

Sicurezza del Software

Riccardo Aziani

Ottobre 2024

Indice

1	Spazio di memoria	2
1.1	Astrazioni di memoria	2
1.1.1	Memoria fisica	2
1.1.2	Memoria virtuale	3
1.1.3	Dati tipati (variabili)	4
1.2	Spazio virtuale Linux userspace	4
1.2.1	Un esempio	5
1.3	Rappresentazione di interi	6
1.4	Rappresentazione di tipi C (x86)	7
1.5	Corruzione di memoria	7
2	Reverse Engineering	8
2.1	La vita di un programma	9
2.2	Eseguibili	10
2.3	Gli strumenti	10
2.3.1	Analisi statica	10
2.3.2	Analisi dinamica	11
2.4	Assembly x86_64 (64 bit)	11
2.4.1	Registri x86_64	11
2.4.2	Alcune istruzioni di base	12
2.4.3	Salti condizionali	13
2.5	Reversing statico con Ghidra	13
3	Buffer Overflows	15
3.1	Accessi out-of-bounds	16
3.2	Pattern pericolosi	17
3.3	Un primo overflow	18

Capitolo 1

Spazio di memoria

Non è immediato definire cosa sia la memoria, perché è necessario scegliere un **livello di astrazione** a cui fare riferimento: per un programmatore, la memoria potrebbe essere un insieme di variabili tipate; invece, per un ingegnere elettronico potrebbe essere un insieme di celle elettroniche.

A seconda del livello di astrazione la memoria può essere definita in modi diversi.

1.1 Astrazioni di memoria

Per quanto riguarda le astrazioni a livello software, ci sono principalmente tre livelli.



1.1.1 Memoria fisica

È il livello più basso di astrazione. Ha una struttura semplice: è una **lunga sequenza di byte** dove ogni byte ha un **indirizzo**; il primo byte ha indirizzo 0, il secondo 1, e così via.

È possibile leggere o scrivere dei byte in questa memoria usando delle istruzioni del processore (che parlerà effettivamente con i banchi di RAM).

Problematiche dell'utilizzo della memoria fisica

Questo strato di memoria non è adatto alle esigenze dei sistemi moderni, come ad esempio la programmazione, perché?

Immaginiamo di avere due programmi in esecuzione nello stesso momento sulla stessa macchina; entrambi questi programmi devono fare delle operazioni sulla memoria, operando quindi su certi indirizzi di memoria.

Se questi programmi fossero scritti scegliendo gli stessi indirizzi, finirebbero per *"pestarsi i piedi"*. Da una parte è problematico far convivere applicazioni diverse, assicurandosi che utilizzino indirizzi di memoria diversi; dall'altra ci sono problemi di sicurezza nell'usare direttamente la memoria fisica: un'applicazione malevola potrebbe andare a toccare la memoria di un'altra applicazione influenzandone il comportamento.

Per queste ragioni nessuna applicazione moderna utilizza direttamente la memoria fisica, ma una sua astrazione chiamata *memoria virtuale*.

1.1.2 Memoria virtuale

È una astrazione della memoria fisica realizzata dal sistema operativo, che è l'unico componente che lavora effettivamente sulla memoria fisica.

A livello di *contenuto*, segue la stessa struttura della memoria fisica: è una sequenza di byte indirizzabili.

Però, a differenza della memoria fisica, **ogni processo di sistema ha uno spazio di memoria indipendente**: ogni programma ha la sua memoria **indipendente** ed **isolata** dalla memoria virtuale degli altri processi.

In questo modo, si evita che i processi *"si pestino i piedi"* o che uno vada a toccare la memoria dell'altro.

Mapping

Per realizzare questa astrazione si utilizzano dei mapping: ogni area della memoria virtuale è mappata a un'area della memoria fisica. Questo processo è realizzato dal sistema operativo.

Ne consegue che è possibile avere lo stesso indirizzo virtuale in due processi diversi, mappati a due indirizzi fisici diversi.

La memoria virtuale non è completamente accessibile, ma solo alcune aree che effettivamente servono sono mappate.

Flag di protezione

La memoria virtuale fornisce delle **funzionalità di protezione degli accessi**: si può marcare ciascuna area della memoria virtuale come solo lettura, leggibile/scrivibile, controllare se può contenere codice che può essere eseguito, e così via.

Si può dunque controllare il tipo di accesso che è consentito fare a determinate aree di memoria virtuale.

1.1.3 Dati tipati (variabili)

L'astrazione dei tipi tipati viene realizzata dai linguaggi di programmazione sopra la memoria virtuale. Quando dichiaro una variabile, il linguaggio di programmazione si tiene un pezzettino di memoria virtuale per contenere quella variabile.

Quando scrivo (ad esempio) un intero in quella variabile, il linguaggio di programmazione converte quell'intero in una sequenza di byte per poi metterlo in memoria.

Quando va a leggere quell'intero, viene letta la sequenza di byte dalla memoria per poi riconvertirlo in un intero.

1.2 Spazio virtuale Linux userspace

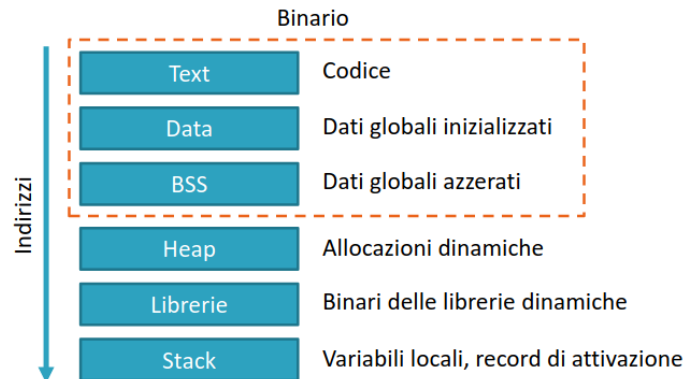
Andiamo a vedere com'è fatto la memoria virtuale di un processo utente su Linux; il kernel avrà uno spazio virtuale diverso.

Quando viene **lanciato un eseguibile** (binario), viene creato un processo e quell'eseguibile viene **caricato in memoria virtuale**.

Il binario contiene varie parti, come quella in cui è contenuta il codice o quella in cui sono contenuti i dati; tutte queste informazioni vengono caricate in memoria.

Altre aree interessanti in memoria virtuali sono:

- **heap**: contiene tutte le allocazioni dinamiche (*malloc()* in C, *new()* in C++)
- **librerie** esterne importate nel programma
- **stack**: è dove vengono memorizzate le variabili locali e tutta una serie di informazioni utilizzate per il flusso di controllo del programma, come gestire le chiamate a funzione



Il **binario** è posizionato ad indirizzi relativamente **bassi**, mentre lo **stack** è l'area che sta ad indirizzi più **alti**.

1.2.1 Un esempio

In Figura 1.1 viene mostrato un programma che stampa quattro indirizzi:

- indirizzo della funzione main, quindi del codice macchina del programma
- indirizzo di una variabile locale
- indirizzo di una locazione sull'heap
- indirizzo di stack

```
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;
}
```

Figura 1.1: Programma C

In Figura 1.2 viene mostrato il risultato dell'esecuzione del programma.

```
~/cc21/o/ss1 demos > bin/address_space
main      @ 0x401146
var_global @ 0x404038
ptr_heap   = 0x1cec2a0
var_stack  @ 0x7ffcaa7adbd4
```

Figura 1.2: Esecuzione del programma C

In Linux è possibile vedere il layout della memoria virtuale di un processo usando il *file system* `'proc'`: `/proc` ha una *sub-directory* per ogni processo del sistema (nominata con l'id del processo nel sistema).

Ciò che viene fatto in Figura 1.3 è stato usare `pgrep` per trovare l'id associato al processo `address_space` (il programma C).

In ogni linea:

- il primo numero è l'indirizzo di partenza in esadecimale, e dopo '-' c'è l'indirizzo di fine
- dopo ci sono i flag di protezione (r, rw, x)

```
~/cc21/oli/ssl_demos > sudo cat /proc/$(pgrep address_space)/maps
00400000-00401000 r--p 00000000 fd:02 5824216 /home/andrea/cc21/oli/ssl_demos/bin/address_space
00401000-00402000 r-xp 00001000 fd:02 5824216 /home/andrea/cc21/oli/ssl_demos/bin/address_space
00402000-00403000 r--p 00002000 fd:02 5824216 /home/andrea/cc21/oli/ssl_demos/bin/address_space
00403000-00404000 r--p 00003000 fd:02 5824216 /home/andrea/cc21/oli/ssl_demos/bin/address_space
00404000-00405000 rw-p 00004000 fd:02 5824216 /home/andrea/cc21/oli/ssl_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
7fd62de15000-7fd62de56000 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
7fd62de7e000-7fd62de80000 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]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Figura 1.3: Layout della memoria virtuale del processo

L'indirizzo del *main* è leggibile ed eseguibile.

La *variabile globale* è leggibile e scrivibile; lo stesso vale per l'*heap*.

Dopo ci sono una serie di mapping di *libc* e *ld*, una serie di librerie standard caricate in tutti i processi. Se il processo utilizza delle librerie aggiuntive saranno caricate in questa zona.

Infine, negli indirizzi più alti, si trova lo *stack*.

1.3 Rappresentazione di interi

Esempio

unsigned int: intero a 32 bit senza segno

Valore = 100.000.000 → 0x05F5E100

La rappresentazione in *esadecimale* è comoda perché **due cifre corrispondono esattamente ad un byte**: si va da 00 a FF, ovvero 255, che è il range di valori possibili per un byte (2^8).

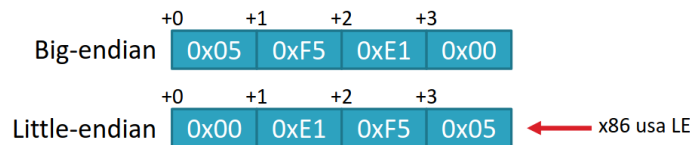


Figura 1.4: Rappresentazioni Big-endian e Little-endian

Esistono due tipi di rappresentazioni:

- Big-Endian: il primo byte è quello più significativo
- Little-endian: il primo byte è quello meno significativo; questa rappresentazione è più comoda in termini computazionali

1.4 Rappresentazione di tipi C (x86)

- **Interi little-endian:**
 - per gli interi con segno si utilizza la notazione *complemento a due*
 - **char, int, short, long:** sono tutti degli interi a diverse lunghezze (bit)
- **Puntatori:** sono *interi unsigned*; il loro valore corrisponde all'**indirizzo di memoria a cui punta**; è necessario un intero sufficientemente grande a contenere un qualsiasi indirizzo virtuale (32/64 bit a seconda del sistema)
- **Array:** C dispone gli elementi sequenzialmente, uno dietro l'altro;
- **Strutture:** i campi sono disposti sequenzialmente in ordine di dichiarazione; il compilatore potrebbe introdurre del *padding* (spazio vuoto) perché alcuni tipi hanno dei requisiti di allineamento (vogliono essere ad indirizzi di memoria multipli di certi numeri)

1.5 Corruzione di memoria

Un processo è determinato dalla sua memoria: l'eseguibile è caricato in memoria, il codice lavora sulla memoria.

Se riuscissimo a controllare e modificare la memoria di un programma in un modo diverso da quello previsto da quello dal programmatore, allora avremmo il controllo del processo!

Capitolo 2

Reverse Engineering

L'ingegneria inversa si può definire in maniera precisa come quel processo in cui andiamo ad *analizzare un sistema per creare rappresentazioni ad alto livello di astrazione*.

Cosa significa?

Immaginiamo di essere *Ferrari* e di voler realizzare un motore; partiremmo da delle specifiche di massima sulla base delle quali svilupperemo un progetto. Man mano il progetto includerà ad esempio i disegni delle componenti meccaniche, le indicazioni su come assemblarle . . .

Tutte queste informazioni saranno poi mandate ad un impianto di produzione che andrà a comporre il motore; questo è il normale *processo ingegneristico*.

Nell'**ingegneria inversa** vogliamo fare il contrario: partire da un *prodotto finito* e e ricavare le *informazioni progettuali*.



Quando caliamo questo processo nel *software reverse engineering*, il prodotto finito è solitamente un **eseguibile compilato**; ciò che noi vogliamo ricavare sono delle informazioni come ad esempio:

- architettura del programma
- come funziona internamente (ad esempio certi algoritmi)
- completo recupero del codice del programma (caso estremo, in genere è sufficiente ricavare certe informazioni)

Perché si fa reverse engineering?

Esistono molte ragioni per fare *reversing*:

- un'azienda che produce un software e perde parti del codice sorgente, ha bisogno di **recuperare codice** per evitare perdite
- devo interagire con un sistema la cui **documentazione è mancante o insufficiente**
- **analisi dei prodotti dei concorrenti** nell'ambito industriale
- **"aprire" delle piattaforme proprietarie**, come ad esempio le console dei videogiochi
- **auditing di sicurezza**: ho un eseguibile ma non ho il codice sorgente, e voglio analizzarlo per cercare delle vulnerabilità di sicurezza
- *curiosità*

2.1 La vita di un programma

Perché è difficile fare analizzare un binario compilato?

Perché nel codice sorgente ci sono una serie di informazioni ad alto livello utili per aiutare l'uomo a comprendere il software (nomi delle variabili, funzioni, commenti ...), ma che sono inutili per la macchina; nella fase di **compilazione** queste informazioni vengono scartate.

Avviene anche una traduzione dai linguaggi ad *alto livello* in **codice macchina**, comprensibile ed eseguibile dalla macchina; questo codice è progettato per essere eseguibile dalla macchina, non per essere facilmente comprensibile da un umano.

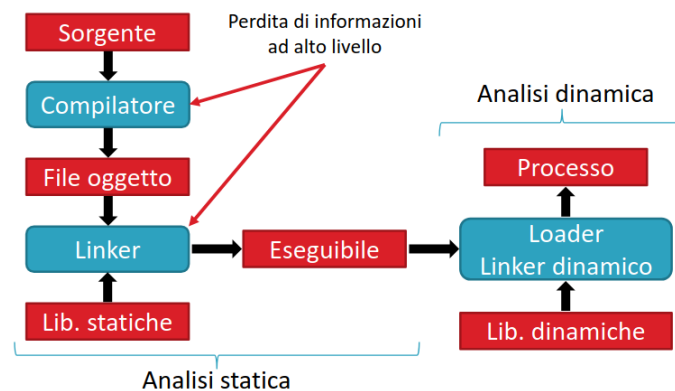


Figura 2.1: Vita di un programma

Nel reversing distinguiamo due tipi principali di *analisi*, che corrispondono alle due fasi della vita di un programma.

Dopo la fase di compilazione otteniamo un *file eseguibile*; analizzare tale file (codice, dati, ...) senza eseguirlo prende il nome di **analisi statica**.

Diversamente, un eseguibile può diventare un *processo*; studiare il programma mentre esegue (ad esempio un debugger) prende il nome di **analisi dinamica**.

2.2 Eseguibili

Esistono molti formati di eseguibili a seconda del sistema operativo; noi ci focalizziamo su **ELF** (*Executable Linking Format*), il formato dell'ambiente Linux.

In generale, i concetti di base sono gli stessi per tutti i formati: ogni eseguibile definisce una serie di **sezioni** o **segmenti** che sono delle parti del programma che saranno mappate in memoria virtuale a *runtime*. Ad esempio, potremmo avere:

- un segmento per il codice macchina
- un segmento per i dati, a loro volta divisi in:
 - un segmento per i dati *read-only*
 - un segmento per i dati scrivibili

Vengono usati segmenti diversi per permettere di mapparli in aree di memoria virtuale con **permessi diversi**.

2.3 Gli strumenti

2.3.1 Analisi statica

- **Disassemblatore:** il linguaggio macchina grezzo è estremamente difficile da leggere; con esso viene tradotto nel linguaggio *assembly*, il quale permette di facilitare la comprensione
- **Decompilatore:** partendo dall'*assembly* cerca di **ricostruire il codice sorgente del programma** (in un linguaggio *simil-C*); spesso la ricostruzione non è fedelissima per svariate ragioni (si perdono nomi delle variabili/funzioni, alcune strutture di controllo ad alto livello vengono ottimizzate dal compilatore facendolo apparire diversamente nel codice macchina, ...), ma sono comunque uno strumento molto utile

Ghidra è un tool di reversing che integra entrambi questi strumenti.

Un'altro strumento avanzato è *Anger*, il quale effettua un'**esecuzione simbolica**: ogni istruzione del programma viene trasformata in dei vincoli logici per rispondere alla domanda *"che input devo dare al programma per far sì che arrivi in un certo punto con un certo stato?"*.

2.3.2 Analisi dinamica

- **Debugger:** sono in grado di lavorare su file binari (eseguibili senza sorgente); un esempio è *GDB*. Permette di studiare l'evoluzione del programma durante la sua esecuzione
- **strumentazione dinamica:** è una tecnica che permette di iniettare del codice che verrà eseguito in risposta a determinati eventi; un tool è *Frida*

2.4 Assembly x86_64 (64 bit)

Il processore non lavora direttamente sulla memoria, ma su una piccola memoria locale contenente una serie di variabili dette **registri**. Questo viene fatto perché la RAM è **molto lenta** (per il processore), mentre i registri essendo una memoria molto piccola riescono a girare alla sua stessa velocità; dalla memoria dunque le informazioni necessarie vengono caricate nei registri.

Ogni istruzione assembly ha:

- uno **mnemonico** che indica ciò che quell'istruzione fa (ad esempio **add** per l'addizione)
- degli **operandi**, che possono essere dei registri o delle locazioni di memoria, che indicano i dati su cui opera l'istruzione

Notazione *Intel*: <op> <destinazione>, <sorgente>

Ad esempio: **add r1, r2**

2.4.1 Registri x86_64

Ha dei registri che sono l'estensione a 64 bit di alcuni registri di x86 (a 32 bit), con il nome di: **rax, rbx, rcx, rdx**

Ci sono alcuni registri speciali:

- **rip:** sta per *instruction pointer*, è il registro che prende l'**indirizzo dell'istruzione corrente**; prende anche il nome di **program counter**
- registri per la **gestione dello stack**:
 - **rsp:** è lo **stack pointer**, punta alla cima dello stack
 - **rbp:** è il **frame pointer**, è utilizzato per tenere traccia della porzione di stack che contiene le variabili locali della funzione corrente

Sono infine presenti dei **registri generici**, nominati da **r8** a **r15**.

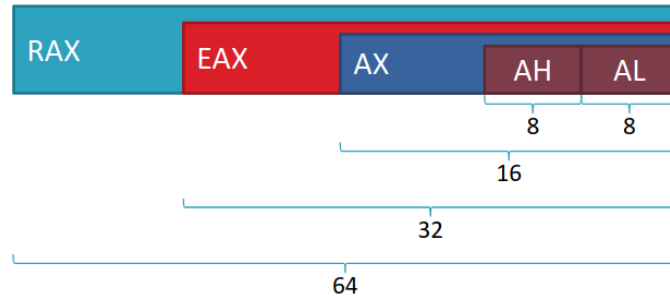


Figura 2.2: Registri x86

La struttura dei registri

I registri x86_64 sono un po' *complicati*, in quanto presentano una forma a matricosca, mostrata in Figura 2.2.

Quando x86 era a 16 bit, c'era il registro **ax** (oppure **bx**, ...); era possibile accedere agli 8 bit *alti* o *bassi* utilizzando i registri **ah** (*high*) o **al** (*low*).

Quando è stato introdotto x86 a 32 bit, si voleva mantenere compatibilità con le applicazioni a 16 bit; è stato quindi introdotto un registro **eax** (*extended ax*), mantenendo quello che prima era **ax** come la metà bassa del nuovo registro a 32 bit.

Quando è stato introdotto x86 a 64 bit, per mantenere nuovamente la compatibilità è stata fatta la stessa cosa introducendo il registro **rax**, e tenendo il vecchio **eax** come i 32 bit bassi del nuovo registro a 64 bit.

Questo schema si applica anche ai registri **rbx**, **rcx**, **rdx**.

2.4.2 Alcune istruzioni di base

- **MOV <dst>, <src>** | **MOV rax, rbx** → copia da una sorgente ad una destinazione
- **PUSH <src>** / **POP <dst>** → sono utilizzate per fare dei *push/pop* sullo stack
- **ADD/SUB <dst>, <src>** → utilizzate per fare addizioni/sottrazioni
- **CALL <pc>** / **RET** → sono utilizzate per fare le chiamate a funzioni (**CALL**) o per ritornare da una istruzione al chiamante (**RET**)

2.4.3 Salti condizionali

Una feature fondamentale per un processore è essere in grado di prendere delle decisioni (come eseguire un *statement if*); in assembly questo processo è diviso in due step:

1. `CMP <op1> <op2>` → è una funzione che confronta i valori della condizione; successivamente, imposta alcune flag che descrivono l'esito del confronto
2. `J <condizione> <pc>` → permette di **cambiare il program counter saltando ad un'altra locazione del programma** se una certa flag di condizione è stata impostata

Un *loop* può essere implementato con un salto all'indietro, ovvero salto all'inizio del loop finché la condizione non si falsifica.

2.5 Reversing statico con Ghidra

Abbiamo un piccolo binario, mostrato a destra in Figura 2.3, che chiede di inserire una chiave e risponde se è corretta o meno.

Facendo l'analisi con Ghidra possiamo ottenere le istruzioni assembly del binario; possiamo inoltre attivare la vista del **control flow graph**, mostrato a sinistra in Figura 2.3.

Esso mostra il **flusso di esecuzione del programma**: ogni blocco è un insieme di istruzioni dette *straight line*, cioè che non contengono salti (eccetto alla fine). Ogni blocco è collegato a quelli che possono essere i suoi successori.

Possiamo inoltre decompilare il binario ottenendo ciò che è mostrato in Figura 2.4 (rinominando le variabili).

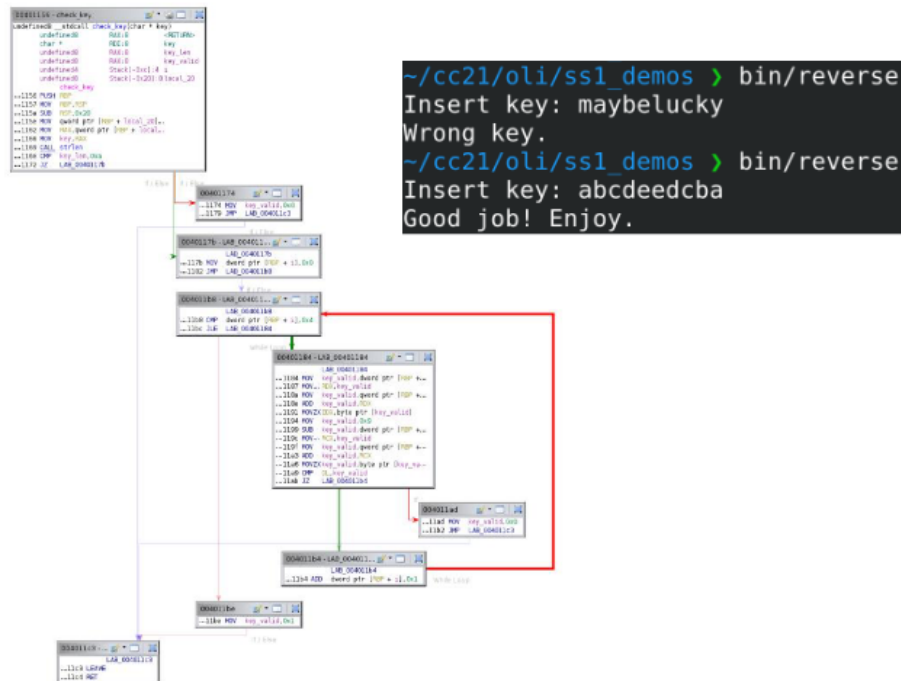


Figura 2.3: Binario e Control Flow Graph

```

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 }

```

Figura 2.4: Codice ottenuto attraverso la decompilazione e rinominando le variabili

Capitolo 3

Buffer Overflows

Consideriamo il codice mostrato in Figura:

1. dichiara un array di caratteri di 100 elementi (in C le stringhe sono rappresentate come un array di caratteri, terminata da un carattere nullo); in un array di 100 caratteri possiamo mettere una stringa di al più 99 caratteri, perché il 100esimo sarà il carattere terminatore.
2. usa `printf` per stampare a schermo la domanda
3. prende un input usando la funzione `scanf`: prende come primo argomento una stringa che definisce in che formato vogliamo prendere l'input (ad esempio, `%d` per gli interi); il secondo argomento è un puntatore all'array `name`
4. stampa a video sostituendo al placeholder `%s` il nome contenuto nell'array

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

Questo programma è **vulnerabile** a un **buffer overflow**: quando prendiamo in input il nome, non c'è nessun controllo che l'utente immetta al massimo 99 caratteri: se l'utente digita una stringa più lunga, la parte in eccesso andrà a sovrascrivere la parte in memoria che segue l'array preso in considerazione, dato che C non fa nessun tipo di controllo sui *bound*.

È possibile usare `scanf` in modo sicuro specificando la dimensione massima di caratteri da leggeri: ad esempio con `%99s` viene specificato di legger al più 99 caratteri.

Buffer overflow

Si ha un **buffer overflow** quando il programma scrive oltre la fine di un buffer/array, perché ci sta copiando dentro dei dati che sono più grandi della dimensione del buffer.

È una delle vulnerabilità di **corruzione della memoria** classiche: stiamo scrivendo dei dati dell'attaccante in delle locazioni di memoria che il programmatore non aveva previsto potessero essere modificate.

Conseguenze del buffer overflow

Le garanzie di correttezza del programma **cadono completamente**. Se corrompiamo abilmente la memoria, possiamo ottenere il **controllo completo** del processo (poiché il processo *vive* in memoria); questo caso prende il nome di *Arbitrary code execution*, dato che posso far eseguire del codice arbitrario al programma che sto attaccando.

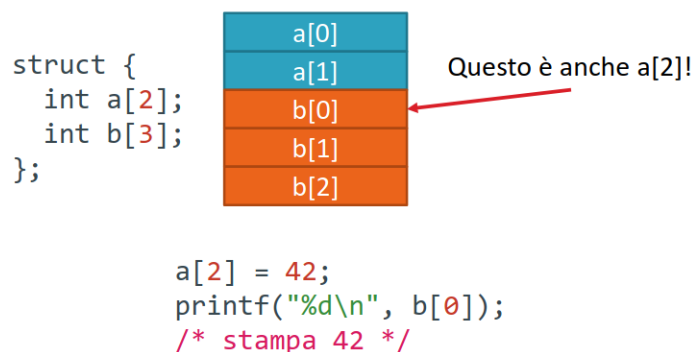
3.1 Accessi out-of-bounds

Perché nel buffer overflow la funzione che *prende l'input* scrive dopo la fine dell'array?

Questo dipende dal modo in cui C fa gli accessi agli array: considerando la situazione in Figura, abbiamo due array uno dietro l'altro in memoria (dentro ad una **struct** in modo che siano sequenziali in memoria); C non fa controllo sui *bound* dell'array, ma si limita a prendere l'indirizzo di partenza ed aggiungere l'indice moltiplicato per la dimensione dell'elemento.

Questo significa che accedere a **a[2]** equivale ad accedere a **b[0]**.

Accedere ad un indice fuori dai bound di un array è detto **accesso out-of-bounds**.



Un altro esempio

Questo programma ha una `struct` con due array; chiede un indice e un valore, ed imposta quell'indice di `a` a quel valore. Poi stampa tutti i valori di `b`.

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

Scritture OOB

3.2 Pattern pericolosi

Vediamo in questa sezione alcuni pattern pericolosi che possono dare origine a buffer overflows.

- **senza bound checking**

- `gets` → funzione deprecata, prende come unico parametro un puntatore ad un array di char, da cui legge l'input; non esiste alcun modo per dire a `gets` quando fermarsi (si ferma quando trova *a capo*)
- `scanf %s` → con `%s` non verifica che la lunghezza dell'input sia minore della dimensione del buffer; tuttavia, è possibile usarla in modo sicuro specificando la dimensione
- `sprintf` → scrive l'output in un buffer; se la stringa generata è più lunga del buffer c'è un overflow
- `strcpy` → copia la stringa da un buffer a un altro

- **bound checking usato impropriamente** (la dimensione che specifico del buffer può essere più grande di quella effettiva)

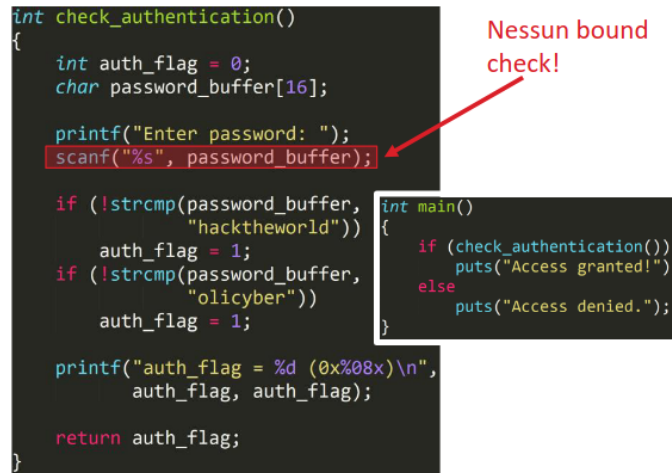
- `fgets`
- `snprintf`
- `strncpy`

- **manipolazioni manuali**

- loop di copia
- accessi ad array

3.3 Un primo overflow

Abbiamo un *main* semplice che chiama la funzione `check_authentication()`; se la funzione ritorna un valore diverso da 0 l'accesso viene consentito, altrimenti l'accesso è negato.



```
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;
}

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

Nessun bound check!

Il nostro obiettivo è ottenere l'accesso.

Il programma chiede una password e la legge in maniera insicura usando `scanf %s`; dopo aver letto la password la confronta con le password che conosce, e se matcha con una di esse mette la variabile `auth_flag = 1`; al termine del programma ritorna questo valore.

Come possiamo fare?

Le variabili nello stack sono disposte in ordine inverso rispetto a quello di dichiarazione, quindi `password_buffer` sarà prima di `auth_flag`.

Possiamo dunque fare overflow per sovrascrivere `auth_flag` impostandola ad un valore diverso da 0, in modo da ottenere l'accesso.

Nell'esempio, viene fatto overflow degli ultimi due caratteri dell'input.

```
~/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: 012345678901234567890123456789
auth_flag = 14648 (0x00003938) ← Overflow dei
Access granted!                  caratteri 89
```