



UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE
DIPARTIMENTO DI INFORMATICA

Reverse engineering: *executable file format*

Srdjan Matic <srđjan@security.di.unimi.it>
Aristide Fattori <aristide@security.di.unimi.it>

A.A. 2013–2014

Reverse engineering

reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made

- contesti differenti (e.g., elettronica, informatica, . . .)

Reverse engineering

reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made

- contesti differenti (e.g., elettronica, informatica, . . .)

Software reverse engineering: *perché?*

- studiare architettura interna (dal sorgente)
- ricostruire codice sorgente (dal binario)
- modificare comportamento (programmi *closed-source*)
- comprendere l'attività di rete (protocolli proprietari)

Applicazioni

- plagiarismo
- interoperabilità. Alcuni esempi:
 - Samba → protocollo SMB
 - WINE → Windows API
 - ReactOS → Windows XP/2003
 - OpenOffice → Microsoft Office
- malware analysis
- *cracking*

Come?

- 1 tecniche **statiche**
- 2 tecniche **dinamiche**

Tecniche statiche

il programma *non* viene eseguito

- documentazione
- codice sorgente
- analisi di stringhe, simboli, funzioni di libreria
- **disassembly** e analisi codice assembly

Tecniche dinamiche

esecuzione (monitorata) dell'applicazione

- interazioni con l'ambiente (e.g., file system, rete, registro)
- interazioni con il sistema operativo (system call)
- **debugging**

Background

I formati eseguibili

- file eseguibili “moderni” hanno una struttura complessa
 - Linux → Executable and Linkable Format (**ELF**)
 - Windows → Portable Executable (**PE**)

ELF: Executable and Linkable Format

Tipologie di file ELF

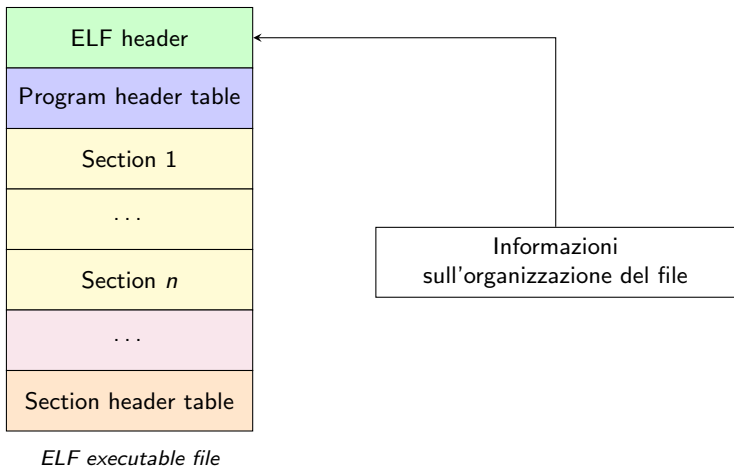
- 1 *relocatable*
- 2 *executable*
- 3 *shared object*

Esempio

```
sicurezza@sicurezza:/tmp$ gcc -c test.c -o test.o
sicurezza@sicurezza:/tmp$ file test.o
test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV) ...
sicurezza@sicurezza:/tmp$ gcc test.c -o test
sicurezza@sicurezza:/tmp$ file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV) ...
sicurezza@sicurezza:/tmp$ gcc -shared test.c -o test.so
sicurezza@sicurezza:/tmp$ file test.so
test.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV) ...
```

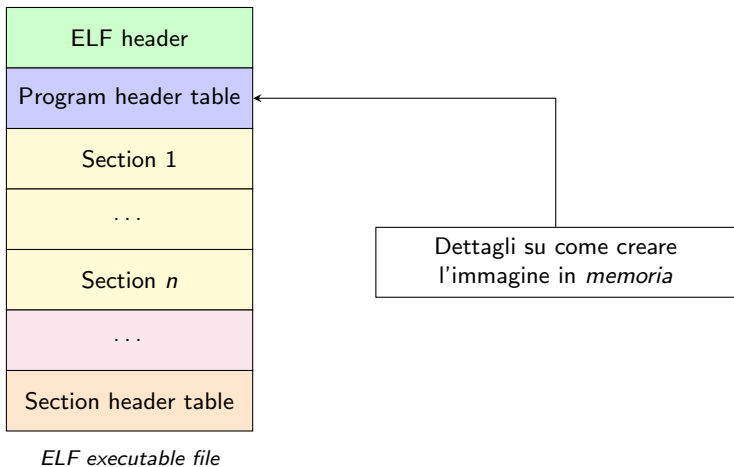

ELF: Executable and Linkable Format

Struttura file ELF



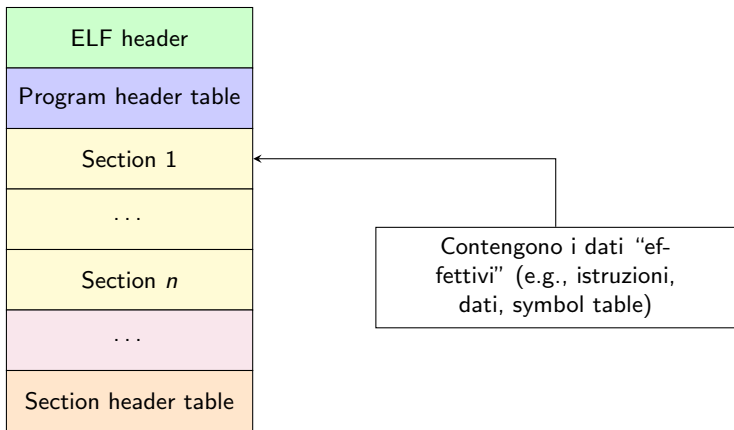
ELF: Executable and Linkable Format

Struttura file ELF



ELF: Executable and Linkable Format

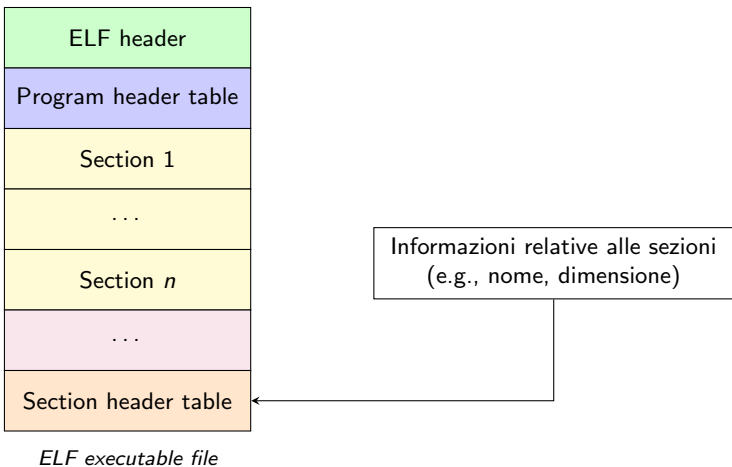
Struttura file ELF



ELF executable file

ELF: Executable and Linkable Format

Struttura file ELF



ELF: Executable and Linkable Format

ELF header

Elf32_Ehdr in `/usr/include/linux/elf.h`

- magic number (`\x7fELF`), ELF type & version
- architettura & *endianness*
- entry point (virtual address)
- offset (nel file) di program & section header table
- ...

Esempio

```
sicurezza@sicurezza:/tmp$ readelf -h test
```

```
ELF Header:
```

```
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
```

```
  ...
```

```
  Type:                                EXEC (Executable file)
```

```
  Machine:                              Intel 80386
```

```
  Version:                               0x1
```

```
  Entry point address:                   0x80482f0
```

```
  ...
```

ELF: Executable and Linkable Format

Section header table

Array di `Elf32_Shdr` in `/usr/include/linux/elf.h`

specifica, per ogni sezione

- nome (i.e., indice nella section header string table)
- posizione nel file e memoria
- dimensione
- ...

Esempio

```
sicurezza@sicurezza:/tmp$ objdump -h test
Idx Name           Size      VMA       LMA       File off  Algn
  0 .interp         00000013  08048114  08048114  00000114  2**0
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
  ...
 12 .text           0000017c  080482f0  080482f0  000002f0  2**4
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  ...
 22 .data           00000008  080495a4  080495a4  000005a4  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 23 .bss            00000008  080495ac  080495ac  000005ac  2**2
                   ALLOC
```

ELF: Executable and Linkable Format

Program header table

Array di `Elf32_Phdr` in `/usr/include/linux/elf.h`

array di strutture, ciascuna delle quali definisce

- un segmento (i.e., 1⁺ sezioni in memoria); oppure,
- altre informazioni necessarie per organizzare l'immagine in memoria

Esempio

```
sicurezza@sicurezza:/tmp$ objdump -p test
```

```
Program Header:
```

```
  PHDR off 0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
        filesz 0x000000e0 memsz 0x000000e0 flags r-x
  INTERP off 0x00000114 vaddr 0x08048114 paddr 0x08048114 align 2**0
        filesz 0x00000013 memsz 0x00000013 flags r--
  LOAD  off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
        filesz 0x000004a4 memsz 0x000004a4 flags r-x
  LOAD  off 0x000004a4 vaddr 0x080494a4 paddr 0x080494a4 align 2**12
        filesz 0x00000108 memsz 0x00000110 flags rw-
  DYNAMIC off 0x000004b8 vaddr 0x080494b8 paddr 0x080494b8 align 2**2
        filesz 0x000000d0 memsz 0x000000d0 flags rw-
  NOTE  off 0x00000128 vaddr 0x08048128 paddr 0x08048128 align 2**2
        filesz 0x00000020 memsz 0x00000020 flags r--
  STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
        filesz 0x00000000 memsz 0x00000000 flags rw-
```

ELF: Executable and Linkable Format

Symbol table

- sezione con una entry per simbolo
- debugging, dynamic linking, relocation, ...

Esempio

```
sicurezza@sicurezza:/tmp$ readelf -s test
```

```
Symbol table '.dynsym' contains 5 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
...							
3:	00000000	460	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.0 (2)
4:	0804848c	4	OBJECT	GLOBAL	DEFAULT	15	_IO_stdin_used

```
Symbol table '.symtab' contains 74 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	08048114	0	SECTION	LOCAL	DEFAULT	1	
...							
70:	080495ac	0	NOTYPE	GLOBAL	DEFAULT	ABS	_edata
71:	0804843a	0	FUNC	GLOBAL	HIDDEN	13	__i686.get_pc_thunk.bx
72:	080483a4	38	FUNC	GLOBAL	DEFAULT	13	main
73:	08048274	0	FUNC	GLOBAL	DEFAULT	11	_init

- sezione con una entry per simbolo
- debugging, dynamic linking, relocation, ...

Esempio

```
sicurezza@sicurezza:/tmp$ readelf -S test | grep '\.symtab'
[34] .symtab          SYMTAB    ...
sicurezza@sicurezza:/tmp$ strip test
sicurezza@sicurezza:/tmp$ readelf -s test

Symbol table '.dynsym' contains 5 entries:
  Num:  Value  Size Type   Bind  Vis      Ndx Name
    0: 00000000      0 NOTYPE LOCAL DEFAULT UND
    ...
    3: 00000000  460 FUNC   GLOBAL DEFAULT UND puts@GLIBC_2.0 (2)
    4: 0804848c    4 OBJECT GLOBAL DEFAULT 15 _IO_stdin_used
sicurezza@sicurezza:/tmp$ readelf -S test | grep '\.symtab'
sicurezza@sicurezza:/tmp$
```

ELF: supporto al dynamic linking

Dynamic linking

parte delle operazioni di *linking* vengono rimandate a *runtime*

- facilita l'aggiornamento delle *shared library*
- consente caricamento delle librerie a *runtime*
- *overhead* a *runtime* superiore rispetto a *linking* statico
- simboli linkati dinamicamente devono essere risolti "al volo"

ELF: supporto al dynamic linking

GOT e PLT: cosa sono?

Global Offset Table e Procedure Linkage Table

- sezioni (.got e .plt) presenti in eseguibili che utilizzano *dynamic linking*
- PLT aggiunge un livello di indirettezza alle chiamate a funzione, consentendo anche il *lazy binding* dei relativi indirizzi

Esempio

```
sicurezza@sicurezza:~$ objdump -h $(which ls)
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
...						
11	.plt	00000610	08049538	08049538	00001538	2**2
					CONTENTS, ALLOC, LOAD, READONLY, CODE	
...						
21	.got	00000008	0805b5d4	0805b5d4	000125d4	2**2
					CONTENTS, ALLOC, LOAD, DATA	

ELF: supporto al dynamic linking

GOT e PLT: come funzionano?

Chiamata ad una funzione *shared*

- 1 chiamata `f()` *shared* → chiamata ad una entry della PLT
- 2 PLT: `jmp` indiretto ad una *entry* della GOT che conterrà:
 - la *prima volta*, l'indirizzo di una *entry* della PLT che trasferisce il controllo al linker, per risoluzione e aggiornamento della GOT
 - *successivamente*, l'indirizzo effettivo della *shared function*

Esempio

```
sicurezza@sicurezza:/tmp$ objdump -d test
...
0804830c <puts@plt>:
 804830c:    ff 25 08 96 04 08      jmp     *0x8049608
 8048312:    68 18 00 00 00        push   $0x18
 8048317:    e9 b0 ff ff ff       jmp     80482cc <_init+0x30>
...
080483d4 <main>:
...
80483ec:    e8 1b ff ff ff       call   804830c <puts@plt>
...
```

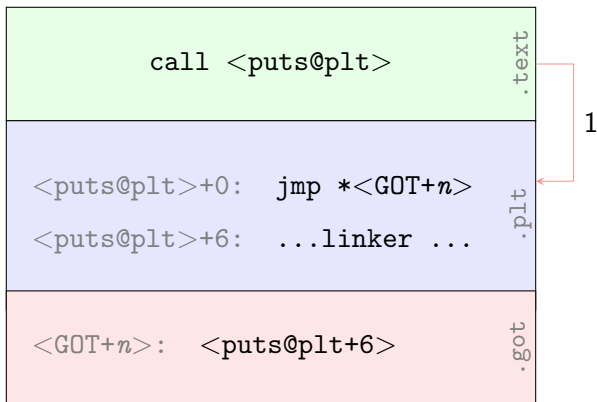
ELF: supporto al dynamic linking

Esempio `.got` e `.plt`

<code>call <puts@plt></code>	<code>.text</code>
<code><puts@plt>+0: jmp *<GOT+n></code> <code><puts@plt>+6: ...linker ...</code>	<code>.plt</code>
<code><GOT+n>: <puts@plt+6></code>	<code>.got</code>

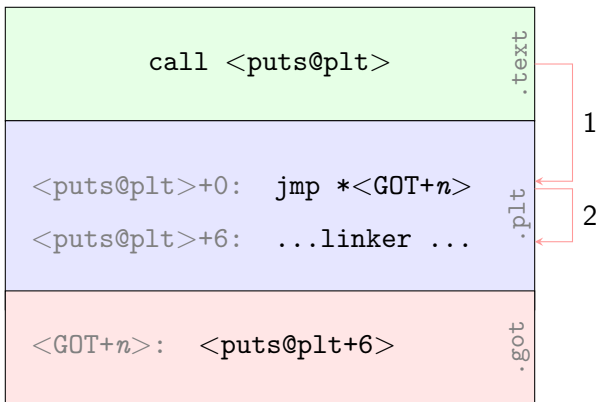
ELF: supporto al dynamic linking

Esempio `.got` e `.plt`



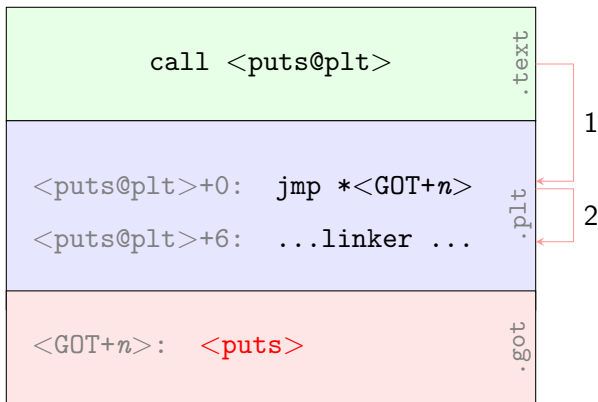
ELF: supporto al dynamic linking

Esempio `.got` e `.plt`



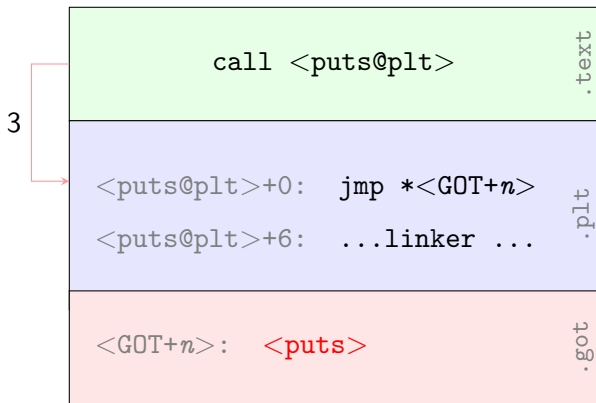
ELF: supporto al dynamic linking

Esempio .got e .plt



ELF: supporto al dynamic linking

Esempio .got e .plt



Lazy binding: performance

```
$ time (mplayer >/dev/null)      $ time (LD_BIND_NOW=1 mplayer >/dev/null)
real    0m0.039s                  real    0m0.141s
user    0m0.024s                  user    0m0.100s
sys     0m0.012s                  sys     0m0.016s
```

Esempio: dump librerie e dynamic loader

```
$ ldd /tmp/test
linux-gate.so.1 => (0xb7faf000)
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7e43000)
/lib/ld-linux.so.2 (0xb7fb0000)

sicurezza@sicurezza:/tmp$ readelf ./test -x .interp

Hex dump of section '.interp':
0x08048114 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
0x08048124 2e3200                                     .2.
```

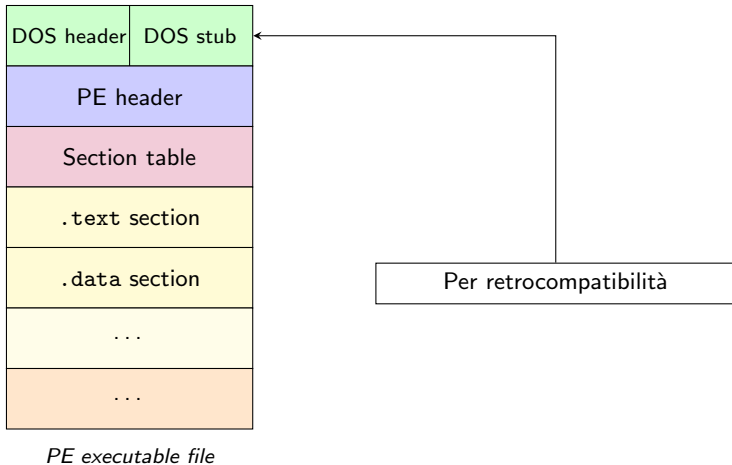
- 1 *executable*
- 2 *object*
- 3 *DLL*
- 4 *COM files, OCX controls, CLP applets, .NET executables*
- 5 *device driver*
- 6 ...

Esempio

```
sicurezza@sicurezza:/tmp$ file test.exe
test.exe: PE32 executable for MS Windows (console) Intel 80386 32-bit
sicurezza@sicurezza:/tmp$ file test.o
test.o: 80386 COFF executable not stripped - version 30821
sicurezza@sicurezza:/tmp$ file test.dll
test.dll: PE32 executable for MS Windows (DLL) (console) Intel ...
sicurezza@sicurezza:/tmp$ file test.sys
test.sys: PE32 executable for MS Windows (DLL) (native) Intel ...
```

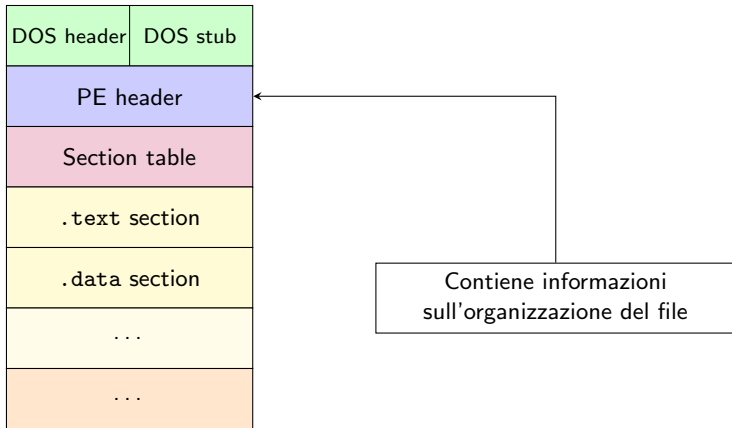
PE: Portable Executable

Struttura file PE



PE: Portable Executable

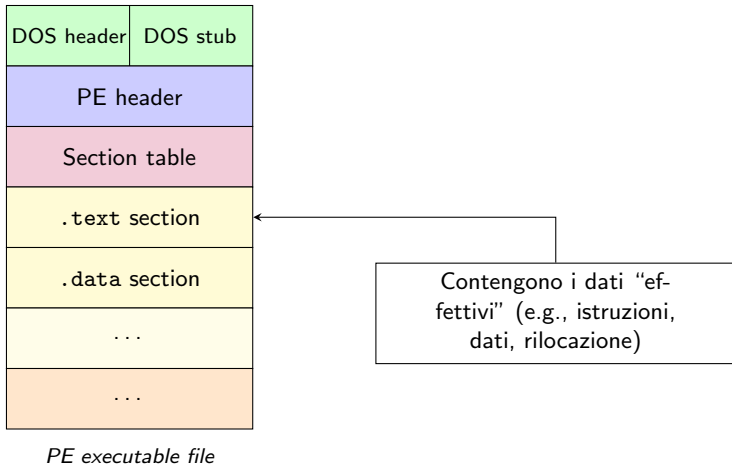
Struttura file PE



PE executable file

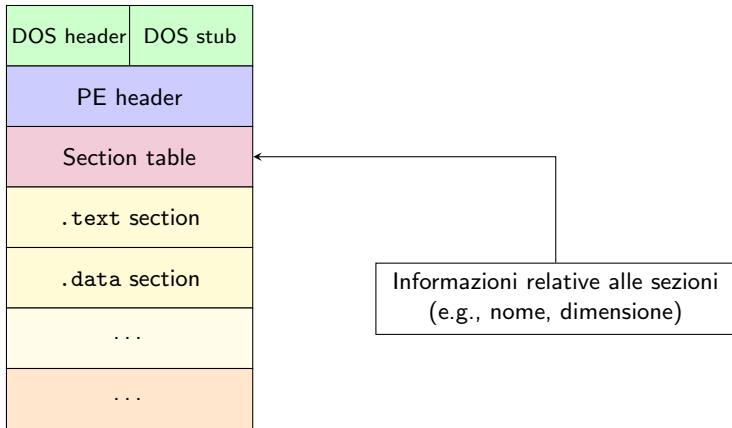
PE: Portable Executable

Struttura file PE



PE: Portable Executable

Struttura file PE



PE executable file

- *modulo*: file PE caricato in memoria (e “riorganizzato” dal loader)
- **RVA: Relative Virtual Address**
 - indirizzo relativo al *base address* dell’immagine in memoria
 - CODE @ 0x401000, base address 0x400000
RVA(CODE) = 0x1000
 - RVA \neq file offset
- strutture definite in `winnt.h`

PE: Portable Executable

DOS header & stub

DOS header

```
typedef struct _IMAGE_DOS_HEADER {  
    WORD e_magic;  
    ...  
    LONG e_lfanew;  
} IMAGE_DOS_HEADER,*PIMAGE_DOS_HEADER;
```

- 64 byte, magic number `\x4d\x5a` (“MZ”)
- `e_lfanew`: offset PE header
- segue DOS stub

DOS stub

- *“This program cannot be run in DOS mode”*

PE: Portable Executable

PE header

```
typedef struct _IMAGE_NT_HEADERS {  
    DWORD Signature; // "\x50\x45\x00\x00" ("PE\x00\x00")  
    IMAGE_FILE_HEADER FileHeader;  
    IMAGE_OPTIONAL_HEADER OptionalHeader;  
} IMAGE_NT_HEADERS32,*PIMAGE_NT_HEADERS32;
```

```
typedef struct _IMAGE_FILE_HEADER {  
    WORD Machine;  
    WORD NumberOfSections;  
    ...  
} IMAGE_FILE_HEADER,*PIMAGE_FILE_HEADER;
```

```
typedef struct _IMAGE_OPTIONAL_HEADER { // non e' opzionale!  
    ...  
    DWORD AddressOfEntryPoint;  
    ...  
    DWORD ImageBase;  
    DWORD SectionAlignment;  
    DWORD FileAlignment;  
    ...  
    WORD Subsystem;  
    ...  
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];  
} IMAGE_OPTIONAL_HEADER32,*PIMAGE_OPTIONAL_HEADER32;
```

```
typedef struct _IMAGE_SECTION_HEADER {  
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];  
    union {  
        DWORD PhysicalAddress;  
        DWORD VirtualSize;  
    } Misc;  
    DWORD VirtualAddress;  
    DWORD SizeOfRawData;  
    DWORD PointerToRawData;  
    ...  
    DWORD Characteristics;  
} IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER;
```

Alcune sezioni

- 1 *executable code section* (.text) (di solito, AddressOfEntryPoint ∈ .text)
- 2 *data sections* (.data, .rdata, .bss, ...)

Reverse engineering

Alcuni esempi

Reverse engineering: alcuni esempi

Variabili

```
1  #include <string.h>
2
3  int global;
4
5  int main(int argc, char **argv)
6  {
7      int local;
8      char v[256];
9
10     local = 10;
11     global = 11;
12
13     v[5] = 'a';
14
15     return 0;
16 }
```

```
1  <main>:
2  ...
3  8048374: lea 0x4(%esp),%ecx
4  8048378: and $0xfffffff0,%esp
5  804837b: pushl -0x4(%ecx)
6  804837e: push %ebp
7  804837f: mov %esp,%ebp
8  8048381: push %ecx
9  8048382: sub $0x110,%esp
10 8048388: movl $0xa,-0x8(%ebp)
11 804838f: movl $0xb,0x8049590
12 8048399: movb $0x61,-0x103(%ebp)
13 80483a0: mov $0x0,%eax
14 80483a5: add $0x110,%esp
15 80483ab: pop %ecx
16 80483ac: pop %ebp
17 80483ad: lea -0x4(%ecx),%esp
18 80483b0: ret
19  ...
```

Reverse engineering: alcuni esempi

Chiamata a funzione

```
1  int sum(int a, int b)
2  {
3      int c;
4
5      c = a + b;
6      return c;
7  }
8
9  int main(int argc, char **argv)
10 {
11     int a;
12
13     a = sum(2,3);
14     return 0;
15 }
```

```
1  <sum>:
2      push %ebp
3      mov %esp,%ebp
4      sub $0x10,%esp
5      mov 0xc(%ebp),%edx
6      mov 0x8(%ebp),%eax
7      add %edx,%eax
8      mov %eax,-0x4(%ebp)
9      mov -0x4(%ebp),%eax
10     leave
11     ret
12
13 <main>:
14     ...
15     sub $0x18,%esp
16     movl $0x3,0x4(%esp)
17     movl $0x2,(%esp)
18     call <sum>
19     mov %eax,-0x8(%ebp)
20     ...
```

Reverse engineering: alcuni esempi

Costrutto `if`

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     if (argc < 2) {
6         printf("! syntax error.\n");
7     } else {
8         printf("ok.\n");
9     }
10
11     return 0;
12 }
```

```
1 <main>:
2 ...
3 80483b7: mov %eax,-0x8(%ebp)
4 80483ba: cml $0x1,(%ecx)
5 80483bd: jg 80483cd <main+0x29>
6 80483bf: movl $0x80484b0,(%esp)
7 80483c6: call 80482d4 <puts@plt>
8 80483cb: jmp 80483d9 <main+0x35>
9 80483cd: movl $0x80484c0,(%esp)
10 80483d4: call 80482d4 <puts@plt>
11 80483d9: mov $0x0,%eax
12 ...
```

Reverse engineering: alcuni esempi

Costrutto `while`

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int i;
6
7      i = 0;
8      while (i < 10) {
9          printf("* %02d\n", i);
10         i += 1;
11     }
12
13     return 0;
14
15 }
```

```
1  <main>:
2  ...
3  80483b5: movl $0x0,-0x8(%ebp)
4  80483bc: jmp 80483d5 <main+0x31>
5  80483be: mov -0x8(%ebp),%eax
6  80483c1: mov %eax,0x4(%esp)
7  80483c5: movl $0x80484b0,(%esp)
8  80483cc: call 80482d8 <printf@plt>
9  80483d1: addl $0x1,-0x8(%ebp)
10 80483d5: cmpl $0x9,-0x8(%ebp)
11 80483d9: jle 80483be <main+0x1a>
12 80483db: mov $0x0,%eax
13 ...
```


Reverse engineering: alcuni esempi

Costrutto for

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int i;
6
7      for(i=0; i<10; i++) {
8          printf(" * %02d\n", i);
9      }
10
11     return 0;
12
13 }
```

```
1  <main>:
2      ...
3      80483b5: movl $0x0,-0x8(%ebp)
4      80483bc: jmp 80483d5 <main+0x31>
5      80483be: mov -0x8(%ebp),%eax
6      80483c1: mov %eax,0x4(%esp)
7      80483c5: movl $0x80484b0,(%esp)
8      80483cc: call 80482d8 <printf@plt>
9      80483d1: addl $0x1,-0x8(%ebp)
10     80483d5: cmpl $0x9,-0x8(%ebp)
11     80483d9: jle 80483be <main+0x1a>
12     80483db: mov $0x0,%eax
13     ...
```

Reverse engineering: alcuni esempi

Costrutto switch

```
1  int main(int argc, char **argv)
2  {
3      int a;
4
5      a = argc;
6
7      switch(a) {
8          case 0:
9              printf(" case 0\n"); break;
10         case 1:
11             printf(" case 1\n"); break;
12         ...
13         case 4:
14             printf(" case 4\n"); break;
15         default:
16             printf(" default\n"); break;
17     }
18
19     return 0;
20 }
```

```
1  <main>:
2      ...
3      80483b7: mov %eax,-0x8(%ebp)
4      80483ba: cmpl $0x4,-0x8(%ebp)
5      80483be: ja 8048414 <main+0x70>
6      80483c0: mov -0x8(%ebp),%eax
7      80483c3: shl $0x2,%eax
8      80483c6: mov 0x8048520(%eax),%eax
9      80483cc: jmp *%eax
10     80483ce: movl $0x80484f0,(%esp)
11     80483d5: call 80482d4 <puts@plt>
12     ...
13     8048406: movl $0x804850c,(%esp)
14     804840d: call 80482d4 <puts@plt>
15     8048412: jmp 8048420 <main+0x7c>
16     8048414: movl $0x8048513,(%esp)
17     804841b: call 80482d4 <puts@plt>
18     8048420: mov $0x0,%eax
19     ...
```



Hands on!

```
$ wget http://security.di.unimi.it/sicurezza1314/samples.tar.gz
```

Usando **solo** objdump:

- 1 sampleA: quali variabili (e di che tipo) sono usate dalla procedura `disass_me`?
- 2 sampleB: con quali parametri viene chiamata la procedura `disass_me`?
- 3 sample(C|D|E): quali costrutti sono usati nelle varie procedure `disass_me`?

Vale tutto:

- 1 `sample_switch`: è possibile fare in modo che venga eseguito il case 0, *senza modificare il codice*? Se sì, come?

- tecniche di disassembly
- per chi non l'avesse già fatto: **installare IDA Pro!**
 - versione freeware:
<http://www.hex-rays.com/idapro/idadownfreeware.htm>