Software Security 1

Andrea BIONDO
Università di Padova

https://cybersecnatlab.it

License & Disclaimer

License Information

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.

Obiettivi

- Comprensione della memoria a basso livello
- Conoscenza di base dei buffer overflows
- Conoscenze di base di reverse engineering

Prerequisiti

- Sistemi di numerazione
  - Esadecimale
- Conoscenza di base del linguaggio C
  - Tipi primitivi e strutturati, puntatori
  - Funzioni di I/O

Argomenti

- Spazio di memoria
- Reverse engineering
- Buffer overflows

Argomenti

- Spazio di memoria
- Reverse engineering
- Buffer overflows

Cos’è la memoria?

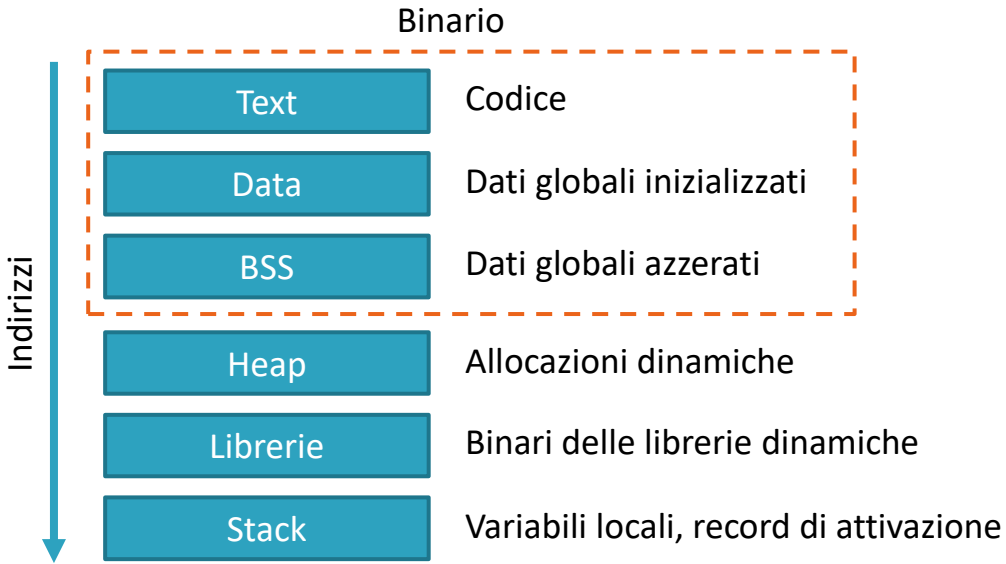
- Per un programmatore potrebbe essere un insieme di variabili tipate
- Per un ingegnere elettronico potrebbe essere un insieme di celle bistabili
- Dobbiamo scegliere un livello di astrazione

Astrazioni di memoria

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

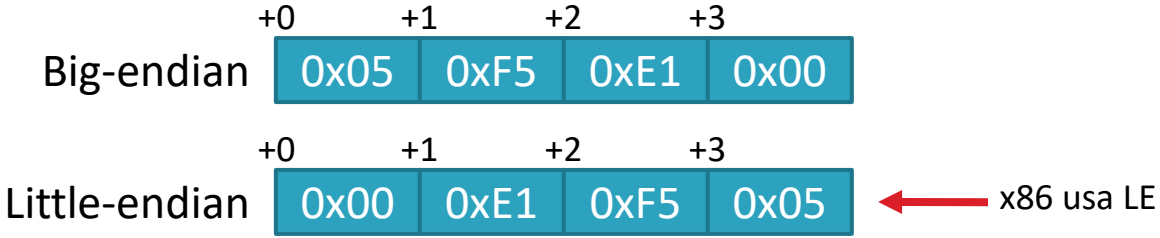

```c
int var_global;

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 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)

Corruzione di memoria

- Modificare la memoria di un processo in un modo diverso da quello previsto dal programmatore
- Controllare la memoria = controllare il processo!

Corruzione di memoria in the wild

- Malware
  - Morris worm (1988!), Blaster, Sasser, Conficker, …
  - Più recentemente, StuxNet e WannaCry
- Servizi remoti e applicazioni utente
  - Attacchi su server e browser
- Sbloccaggio di dispositivi
  - Root Android, jailbreak iOS, console per videogiochi

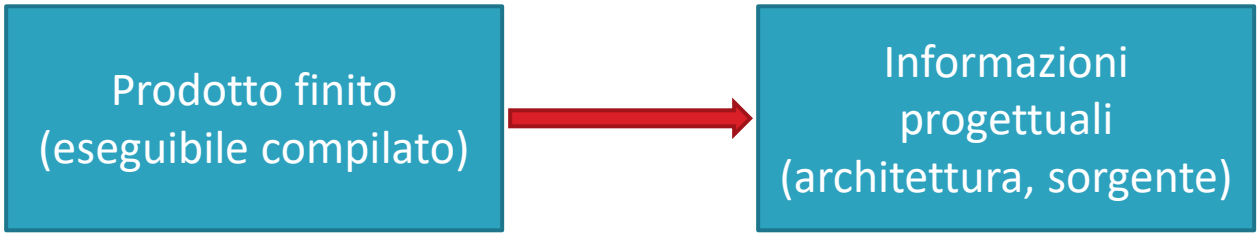
Argomenti

- Spazio di memoria
- Reverse engineering
- Buffer overflows

Reverse engineering

“Analizzare un sistema per creare rappresentazioni ad alto livello di astrazione”

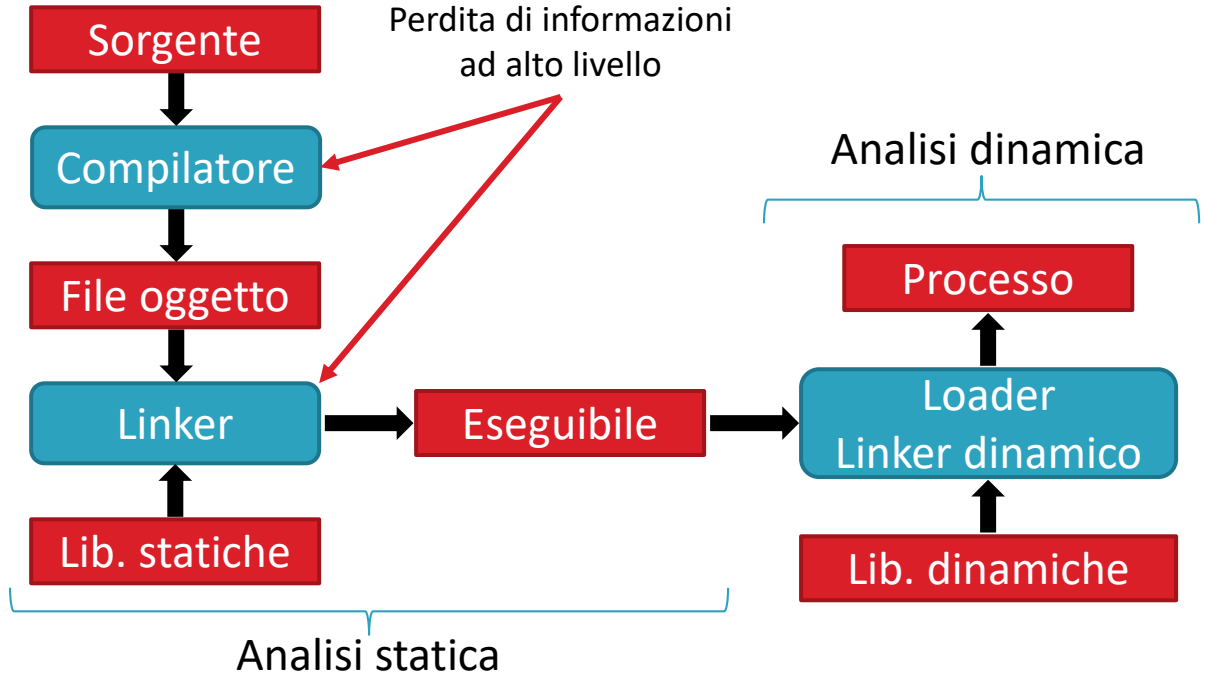
(Chikofsky, Cross 1990)



Reverse engineering

- Molte ragioni per fare reversing
  - Recupero di codice sorgente
  - Documentazione mancante o insufficiente
  - Analisi di prodotti concorrenti
  - “Aprire” piattaforme proprietarie
  - Auditing di sicurezza
  - Curiosità

La vita di un programma



Eseguibili

- Molti formati, a seconda dell'OS
  - PE (Windows), Mach-O (MacOS, iOS), ELF (*nix), …
- Divisi in varie *sezioni/segmenti* mappabili
  - Cioè che diventano aree di memoria virtuale a runtime

Gli strumenti

- Analisi statica
  - Disassemblatori, decompilatori (e.g., Ghidra)
  - Avanzati: interpretazione astratta, esecuzione simbolica, …
- Analisi dinamica
  - Debugger (e.g., GDB)
  - Avanzati: tracer, instrumentazione dinamica, …

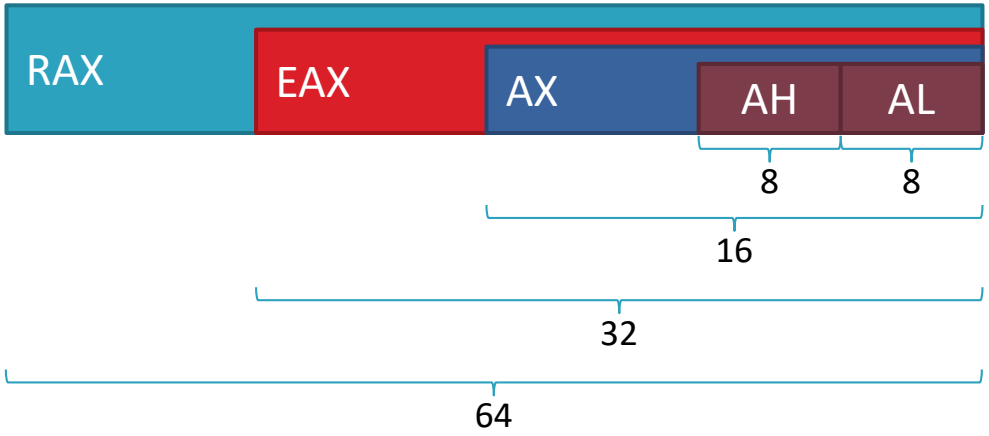
Assembly x86_64

- La CPU ha una piccola memoria locale composta da *registri*
- Ogni istruzione assembly ha degli operandi
  - Registri: rax, ebx, r13d, …
  - Memoria: [rax+4]
- Notazione Intel: <op> <destinazione>, <sorgente>

Registri x86_64

- Estesi da x86: r{a,b,c,d}x
- Program counter: rip
- Gestione stack
  - rsp (stack pointer)
  - rbp (frame pointer)
- Generici: r8-r15

Registri x86_64



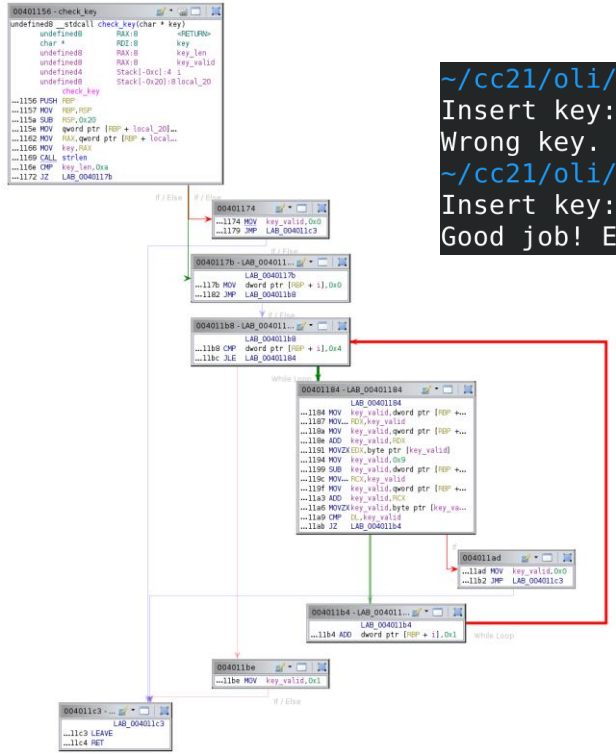
Alcune istruzioni di base

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

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 …

Reversing statico con Ghidra



```
~/cc21/oli/ss1_demos ❯ bin/reverse
Insert key: maybelucky
Wrong key.
~/cc21/oli/ss1_demos ❯ bin/reverse
Insert key: abcdeedcba
Good job! Enjoy.
```

```
2  undefined8 check_key(char *key)
3
4  {
5    size_t key_len;
6    undefined8 key_valid;
7    int i;
8
9    key_len = strlen(key);
10   if (key_len == 10) {
11     i = 0;
12     while (i < 5) {
13       if (key[i] != key[9 - i]) {
14         return 0;
15       }
16       i = i + 1;
17     }
18     key_valid = 1;
19   }
20   else {
21     key_valid = 0;
22   }
23   return key_valid;
24 }
```

Argomenti

- Spazio di memoria
- Reverse engineering
- Buffer overflows

Buffer overflows


```
char name[100];
printf("Come ti chiami? ");
scanf("%s", name);
printf("Ciao, %s!\n", name);
```

Buffer overflows

- Cosa succede se l'utente ha un nome più lungo di 100 caratteri?
  - O se è malevolo e *vuole* dare più di 100 caratteri in input...
- `scanf` (in questo caso) non controlla i bound
  - Continuerà a scrivere caratteri oltre la fine di `name`, sovrascrivendo la memoria che lo segue

Buffer overflows

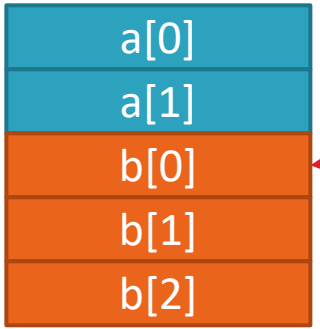
- Si ha un buffer overflow quando il programma scrive oltre la fine di un buffer
  - Perché i dati sono più lunghi del buffer
- Classica vulnerabilità di corruzione della memoria
  - Scriviamo dati dall'attaccante (input utente) in posizioni di memoria che il programmatore non aveva previsto potessero essere modificate (oltre la fine del buffer)

Conseguenze del buffer overflow

- Le garanzie di correttezza cadono completamente
- Se corrompiamo abilmente la memoria, possiamo ottenere il controllo completo del processo
  - Arbitrary code execution

Accessi out-of-bounds


```
struct {
  int a[2];
  int b[3];
};
```



Questo è anche a[2]!

```
a[2] = 42;
printf("%d\n", b[0]);
/* stampa 42 */
```

Accessi out-of-bounds


```c
int main()
{
    struct {
        int a[4];
        int b[3];
    } s;

    memset(&s, 0, sizeof(s));

    int idx;
    printf("Index: ");
    scanf("%d", &idx);
    int value;
    printf("Value: ");
    scanf("%d", &value);
    s.a[idx] = value;

    for (int i = 0; i < 3; i++)
        printf("b[%d] = %d (0x%08x)\n",
               i, s.b[i], s.b[i]);
}
```

Nessun bound check!

```
~/cc21/o/ss1_demos ❯ bin/oob1
Index: 3
Value: 42
b[0] = 0 (0x00000000)
b[1] = 0 (0x00000000)
b[2] = 0 (0x00000000)
~/cc21/o/ss1_demos ❯ bin/oob1
Index: 4
Value: 42
b[0] = 42 (0x0000002a)
b[1] = 0 (0x00000000)
b[2] = 0 (0x00000000)
~/cc21/o/ss1_demos ❯ bin/oob1
Index: 5
Value: 42
b[0] = 0 (0x00000000)
b[1] = 42 (0x0000002a)
b[2] = 0 (0x00000000)
```

Scritture OOB

Accessi out-of-bounds


```c
int main()
{
    struct {
        char a[4];
        int b[3];
    } s;

    memset(&s, 0, sizeof(s));

    int idx;
    printf("Index: ");
    scanf("%d", &idx);
    int value;
    printf("Value: ");
    scanf("%d", &value);
    s.a[idx] = value;

    for (int i = 0; i < 3; i++)
        printf("b[%d] = %d (0x%08x)\n",
               i, s.b[i], s.b[i]);
}
```

Nessun bound check!

```
~/cc21/o/ss1_demos ❯ bin/oob2
Index: 3
Value: 42
b[0] = 0 (0x00000000)
b[1] = 0 (0x00000000)
b[2] = 0 (0x00000000)
~/cc21/o/ss1_demos ❯ bin/oob2
Index: 4
Value: 42
b[0] = 42 (0x0000002a)
b[1] = 0 (0x00000000)
b[2] = 0 (0x00000000)
~/cc21/o/ss1_demos ❯ bin/oob2
Index: 5
Value: 42
b[0] = 10752 (0x00002a00)
b[1] = 0 (0x00000000)
b[2] = 0 (0x00000000)
```

Scritture OOB

Pattern pericolosi

- Senza bound checking
  - gets, scanf %s, sprint, strcpy
- Bound checking usato impropriamente
  - fgets, (f)read, snprintf, memcpy, strncpy, …
- Manipolazioni manuali
  - Loop di copia, accessi ad array, …

Un primo overflow


```c
int check_authentication()
{
    int auth_flag = 0;
    char password_buffer[16];

    printf("Enter password: ");
    scanf("%s", password_buffer);

    if (!strcmp(password_buffer,
                "hacktheworld"))
        auth_flag = 1;
    if (!strcmp(password_buffer,
                "olicyber"))
        auth_flag = 1;

    printf("auth_flag = %d (0x%08x)\n",
           auth_flag, auth_flag);

    return auth_flag;
}
```

Nessun bound check!

```c
int main()
{
    if (check_authentication())
        puts("Access granted!");
    else
        puts("Access denied.");
}
```

```
~/cc21/o/ss1_demos ❯ bin/auth_overflow
Enter password: hacktheworld
auth_flag = 1 (0x00000001)
Access granted!
~/cc21/o/ss1_demos ❯ bin/auth_overflow
Enter password: olicyber
auth_flag = 1 (0x00000001)
Access granted!
~/cc21/o/ss1_demos ❯ bin/auth_overflow
Enter password: 0123456789
auth_flag = 0 (0x00000000)
Access denied.
~/cc21/o/ss1_demos ❯ bin/auth_overflow
Enter password: 01234567890123456789012345678
auth_flag = 14648 (0x00003938)
Access granted!
```

Overflow dei caratteri 89

Software Security 1

Andrea BIONDO
Università di Padova

https://cybersecnatlab.it