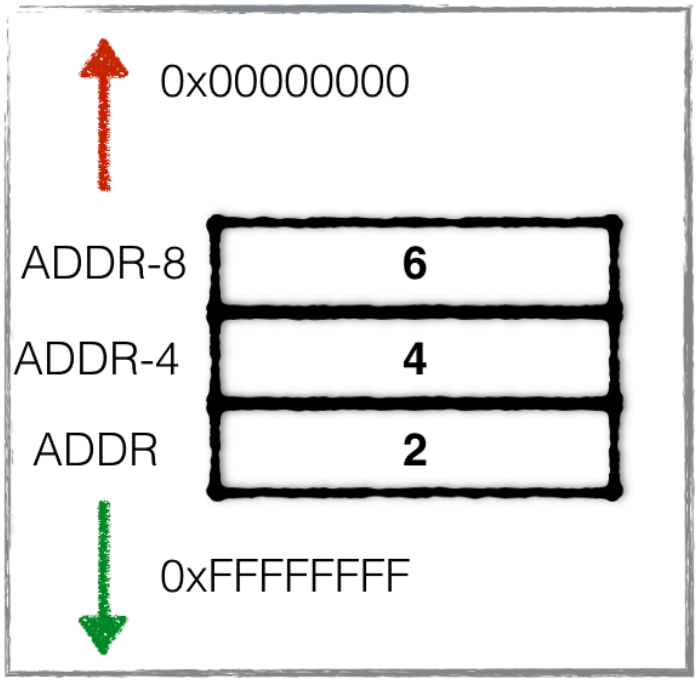
- Struttura dati usata per salvare ed ottenere una serie di elementi
- Last In First Out (LIFO)
- Cresce verso indirizzi di memoria bassi!
- Due operazioni possibili:
  - PUSH() : inserisce un elemento in cima (top)
  - POP() : ritorna e rimuove l'elemento dalla cima (top)

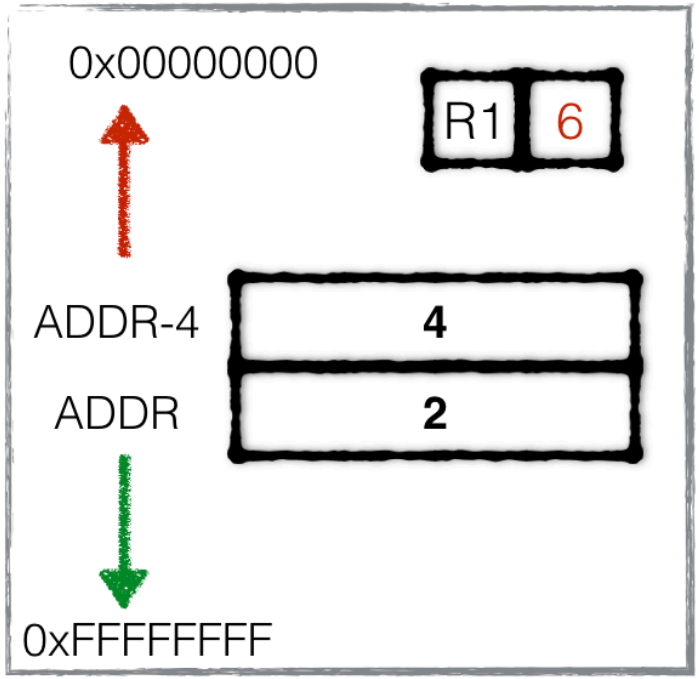



PUSH #4
0x00000000
ADDR
2
0xFFFFFFFF
0x00000000
ADDR-4
4
ADDR
2
0xFFFFFFFF

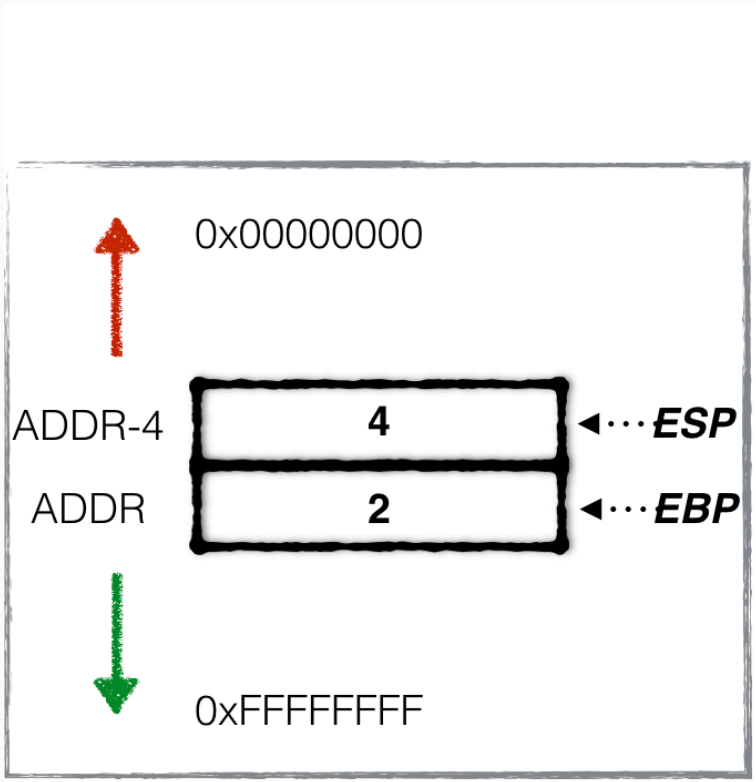

PUSH #6
0x00000000
ADDR-4
4
ADDR
2
0xFFFFFFFF
0x00000000
ADDR-8
6
ADDR-4
4
ADDR
2
0xFFFFFFFF

POP R1

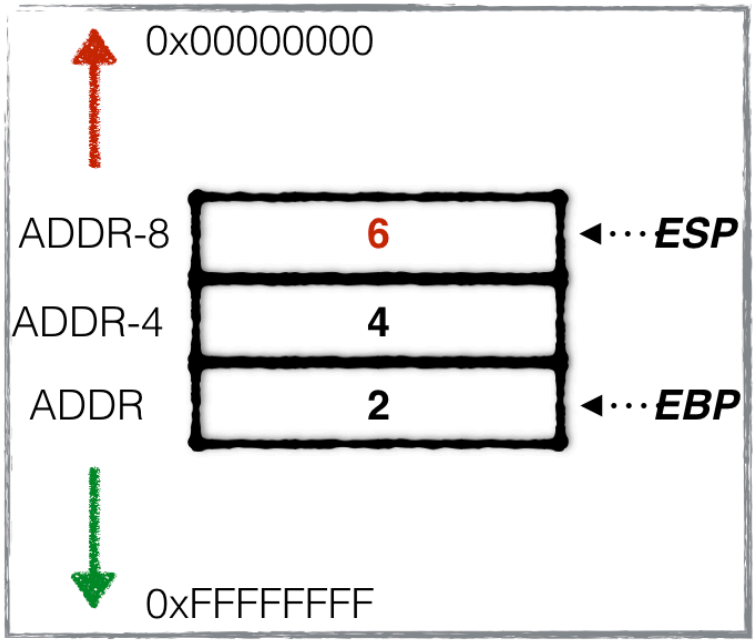

0x00000000
ADDR-8
6
ADDR-4
4
ADDR
2
0xFFFFFFFF

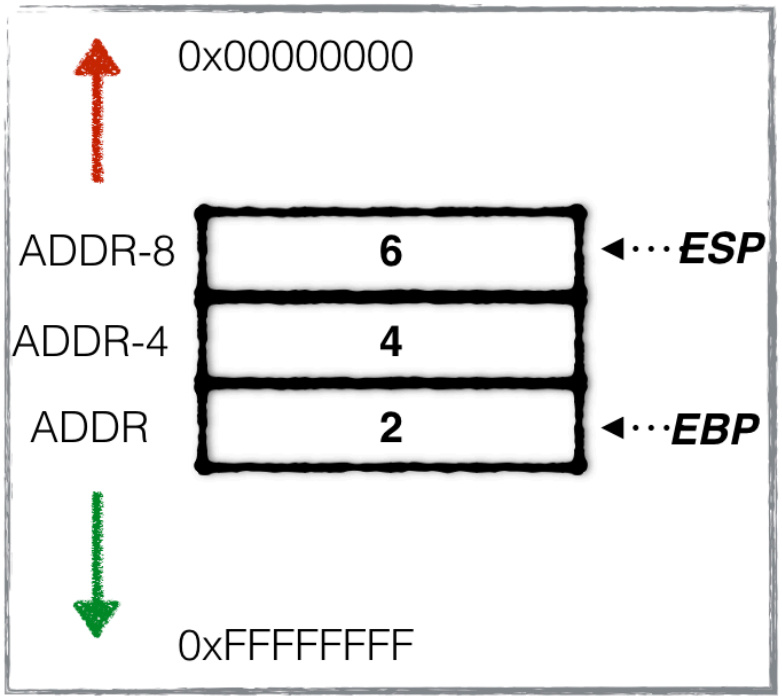

0x00000000
R1
6
ADDR-4
4
ADDR
2
0xFFFFFFFF

Stack pointer e Base Pointer Registers

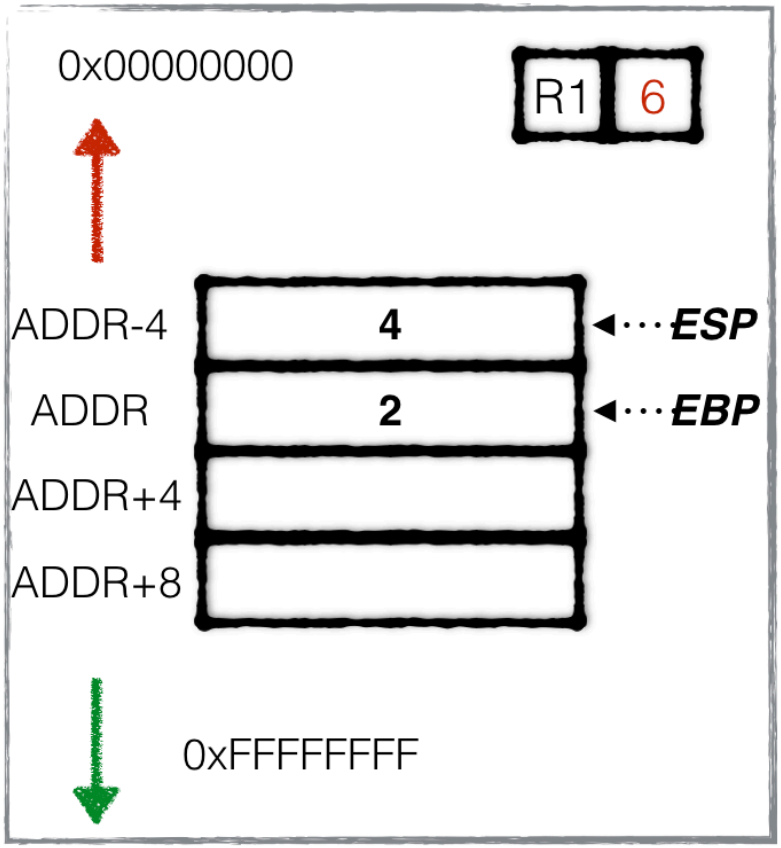


PUSH #6




0x00000000
ADDR-8
6
ESP
ADDR-4
4
ADDR
2
EBP
0xFFFFFFFF

POP R1


0x00000000
R1
6
ADDR-4
4
ESP
ADDR
2
EBP
ADDR+4
ADDR+8
0xFFFFFFFF

Ogni funzione ha un proprio *stack frame*, compreso tra EBP e ESP, in cui memorizza:

- variabili locali
- machine state (indirizzo di ritorno, old EBP)
- parametri delle chiamate a funzioni

Il registro EBP viene utilizzato (se non specificato diversamente a compile time) come *base* per accedere alle variabili locali dello *stack frame*.

```
1  int function_B(int a, int b)
2  {
3      int x, y;
4      x = a * a;
5      y = b * b;
6      return (x + y);
7  }
8  int function_A(int p, int q)
9  {
10     int c;
11     c = function_B(p,q);
12     return c;
13 }
14 int main(int argc, char** argv, char** envp)
15 {
16     int ret;
17     ret = function_A(1,2);
18     return ret;
19 }
```

Stack

function_B code:

```
1  int function_B(int a, int b)
2  {
3  int x, y;
4      x = a * a;
5      y = b * b;
6      return x + y;
7  }
8
```

Assembler code:

```
1  <+0>:  push   ebp
2  <+1>:  mov    ebp,esp
3  <+3>:  sub    esp,0x10
4  <+6>:  mov    eax,DWORD PTR [ebp+0x8]
5  <+9>:  imul   eax,DWORD PTR [ebp+0x8]
6  <+13>: mov    DWORD PTR [ebp-0x4],eax
7  <+16>: mov    eax,DWORD PTR [ebp+0xc]
8  <+19>: imul   eax,DWORD PTR [ebp+0xc]
9  <+23>: mov    DWORD PTR [ebp-0x8],eax
10 <+26>: mov    eax,DWORD PTR [ebp-0x8]
11 <+29>: mov    edx,DWORD PTR [ebp-0x4]
12 <+32>: add    eax,edx
13 <+34>: leave
14 <+35>: ret
15
```

Stack

function_A code:

```
1 int function_A(int p, int q){
2 int c;
3     c = function_B(p,q);
4     return c;
5 }
6
```

Assembler code:

```
1  <+0>:  push   ebp
2  <+1>:  mov    ebp,esp
3  <+3>:  sub    esp,0x18
4  <+6>:  mov    eax,DWORD PTR [ebp+0xc]
5  <+9>:  mov    DWORD PTR [esp+0x4],eax
6  <+13>: mov    eax,DWORD PTR [ebp+0x8]
7  <+16>: mov    DWORD PTR [esp],eax
8  <+19>: call   0x80483ed <function_B>
9  <+24>: mov    DWORD PTR [ebp-0x4],eax
10 <+27>: mov    eax,DWORD PTR [ebp-0x4]
11 <+30>: leave
12 <+31>: ret
13
```

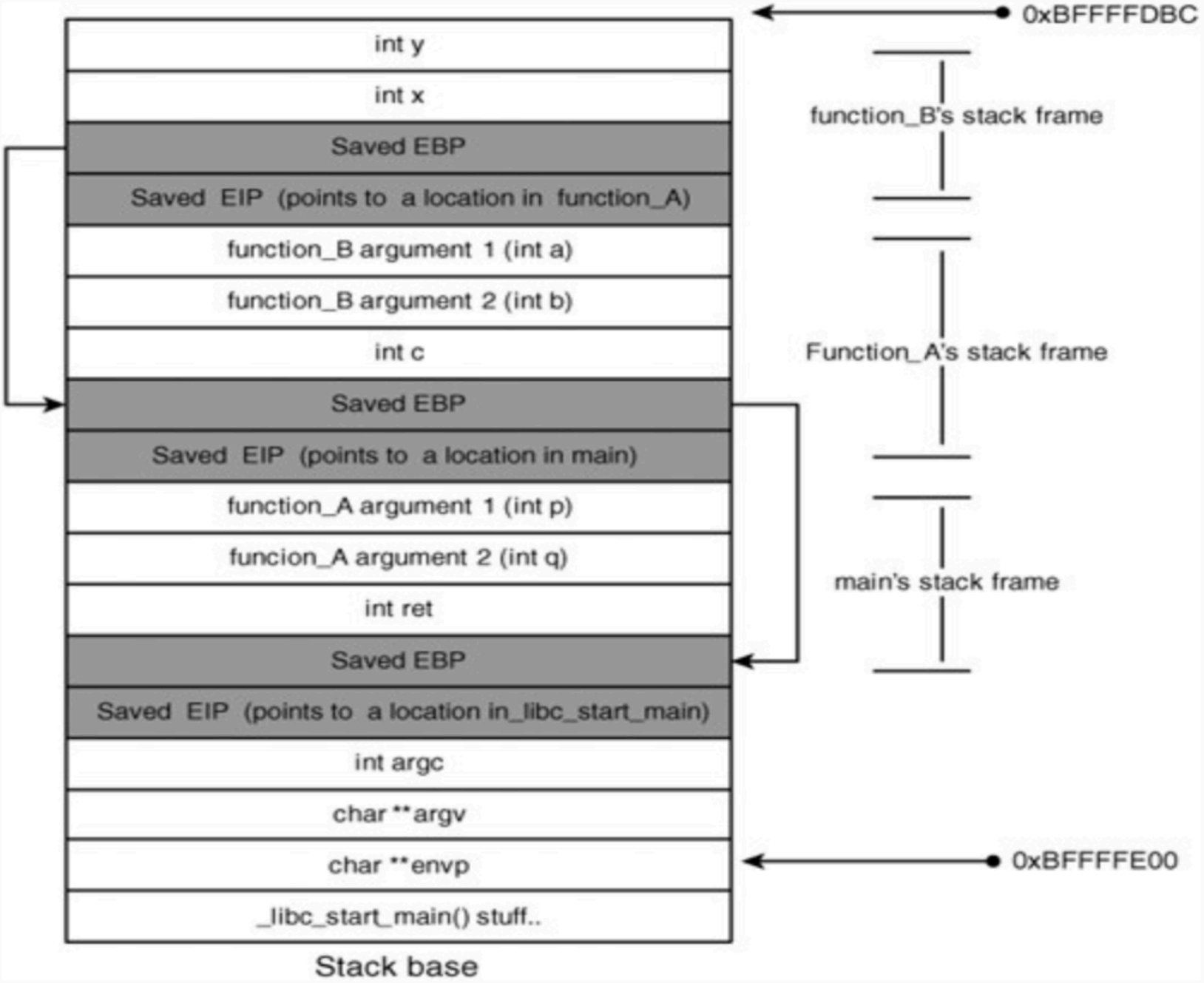
Stack

main code:

```
1  int main(int argc, char** argv, char**
       envp){
2  int ret;
3      ret  = function_A(1,2);
4      return ret;
5  }
6
```

Assembler code:

```
1  <+0>:  push   ebp
2  <+1>:  mov    ebp,esp
3  <+3>:  sub    esp,0x18
4  <+6>:  mov    DWORD PTR [esp+0x4],0x2
5  <+14>: mov    DWORD PTR [esp],0x1
6  <+21>: call   0x8048411 <function_A>
7  <+26>: mov    DWORD PTR [ebp-0x4],eax
8  <+29>: mov    eax,DWORD PTR [ebp-0x4]
9  <+32>: leave
10 <+33>: ret
11
```


0xBFFFFDBC
int y
int x
Saved EBP
Saved EIP (points to a location in function_A)
function_B argument 1 (int a)
function_B argument 2 (int b)
int c
Saved EBP
Saved EIP (points to a location in main)
function_A argument 1 (int p)
funcion_A argument 2 (int q)
int ret
Saved EBP
Saved EIP (points to a location in_libc_start_main)
int argc
char **argv
char **envp
_libc_start_main() stuff..
Stack base
function_B's stack frame
Function_A's stack frame
main's stack frame
0xBFFFFE00

Function Prologue

```
1 <+0>:  push   ebp
2 <+1>:  mov    ebp,esp
3 <+3>:  sub    esp,VALORE
```

Function Epilogue

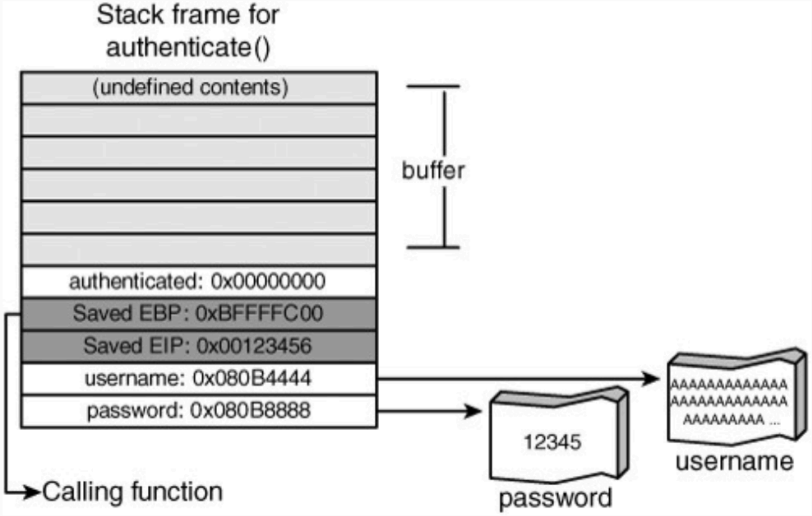
```
1 mov   esp, ebp
2 pop   ebp
3 ret
```

Stack abuse

```
1  int authenticate(char *username, char*
       password)
2  {
3  int authenticated;
4     char buffer[1024];
5
6     authenticated = verify_password(username,
       password);
7
8     if(authenticated == 0){
9      sprintf(buffer, "password is incorrect for user:
         %s \n" , username);
10       log("%s", buffer);
11    }
12    return authenticated;
13 }
```

Stack abuse

```
1  int authenticate(char *username, char*
       password)
2  {
3  int authenticated;
4     char buffer[1024];
5
6     authenticated = verify_password(username,
       password);
7
8     if(authenticated == 0){
9      sprintf(buffer, "password is incorrect for user:
         %s \n" , username);
10       log("%s", buffer);
11    }
12    return authenticated;
13 }
```



BUFFER OVERFLOW

Un **buffer overflow** é un bug che si verifica quando dati copiati in una locazione di memoria eccedono la grandezza riservata per tale variabile.
Quando si verifica un overflow, i dati in eccesso vengono copiati nelle locazioni di memoria adiacenti.
I buffer overflow sono i tipi piú comuni di **memory corruption**.

Stack Overflow - Can you spot the bug ?

```
1 | int main() {
2 | int cookie;
3 | char buf[80];
4 | printf("buf: %08x cookie: %08x\n", &buf, &cookie);
5 | gets(buf);
6 | if (cookie == 0x41424344)
7 | printf("you win!\n");
8 | }
```

Esercizio 1

Descrizione

Scaricare il materiale del corso dal repository `https://github.com/vaioco/sicII` e seguire le istruzioni contenute nel file README

Summary

Get the source of this course from

`https://github.com/vaioco/sicII`

Grazie per l'attenzione