Software Security 1
Introduzione al Reverse Engineering

Nicola VELLA
Università di Pisa

https://cybersecnatlab.it

License & Disclaimer

License Information

This presentation is licensed under the Creative Commons BY-NC License

To view a copy of the license, visit:

http://creativecommons.org/licenses/by-nc/3.0/legalcode

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Argomenti

- Introduzione alla categoria ***binary***
- Comprensione della **memoria** a basso livello
- Introduzione **assembly x86-64**
- Introduzione alle metodologie di base per il ***reverse engineering***

Argomenti

- Introduzione alla categoria ***binary***
- Comprensione della **memoria** a basso livello
- Introduzione **assembly x86-64**
- Introduzione alle metodologie di base per il ***reverse engineering***

Introduzione binary

Nella categoria **binary** rientrano tutte quelle **sfide** dove si ha a che fare con eseguibili **nativi**

```
root@oc23:/ctf/work# xxd a.out | head -n 20
00000000: 7f45 4c46 0201 0103 0000 0000 0000 0000  .ELF............
00000010: 0200 3e00 0100 0000 5016 4000 0000 0000  ..>.....P.@.....
00000020: 4000 0000 0000 0000 c0b4 0d00 0000 0000  @...............
00000030: 0000 0000 4000 3800 0a00 4000 1f00 1e00  ....@.8...@.....
00000040: 0100 0000 0400 0000 0000 0000 0000 0000  ................
00000050: 0000 4000 0000 0000 0000 4000 0000 0000  ..@.......@.....
00000060: 0005 0000 0000 0000 0005 0000 0000 0000  ................
00000070: 0010 0000 0000 0000 0100 0000 0500 0000  ................
00000080: 0010 0000 0000 0000 0010 4000 0000 0000  ..........@.....
00000090: 0010 4000 0000 0000 bd66 0900 0000 0000  ..@......f......
000000a0: bd66 0900 0000 0000 0010 0000 0000 0000  .f..............
000000b0: 0100 0000 0400 0000 0080 0900 0000 0000  ................
000000c0: 0080 4900 0000 0000 0080 4900 0000 0000  ..I.......I.....
000000d0: f484 0200 0000 0000 f484 0200 0000 0000  ................
000000e0: 0010 0000 0000 0000 0100 0000 0600 0000  ................
000000f0: b007 0c00 0000 0000 b017 4c00 0000 0000  ..........L.....
00000100: b017 4c00 0000 0000 e05a 0000 0000 0000  ..L......Z......
00000110: 90b4 0000 0000 0000 0010 0000 0000 0000  ................
00000120: 0400 0000 0400 0000 7002 0000 0000 0000  ........p.......
00000130: 7002 4000 0000 0000 7002 4000 0000 0000  p.@.....p.@.....
root@oc23:/ctf/work# file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically
linked, for GNU/Linux 3.2.0, not stripped
root@oc23:/ctf/work# ./a.out
Ciao mondo!
```

Cosa sono gli **eseguibili nativi**?

- Sono dei **file** che contengono **codice macchina** che può essere eseguito direttamente dal processore.
- Oltre al **codice macchina** contengono alcune informazioni utilizzate dal sistema operativo per **caricarlo** in memoria.

Formato ELF

ELF (acronimo di **Executable and Linkable Format**), è un formato estremamente flessibile (e complesso) per rappresentare file binari.

Viene generalmente utilizzato nei sistemi **Linux** moderni per rappresentare **file eseguibili** e **librerie condivise**.

Ad alto livello, possiamo vederlo come un **insieme di strutture** che descrivono come caricare in memoria i dati salvati all'interno dello stesso file.

Esempio di informazione aggiuntive nei binari ELF


```
root@oc23:/ctf/work# gcc main.c
root@oc23:/ctf/work# nm -n a.out
                 w __cxa_finalize@GLIBC_2.2.5
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
                 U puts@GLIBC_2.2.5
000000000000038c r __abi_tag
0000000000001000 T _init
0000000000001060 T _start
0000000000001090 t deregister_tm_clones
00000000000010c0 t register_tm_clones
0000000000001100 t __do_global_dtors_aux
0000000000001140 t frame_dummy
0000000000001149 T main
0000000000001174 T _fini
0000000000002000 R _IO_stdin_used
0000000000002010 r __GNU_EH_FRAME_HDR
00000000000020f0 r __FRAME_END__
0000000000003db8 d __frame_dummy_init_array_entry
0000000000003dc0 d __do_global_dtors_aux_fini_array_entry
0000000000003dc8 d _DYNAMIC
0000000000003fb8 d _GLOBAL_OFFSET_TABLE_
0000000000004000 D __data_start
0000000000004000 W data_start
0000000000004008 D __dso_handle
0000000000004010 B __bss_start
0000000000004010 b completed.0
0000000000004010 D _edata
0000000000004010 D __TMC_END__
0000000000004018 B _end
root@oc23:/ctf/work# ./a.out
Ciao mondo!
```

```
root@oc23:/ctf/work# strip a.out
root@oc23:/ctf/work# nm -n a.out
nm: a.out: no symbols
root@oc23:/ctf/work# ./a.out
Ciao mondo!
```

Strumenti per analizzare binari ELF

readelf: stampa le informazioni contenute nei file ELF

nm: stampa tutti i simboli contenuti nei file ELF

objdump: stampa le informazioni contenute nei file oggetto, ha un compito più specifico rispetto a readelf

lld: stampa gli oggetti condivisi necessari all'esecuzione del programma

lief: Libreria python per analizzare e modificare file ELF

Tipologia di sfide binary: REV e PWN

Le sfide della categoria binary si dividono in 2 rami:
Reverse Engineering e **Pwn**

REV**:** Richiede l'analisi di un binario per comprenderne il funzionamento.

PWN**:** Richiede l'analisi di un binario per trovare vulnerabilità, per poi sfruttarle per cambiare il comportamento originale del programma

In tutti e due i tipi di sfide è **necessaria** una **analisi approfondita** del **binario**.

Argomenti

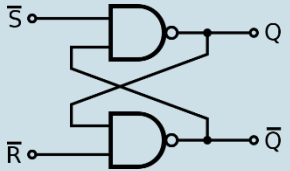
- Introduzione alla categoria ***binary***
- Comprensione della **memoria** a basso livello
- Introduzione **assembly x86-64**
- Introduzione alle metodologie di base per il ***reverse engineering***

Cos’è la memoria?

Per un programmatore potrebbe essere un insieme di variabili tipate

```
int numero
char buffer[10]
```

Per un ingegnere elettronico potrebbe essere un insieme di celle bistabili



Dobbiamo scegliere un livello di astrazione

Astrazioni di memoria

Visione interpretata dei byte

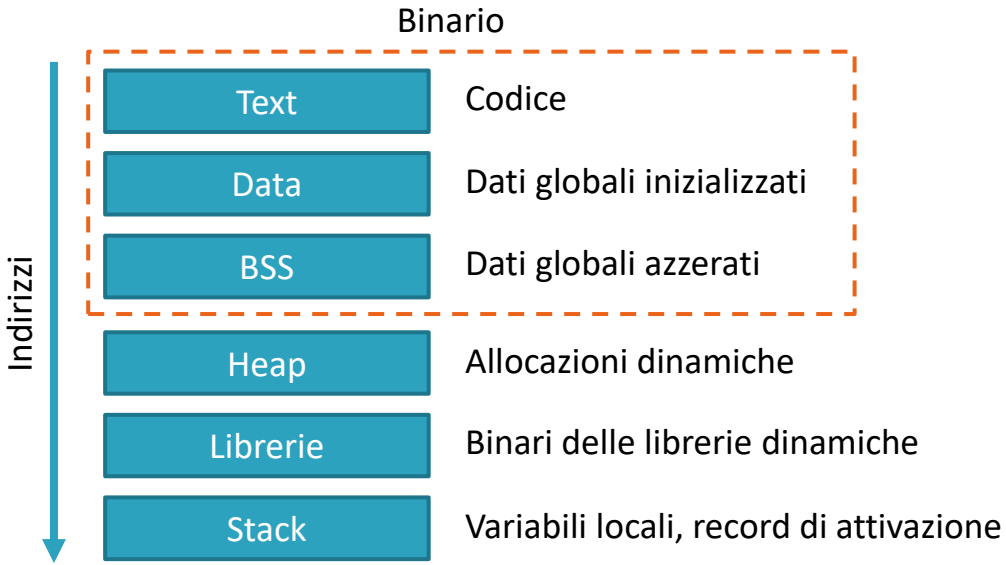
Sequenza di byte indirizzabili
Spazio indipendente per-processo
Solo alcune aree sono *mappate*

Sequenza di byte indirizzabili

Memoria virtuale

- Spazio virtuale di dimensione fissata (4GB / 256TB)
- Mapping fra aree di memoria virtuale e fisica
- Flag di protezione degli accessi
  - Read, write, execute

Spazio virtuale Linux userspace



Spazio virtuale Linux userspace

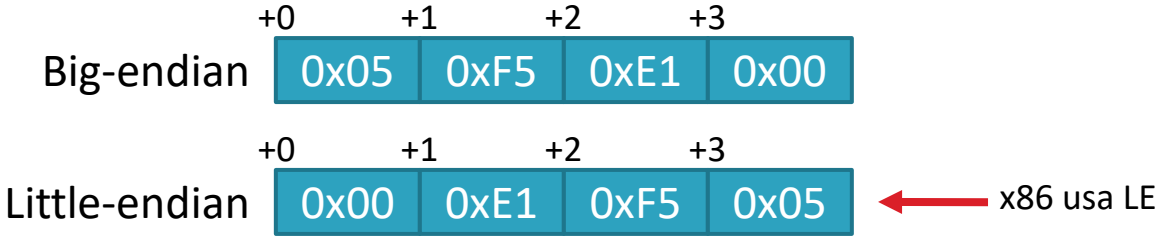

```c
int var_global;

int main()
{
    int var_stack;

    int *ptr_heap = malloc(sizeof(int));

    printf("main       @ %p\n", &main);
    printf("var_global @ %p\n", &var_global);
    printf("ptr_heap   = %p\n", ptr_heap);
    printf("var_stack  @ %p\n", &var_stack);

    getchar();

    return 0;
}
```

```
~/cc21/o/ss1_demos ❯ bin/address_space
main       @ 0x401146
var_global @ 0x404038
ptr_heap   = 0x1cec2a0
var_stack  @ 0x7ffcaa7adbd4
```

```
~/cc21/oli/ss1_demos ❯ sudo cat /proc/$(pgrep address_space)/maps
00400000-00401000 r--p 00000000 fd:02 5824216                            /home/andrea/cc21/oli/ss1_demos/bin/address_space
00401000-00402000 r-xp 00001000 fd:02 5824216                            /home/andrea/cc21/oli/ss1_demos/bin/address_space
00402000-00403000 r--p 00002000 fd:02 5824216                            /home/andrea/cc21/oli/ss1_demos/bin/address_space
00403000-00404000 r--p 00002000 fd:02 5824216                            /home/andrea/cc21/oli/ss1_demos/bin/address_space
00404000-00405000 rw-p 00003000 fd:02 5824216                            /home/andrea/cc21/oli/ss1_demos/bin/address_space
01cec000-01d0d000 rw-p 00000000 00:00 0                                  [heap]
7fd62dc4d000-7fd62dc6f000 r--p 00000000 fd:00 1837347                    /usr/lib64/libc-2.29.so
7fd62dc6f000-7fd62ddbc000 r-xp 00022000 fd:00 1837347                    /usr/lib64/libc-2.29.so
7fd62ddbc000-7fd62de08000 r--p 0016f000 fd:00 1837347                    /usr/lib64/libc-2.29.so
7fd62de08000-7fd62de09000 ---p 001bb000 fd:00 1837347                    /usr/lib64/libc-2.29.so
7fd62de09000-7fd62de0d000 r--p 001bb000 fd:00 1837347                    /usr/lib64/libc-2.29.so
7fd62de0d000-7fd62de0f000 rw-p 001bf000 fd:00 1837347                    /usr/lib64/libc-2.29.so
7fd62de0f000-7fd62de15000 rw-p 00000000 00:00 0
7fd62de55000-7fd62de56000 r--p 00000000 fd:00 1838933                    /usr/lib64/ld-2.29.so
7fd62de56000-7fd62de76000 r-xp 00001000 fd:00 1838933                    /usr/lib64/ld-2.29.so
7fd62de76000-7fd62de7e000 r--p 00021000 fd:00 1838933                    /usr/lib64/ld-2.29.so
7fd62de7f000-7fd62de80000 r--p 00029000 fd:00 1838933                    /usr/lib64/ld-2.29.so
7fd62de80000-7fd62de81000 rw-p 0002a000 fd:00 1838933                    /usr/lib64/ld-2.29.so
7fd62de81000-7fd62de82000 rw-p 00000000 00:00 0
7ffcaa78e000-7ffcaa7b0000 rw-p 00000000 00:00 0                          [stack]
7ffcaa7fa000-7ffcaa7fe000 r--p 00000000 00:00 0                          [vvar]
7ffcaa7fe000-7ffcaa800000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Rappresentazione di interi

- `unsigned int`: intero a 32 bit senza segno
  - Numero in [0, 4294967295]
- Esempio: valore 100.000.000
  - Esadecimale: 0x05F5E100



Rappresentazione di interi


```
In [1]: valore = 100_000_000

In [2]: print(f"0x{valore:08x}")
0x05f5e100

In [3]: from struct import pack

In [4]: pack('>I', valore) # Big-Endian
Out[4]: b'\x05\xf5\xe1\x00'

In [5]: pack('<I', valore) # Little-Endian
Out[5]: b'\x00\xe1\xf5\x05'
```

Rappresentazione di tipi C (x86)

- Interi little-endian
  - Interi con segno rappresentati secondo la notazione complemento a due
  - `char`, `int`, `short`, `long`, `enum`: interi a varie lunghezze
- `float` e `double`: IEEE 754
- I puntatori sono interi unsigned
  - Il loro valore è l'indirizzo puntato
  - 32/64 bit (per indirizzare intero spazio virtuale)

Rappresentazione di tipi C (x86)

- Array
  - Elementi disposti sequenzialmente
  - [i] @ base array + i * size elemento
- Strutture
  - Campi disposti sequenzialmente in ordine di dichiarazione
  - Campo @ base struct + somma size campi precedenti
  - (Il compilatore potrebbe introdurre del padding)

Argomenti

- Introduzione alla categoria ***binary***
- Comprensione della **memoria** a basso livello
- Introduzione **assembly x86-64**
- Introduzione alle metodologie di base per il ***reverse engineering***

Assembly x86-64

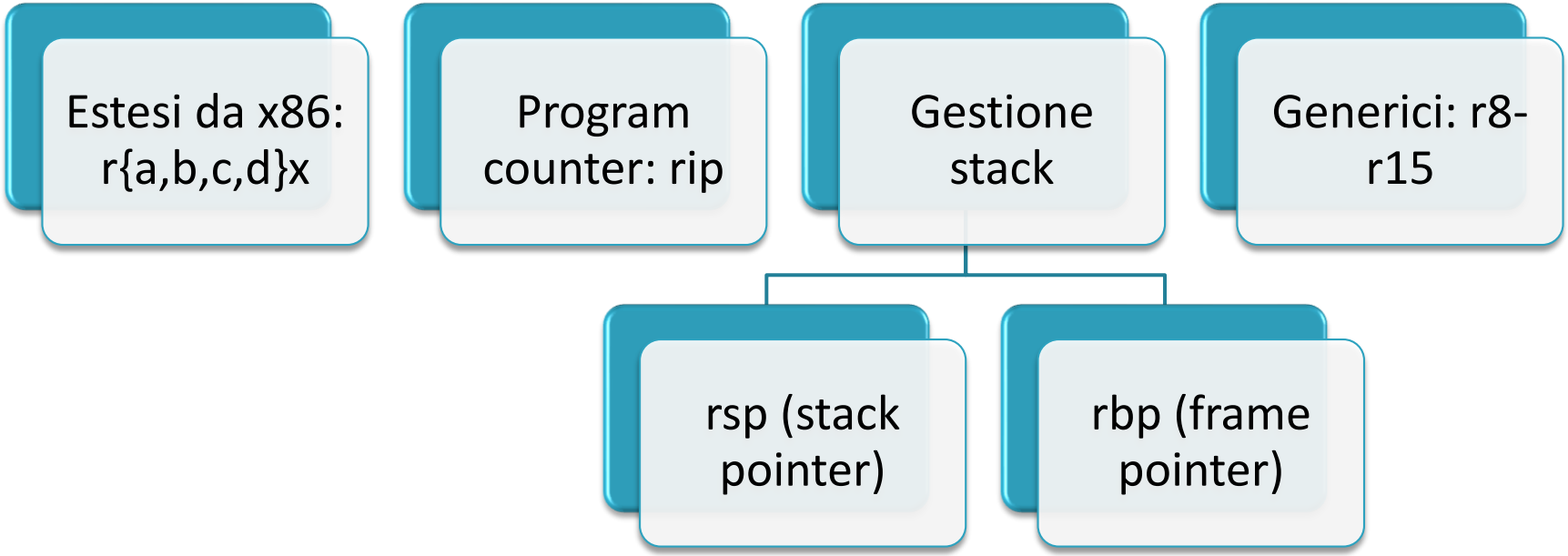
La CPU ha una piccola memoria locale composta da *registri*

- A questa memoria locale ci accediamo attraverso i nomi dei registri:
  - rax, rbx, rcx, …

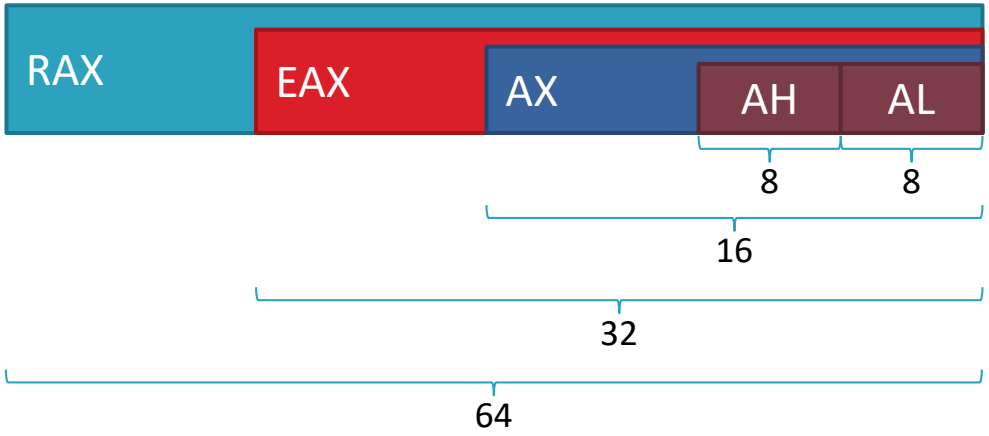
Ogni istruzione assembly ha degli operandi

- Registri: rax, ebx, r13d, …
- Memoria: [rax+4]

Registri x86-64



Registri x86_64



Una generica istruzione x86-64

- Sintassi Intel: operando destinazione a sinistra, operando sorgente a destra
- OPCODE <dst>, <src>: <dst> e <src> possono essere registri, registri utilizzati come indirizzi di memoria, indirizzi di memoria e valori immediate (max a 32 bit)
  - registers[dst] = OPCODE(registers[dst], registers[src])
  - memory[registers[dst]] = OPCODE(memory[registers[dst]], registers[src])
  - ...

Alcune istruzioni di base

- MOV <dst>, <src>
- PUSH <src> / POP <dst>
- ADD/SUB <dst>, <src>
- CALL <pc> / RET
- JMP <pc>

Salti condizionali

- CMP <opnd1>, <opnd2>
  - Confronta due valori e imposta delle flag
- J<condizione> <pc>
  - Salta a <pc> se le flag matchano <condizione>
- Salta se rax != 15:
  - CMP rax, 15
  - JNE …

Argomenti

Introduzione alla categoria ***binary***

Comprensione della **memoria** a basso livello

Introduzione **assembly x86-64**

Introduzione alle metodologie di base per il ***reverse engineering***

Reverse engineering: Perché?

Analisi di Malware

Ricerca di vulnerabilità in programmi closed source

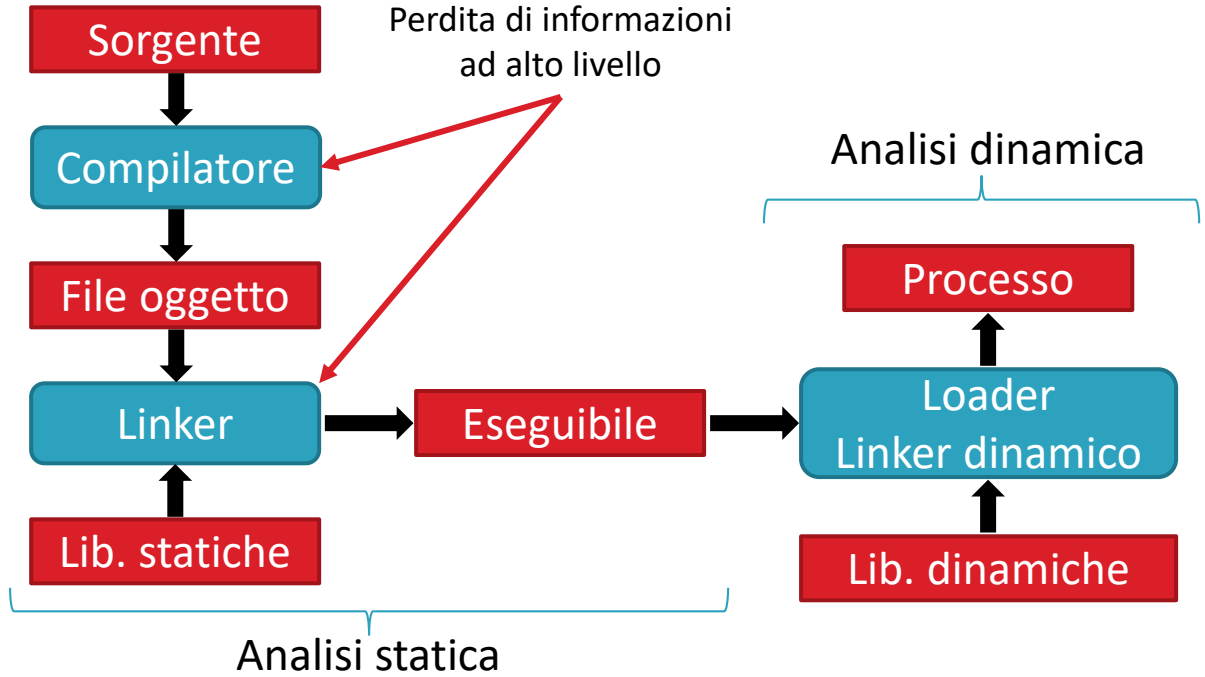
~~Game Cheating~~

Reverse Engineering: Come?

Analisi Statica

Analisi Dinamica

La vita di un programma



Tools Analisi Statica

Generici

- Ghidra
- IDA
- Binary Ninja
- Radare2

Compiti specifici

- JADX
- dnSpy/ILSpy
- uncompyle6/unpyc
- luadec

Tools Analisi Statica

GHIDRA, IDA, Binary Ninja, radare2	• Cercano di analizzare qualsiasi binario in input
JADX	• Reverse di bytecode JAVA
dnSpy, ILSpy	• Reverse di bytecode .NET
uncompyle6/unpyc	• Reverse di bytecode Python
luadec	• Reverse di bytecode LUA

Tools Analisi Dinamica

gdb	• Non pensato per analizzare binari senza informazioni di debugging, da utilizzare con GEF o PWNDBG
radare2	• Integra molte features comode per il reversing a differenza di GDB
rr	• Timeless debugging
frida	• Permette di iniettare codice JS in qualsiasi punto del programma
ida debugger	• Debugger con GUI disponibile anche nella versione free

Software Security 1
Introduzione al Reverse Engineering

Nicola VELLA
Università di Pisa

https://cybersecnatlab.it