SAPIENZA
UNIVERSITÀ DI ROMA

Proprietà di sicurezza nel linguaggio Rust

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

Candidato

Edoardo De Matteis
Matricola 1746561

Relatore

Pietro Cenciarelli

Anno Accademico 2019/2020

Tesi non ancora discussa

Proprietà di sicurezza nel linguaggio Rust
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Versione: 19 settembre 2020

Email dell'autore: edoardodematteis@icloud.com

Indice

Capitolo 1

Introduzione

Il linguaggio di programmazione C è stato sviluppato nel 1978 e ad oggi è ancora uno dei più popolari [17] [13], ciò è dovuto al fatto che circa il 98% [2] dei sistemi informatici sul mercato sono sistemi integrati che richiedono un basso overhead e C risponde a questa esigenza tramite una gestione manuale della memoria. Allo stesso tempo C permette di scrivere programmi non sicuri proprio a causa della gestione esplicita della memoria, esistono linguaggi considerati sicuri quali Java che gestiscono automaticamente la memoria tramite **garbage collection** (definizione 1.1) al prezzo di un maggiore overhead e sono quindi poco adatti alla scrittura di codice per embedded systems.

Garbage collection

Definizione 1.1. È una delle tecniche di gestione automatica della memoria più usate nei linguaggi di programmazione: tramite un modulo noto come **garbage collector** si tiene traccia di ogni allocazione e le zone di memoria non più necessarie vengono periodicamente liberate, rendendole nuovamente disponibili. La garbage collection ha il vantaggio di esonerare il programmatore dal dover gestire la memoria così da potersi concentrare sulla logica del programma e la sua leggibilità, si evitano inoltre vulnerabilità quali double free o dangling pointer (rispettivamente sezioni 3.3 e 3.4). Con la garbage collection l'esecuzione dei programmi è però più lenta data la presenza del garbace collector.

Nel breve futuro ci si aspetta sempre un numero crescente di oggetti d'uso comune connessi tra loro (i.e. internet of things) e si sente la necessità di metodi sicuri per programmare sistemi integrati. Nella ricerca di un linguaggio sicuro con gestione esplicita della memoria Rust sembra essere il miglior candidato: il sistema dei tipi di Rust è detto lineare, questa scelta di sviluppo prende forma nei meccanismi di **ownership** e **borrowing** (definizioni 1.2 e 1.3) e **lifetime**.

Nella sintassi e nella semantica di un sistema tipato (capitolo 4) è presente il contesto Γ ovvero un insieme composto da coppie nella forma $x : T$ con x variabile e T tipo ed un operatore "," che concatena due o più contesti tra loro in un nuovo contesto. Non sono ammesse ripetizioni di variabili e si assume che venga loro applicata automaticamente una ridenominazione, l'α-regola garantisce che il contesto

manterrà la sua consistenza. Consideriamo le tre seguenti proprietà base dette *strutturali*:

Lemma 1.0.1 (Exchange) *All'interno di un contesto l'ordine delle coppie è irrilevante.*

$$\Gamma_1, x_1 : T_1, x_2 : T_2, \Gamma_2 \vdash t : T \longrightarrow \Gamma_1, x_2 : T_2, x_1 : T_1, \Gamma_2 \vdash t : T$$

Lemma 1.0.2 (Weakening) *L'aggiunta nel contesto di coppie non necessarie è irrilevante ai fini del calcolo dei sequenti.*

$$\Gamma_1, \Gamma_2 \vdash t : T \longrightarrow \Gamma_1, x_1 : T_1, \Gamma_2 \vdash t : T$$

Lemma 1.0.3 (Contraction) *Se abbiamo una derivazione con due assunzioni identiche allora possiamo avere lo stesso risultato usandone una sola.*

$$\Gamma_1, x_2 : T_1, x_3 : T_1, \Gamma_2 \vdash t : T_2 \longrightarrow \Gamma_1, x_1 : T_1, \Gamma_2 \vdash \sigma(t) : T$$

Sia σ la sostituzione $\{x_2 \mapsto x_1, x_3 \mapsto x_1\}$.

Una logica è detta *substrutturale* quando almeno una delle seguenti proprietà è omessa e se ad essere omesse sono sia weakening che contraction si parla di logica lineare [4] e, contrariamente alle logiche tradizionali nelle quali si ha a che fare con la verità di proposizioni, le logiche lineari trattano la disponibilità di risorse. Tony Hoare nel 1987 pose questo esempio: supponiamo di rappresentare in logica del primo ordine con la proposizione *candy* il fatto di avere una caramella e con la proposizione *$1* il possesso di un dollaro, per esprimere l'atto di compravendita di una caramella possiamo scrivere $\$1 \rightarrow candy$.

$$\frac{\$1 \rightarrow candy \qquad \$1}{candy} \text{ modus ponens}$$

Se ne deduce $\$1 \wedge candy$ e si ha una caramella senza aver pagato, si può evitare questo problema con modellazioni differenti della base di conoscenza ma si incorre nel *frame problem* [1].

Un sistema lineare [18], basato quindi su logica lineare, permettendo exchange ma non weakening né contraction garantisce che ogni variabile venga usata al più una volta, a tal fine nella sintassi di una logica lineare sono presenti dei qualificatori che indicano se la variabile marcata possa essere usata solo una volta o meno, in Rust ad esempio sui tipi primitivi non è posta alcuna restrizione.

Per garantire la non riusabilità di una variabile vengono imposte due invarianti:

1. Le variabili lineari sono usate esattamente una volta per ogni cammino nel diagramma di flusso.

2. Espressioni unrestricted non possono contenere espressioni lineari; in generale espressioni meno restrittive non possono contenere espressioni più restrittive.

[1] In intelligenza artificiale è il problema di dover rappresentare una conoscenza in logica senza ricorrere a numerosi assiomi i quali indicano solo che l'ambiente non cambia arbitrariamente.

Il perché la prima invariante sia fondamentale è chiaro, per la seconda supponiamo di avere un oggetto `X` unrestricted con un attributo `X.y` lineare, sarebbe possibile sfruttare la permissività di `X` per usare più volte `X.y`, si avrebbe in questo una violazione della linearità.

Ownership

Definizione 1.2. L'ownership rappresenta il possesso di un right value da parte di un left value (rispettivamente l'insieme delle espressioni che compaiono a destra e sinistra di un assegnamento). Al momento dell'inizializzazione di un oggetto di tipo non primitivo la sua variabile, se presente, è l'unico owner e con un assegnamento ad una seconda variabile la prima perde di validità. Un owner viene invalidato quando il suo scope termina, bisogna porre particolare attenzione in questo caso dato che le chiamate di funzioni possono invalidare una variabile passatavi come argomento: l'ownership è passata alla nuova variabile nel corpo della funzione e una volta terminato il suo scope non sarà più possibile accedere all'oggetto.

```
fn destroy(c: Box<i32>) {
    println!("c: {}", c); // "c: 13"
} // c esce dal suo scope

fn main() {
    let a = Box::new(13i32);
    println!("a: {}", a); // "a: 13"

    let b = a;
    //println!("a: {}", a); // ERRORE! a non e' piu' valida
    println!("b: {}", b); // "b: 13"

    destroy(b);

    //println!("b: {}", b); // ERRORE! b non e' piu' valida
}
```

Listing 1.1. Ownership

Borrowing

Definizione 1.3. L'ownership è fondamentale per garantire una gestione automatica e sicura della memoria senza garbage collector ma non è sempre desiderabile prendere possesso di un oggetto, in questi casi si ricorre al borrowing tramite il quale si possono creare molteplici riferimenti [8, 4.2] ad un dato purché in sola lettura, tramite il modificatore `mut` si può creare un riferimento modificabile a patto che in qualsiasi momento per ogni variabile se ne abbia al più uno. Un modulo chiamato **borrow checker** garantisce a tempo di compilazione che finché esistono dei riferimenti ad un oggetto l'oggetto stesso non possa essere distrutto.

```
fn borrow(c: &Box<i32>) {
    println!("c: {}", c); // "c: 13"
} // c esce dal suo scope

fn main() {
    let a = Box::new(13i32);
    let b = &a;

    println!("b: {}", b); // "b: 13"
    borrow(b); // "c: 13"
    println!("a: {}", a); // "a: 13"

}
```

Listing 1.2. Borrowing

Un lifetime è un costrutto che il borrow checker usa per garantire che ogni riferimento sia valido, il lifetime di una variabile inizia quando viene creata e termina quando viene distrutta, quando possibile il compilatore inferisce i lifetime e in caso contrario è necessario gestirli manualmente. È paragonabile allo scope di una variabile ma non sono esattamente la stessa cosa, nel listato 1.3 si può notare come le variabili `x` e `y` abbiano due scope differenti ma lo stesso lifetime `'a`.

```
struct Foo<'a> {
    x : &'a i32,
}

fn main() {
    let & mut x;
    {
        let y = &10;
        let z = Foo {x: &y};
        x = z.x;
    }
    println!("{}", x);
}
```

```
$ rustc life.rs -o ./bin/life_rs
$ ./bin/life_rs
10
```

Listing 1.3. Lifetime e scope

Si parla di **outliving** quando per una variabile è possibile mantenere il riferimento ad un'altra variabile fuori dal suo scope. Nell'ownership (listato 1.4) chiaramente non ci sono problemi perché il possesso viene semplicemente trasferito, nel borrowing invece interviene il borrow checker (listato 1.5)

```
fn main() {
    let x:i32;
    {
        let y = 12;
        x = y;
    }
    println!("{}", x);
}
```

```
$ rustc life_own.rs -o ./bin/life_own_rs
$ ./bin/life_own_rs
12
```

Listing 1.4. Lifetime e ownership

```
fn main() {
    let x: &i32;
    {
        let y = 12;
        x = &y;
        println!("inner y: {}", y);
        println!("inner x: {}", x);
    }
    println!("outer x: {}", x);
}
```

```
$ rustc life_bor.rs -o ./bin/life_bor_rs
error[E0597]: `y` does not live long enough
 --> life_bor.rs:5:13
  |
5 |         x = &y;
  |             ^^ borrowed value does not live long enough
...
8 |     }
  |     - `y` dropped here while still borrowed
9 |     println!("outer x: {}", x);
  |                             - borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.
```

Listing 1.5. Outliving, il borrow checker interviene.

Si può eludere il controllo dell'outliving del borrow checker cambiando il tipo di `y` da `i32` a `&i32` (listato 1.6) purché il dato in questione sia di tipo primitivo e non composto (listato 1.7).

Una **temporary** è una variabile anonima inizializzata con il risultato di un'espressione, verrà usata per le successive valutazioni e perderà validità solo al termine del suo scope come qualsiasi variabile. A `Box::new(12)` corrisponde una temporary e quando si tenta di leggerne il valore fuori dal suo scope il compilatore segnala l'errore. Questo problema non si presenta con i tipi primitivi perché evidentemente Rust

esegue delle **deep copy**, si ha infatti un comportamento simile più all'ownership che al borrowing.

```
fn main() {
    let x: &i32;
    {
        let y = &12;
        x = y;
        println!("inner y: {}", y);
        println!("inner x: {}", x);
    }
    println!("outer x: {}", x);
}
```

```
$ rustc life_bor_allowed.rs -o ./bin/life_bor_allowed_rs
$ ./bin/life_bor_allowed_rs
inner y: 12
inner x: 12
outer x: 12
```

Listing 1.6. Lifetime e borrowing tipi primitivi.

```
fn main() {
    let x: &Box<i32>;
    {
        let y = &Box::new(12);
        x = y;
        println!("inner y: {}", y);
        println!("inner x: {}", x);
    }
    println!("outer x: {}", x);
}
```

```
$ rustc life_temp.rs -o ./bin/life_temp__rs
error[E0716]: temporary value dropped while borrowed
 --> life_temp.rs:4:18
  |
4 |         let y = &Box::new(12);
  |                  ^^^^^^^^^^^^ creates a temporary which is freed
    while still in use
...
8 |     }
  |     - temporary value is freed at the end of this statement
9 |     println!("outer x: {}", x);
  |                             - borrow later used here
  |
  = note: consider using a `let` binding to create a longer lived
    value

error: aborting due to previous error

For more information about this error, try `rustc --explain E0716`.
```

Listing 1.7. Lifetime e borrowing tipi composti.

Quando a seguito di un assegnamento la variabile memorizza un riferimento ad un oggetto si parla di **shallow copy**, se invece si memorizza il dato per intero si ha una una **deep copy**. In Rust per i dati di tipo primitivo si eseguono sempre

deep copy mentre per gli oggetti composti si eseguono di default shallow copy, è comunque possibile copiare l'oggetto nella sua interezza tramite il **trait** (simile a quello che in Java è un'interfaccia) `Clone`; questo avviene poiché di un dato di tipo primitivo la dimensione è nota a tempo di compilazione e non si può dire lo stesso degli oggetti composti. Nel listato 1.8 `x` e `y` puntano a due locazioni differenti.

```
fn main () {
    let x = 13;
    let x = y; // deep copy
}
```

Listing 1.8. Ownership con tipi primitivi.

I controlli su ownership e borrowing sono eseguiti a tempo di compilazione rendendo l'esecuzione molto più rapida e sicura, questi vantaggi si ottengono al prezzo di un linguaggio più verboso e che richiede maggior impegno cognitivo da parte del programmatore.

Capitolo 2

Security policy

Risulta molto difficile parlare di sicurezza in generale dato che non ne esiste una definizione assoluta ma è un concetto relativo che varia in base a cosa ci interessa proteggere e garantire. La sicurezza di un'organizzazione - che essa sia una multinazionale o un singolo individuo - è definita da una **security policy** ovvero un documento contenente regole, principi e pratiche che determinano come garantire che il sistema si trovi in uno stato sicuro [9].

Security policy

Definizione 2.1. In una politica di sicurezza si definiscono quali azioni i **principal** possono eseguire sugli oggetti. Un principal è un'entità qualsiasi che interagisce con il sistema e deve rispettare la politica di sicurezza, nel caso specifico dei linguaggi di programmazione saranno entità software.

Nello sviluppo di linguaggi di programmazione sicuri si è guidati da due principi [10]:

- **Trusted computing base (TCB)**. Un sistema presenta componenti critici per il rispetto della policy, questi formano il TCB ed è fondamentale non presenti vulnerabilità dato che metterebbero a repentaglio la sicurezza del sistema stesso, il TCB deve essere semplice e limitato così da poterne verificare facilmente la correttezza ed evitare fallimenti.
- **Principle of Least Privilege (POLP)**. Durante l'esecuzione ogni entità deve avere il numero minore possibile di permessi necessari per eseguire il suo compito.

È possibile scomporre i requisiti di sicurezza fondamentali durante lo sviluppo in **access control** e **information flow**, il primo limita chi o cosa possa accedere a quali risorse e il secondo definisce quali operazioni siano corrette o meno in seguito ad un accesso conforme all'accesso control, di norma il sistema operativo stesso implementa questi controlli via software o hardware e programmi che sono legali in C potrebbero non esserlo per l'OS. In Rust la sicurezza è garantita da regole semplici e verificabili e si rispetta il POLP, i controlli sono eseguiti a tempo di compilazione e anche quando un programma viene interrotto a runtime non non viene scomodato il sistema operativo (vedere ad esempio il codice nel listato 3.4).

Questa necessità di implementare la sicurezza nel linguaggio stesso e non rimetterla più nelle mani del sistema operativo è nata dall'avvento di internet e la sempre più grande condivisione di codice potenzialmente pericoloso.

Per dimostrare delle proprietà di un linguaggio serve una definizione formale della sua **semantica** ovvero l'insieme delle regole che definiscono il significato di un linguaggio e dei suoi programmi. A tal proposito Rust non ha una semantica formale (figura 2.1) e la sua definizione di sicurezza è poco chiara, un programma è sicuro se rispetta **memory safety** (capitolo 3) e **semantica statica** (capitolo 4) [16, 14] ma la documentazione presenta delle definizioni poco chiare se non addirittura contraddittorie:

"Rust code is incorrect if it exhibits any of the behaviors in the following list. This includes code within `unsafe` blocks and `unsafe` functions. `unsafe` only means that avoiding undefined behavior is on the programmer; it does not change anything about the fact that Rust programs must never cause undefined behavior." [16, 14.3]

"Del codice in Rust non è corretto se presenta almeno uno dei comportamenti in lista. Questo vale anche per codice in blocchi `unsafe` e funzioni `unsafe`. `unsafe` significa solo che evitare undefined behaviour è compito del programmatore; non cambia alcuna garanzia sul fatto che i programmi in Rust non debbano mai causare undefined behaviour."

L'uso del termine "unsafe" è causa di confusione dal momento che `unsafe` è un costrutto sintattico che consente la scrittura di codice che può causare **undefined behavior**, per questo si dice che Rust è composto da due "sottolinguaggi": *unsafe Rust* e *safe Rust* in base alla presenza o meno di un blocco `unsafe` nel codice, questo costrutto è desiderabile perché permette sia di poter comunicare con librerie in *C* che di avere un accesso più esplicito alla memoria [14, 1.2]; piuttosto che una definizione formale di undefined behavior si ha una lista in continuo aggiornamento [16, 14.3].

Nella citazione riportata sono presenti fonti di confusione: nella stessa frase viene detto prima che in un blocco `unsafe` sia compito del programmatore evitare undefined behavior e subito dopo viene affermato come le garanzie sulla safety siano le stesse di safe Rust, quindi anche l'assenza di undefined behavior. Volendo comunque considerare solo l'ultima affermazione, l'esecuzione in un blocco `unsafe` di uno degli esempi in lista non dovrebbe causare undefined behavior; nel listato 2.1 si ha un **raw pointer dereferencing** (presente in lista come dimostrato in figura 2.1): è possibile leggere il valore memorizzato in una locazione di memoria tramite il suo indirizzo come in C.

È ragionevole che questo sia possibile solo in unsafe Rust ma mette in luce come la documentazione sia poco chiara.

```
1 fn main() {
2     unsafe {
3         let x = 42;
4         let ptr = &x as *const i32;
5         println!("Dereferencing: {}", *ptr);
6     }
7 }
```

```
$ rustc raw.rs -o bin/raw
$ ./bin/raw
Dereferencing: 42
```

Listing 2.1. Raw pointer dereferencing in un blocco `unsafe`

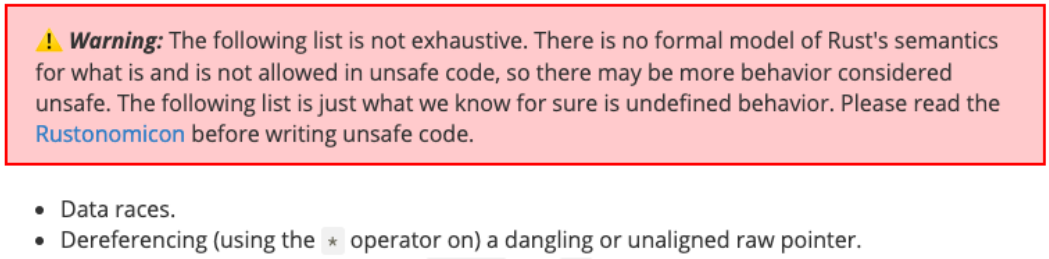

Figura 2.1. Il raw pointer dereferencing è considerato undefined behavior

Capitolo 3

Memory Safety

Durante l'esecuzione di un programma ogni oggetto ha associato un insieme finito di locazioni di memoria cui accedere noto come **address space**, se in un programma nessun oggetto accede mai ad indirizzi fuori dal proprio address space allora è **memory safe** e un linguaggio che garantisce la memory safety dei suoi programmi è detto a sua volta memory safe.

L'address space può essere definito a tempo di compilazione o esecuzione dipendentemente dal tipo di allocazione, in C e Rust ne consideriamo tre tipi:

- **Allocazione statica**. Il compilatore memorizza solo ed esclusivamente le variabili globali ovvero quelle il cui scope corrisponde a tutto il file, si possono definire tramite un modificatore (in C è `static`, da cui il nome) o definendole fuori da ogni funzione.
- **Allocazione automatica**. Si usa per variabili locali e di tipo primitivo. Una variabile è locale quando dichiarata all'interno di una funzione (quindi valida solo in un determinato scope), dal momento che l'ordine di esecuzione non è noto a tempo di compilazione l'allocazione avviene a tempo di esecuzione e si utilizza una struttura dati nota come **stack** o **call stack**. Spesso il nome call stack viene utilizzato per le sole chiamate di funzione.
- **Allocazione dinamica**. L'allocazione dinamica in C è esplicita tramite funzioni come `malloc` e permette di allocare memoria ad un oggetto durante l'esecuzione il che si rivela molto utile per oggetti che non hanno una dimensione fissa. In safe Rust non si hanno funzioni come in C per questioni di sicurezza, si ha però un tipo `Box`, si usa una struttura dati chiamata **heap** che ha la particolarità di crescere verso l'alto contrariamente allo stack che cresce verso il basso e può essere problematico in caso di overflow (sezione 3.1). Un tipo composto è definito dall'unione di tipi primitivi e non è banale prevedere la dimensione di un oggetto composto quindi vengono allocati sull'heap. Un oggetto di tipo primitivo viene salvato sullo stack.

Questo approccio è stato reso popolare dal C ed è oggi adottato da numerosi linguaggi, sempre in C esiste un ulteriore tipo di allocazione della memoria noto come **register allocation** che permette di scrivere direttamente su un blocco del processore, non essendo presente in Rust non viene approfondita.

3.1 Buffer Overflow

Un **buffer** è una qualsiasi zona contigua di memoria contenente istanze dello stesso dato e ne definisce i limiti, se durante l'esecuzione si riescono a superare si parla di **buffer overflow** ed è possibile leggere o scrivere oltre il proprio address space. Un esempio di buffer overflow in C è nel listato 3.1 [15, 7.5]: dato che gli array sono posizionati uno dopo l'altro (in ordine LIFO essendo dati allocati su uno stack) si riesce a leggere e scrivere su `str1` eludendo anche il controllo di uguaglianza, nel terzo caso questo controllo viene superato perché si confrontano solo i primi 8 caratteri. In Rust questo non è possibile, si veda nel listato 3.2 come il linguaggio esegua dei controlli sulle dimensioni degli array, esattamente ciò che la funzione `gets` non fa.

```c
#include <stdio.h>
#include <string.h>

int main(int argc,char* argv[]) {
  int valid = 0;
  char str1[8];
  char str2[8];

  strcpy(str1, "START");
  gets(str2);
  if(strncmp(str1, str2, 8) == 0) valid = 1;
  printf("str1:%s, str2:%s, valid:%d\n", str1, str2, valid);
}
```

```
$ ./bin/stackattack_c
warning: this program uses gets(), which is unsafe.
START
str1:START, str2:START, valid:1

$ ./bin/stackattack_c
warning: this program uses gets(), which is unsafe.
EVILINPUTVALUE
str1:TVALUE, str2:EVILINPUTVALUE, valid:0

$ ./bin/stackattack_c
warning: this program uses gets(), which is unsafe.
BADINPUTBADINPUT
str1:BADINPUT, str2:BADINPUTBADINPUT, valid:1
```

Listing 3.1. Buffer overflow in C

```rust
fn main() {
    let arr: [i32; 2] = [13, 10];
    arr[4];
}
```

```
$ rustc bufof.rs -o ./bin/bufof_rs
error: index out of bounds: the len is 2 but the index is 4
 --> bufof.rs:3:5
  |
3 |     arr[4];
  |     ^^^^^^
  |
```

```
  = note: `#[deny(const_err)]` on by default

error: aborting due to previous error
```

Listing 3.2. Buffer overflow in Rust

La pericolosità del buffer overflow è dovuta alla possibilità di eseguire codice arbitrario ovvero **code injection**, nel listato 3.3 se ne ha un esempio molto semplice, per la stringa in input non si ha alcun tipo di bound check ed è possibile scrivere un comando che verrà eseguito da `system()` con gli stessi permessi con cui si esegue il file.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv) {
 char cat[] = "cat ";
 char *command;
 size_t commandLength;

 commandLength = strlen(cat) + strlen(argv[1]) + 1;
 command = (char *) malloc(commandLength);
 strncpy(command, cat, commandLength);
 strncat(command, argv[1], (commandLength - strlen(cat)) );

 system(command);
 return (0);
}
```

```
$ ./bin/inj file.txt
Nel mezzo del cammin di nostra vita...
$ ./bin/inj "file.txt; ls"
Nel mezzo del cammin di nostra vita...
bin             file.txt        inj.c           output
```

Listing 3.3. Code injection in C

Esistono code injection più sofisticati che sfruttando le istruzioni in assembly di un programma possono eseguire codice arbitrario sfruttando la rappresentazione in memoria di un processo [11], questo è costituito da tre regioni:

- **Text**. Contiene le istruzioni da eseguire e dati in sola lettura, quest'area è read-only quindi un tentativo di scrittura è intercettato dal sistema operativo.
- **Data**. Contiene dati inizializzati o meno con scope globale, è in quest'area che vengono salvati i dati quando si esegue un'allocazione statica.
- **Stack**. Lo stack sul quale si salvano i dati con allocazione automatica. Proprio grazie allo stack si può sfruttare un buffer overflow per puntare ad un processo differente ed eseguire codice non desiderato.

3.2 Integer Overflow

In qualsiasi macchina non astratta si ha memoria finita ed è rappresentabile solo un insieme finito di numeri, quando un valore è troppo grande (o piccolo) per essere rappresentato si ha un overflow. Sono due gli approcci principali per risolvere un integer overflow, ognuno basato su un'aritmetica differente:

- **Modular arithmetic**. Si applica un wrapping ad ogni numero in overflow, dato il valore n e una memoria a m bit si memorizza $w = n \mod m$.
- **Saturation arithmetic**. Si applica un clamp ovvero dati min e max - rispettivamente il numero più grande e più piccolo rappresentabile - e un numero n in memoria viene salvato

$$c = \begin{cases} max & n > max \\ min & n < min \\ n & \text{altrimenti} \end{cases}$$

L'integer overflow ha in passato causato problemi molto seri: durante il volo inaugurale del lanciatore *Ariane 5* un integer overflow dovuto ad una conversione in intero a 16 bit di un float a 64 bit ha causato una reazione a catena per cui il razzo ha virato orizzontalmente distruggendosi poco dopo il lancio, il codice scritto in Ada non prevedeva controlli di overflow come esplicitamente richiesto dai progettisti per motivi di efficienza. In Rust si hanno due modalità di compilazione: in **debug mode** si eseguono dei controlli in più rispetto alla **release mode** tra cui controlli dinamici per l'integer overflow - listato 3.4 - contrariamente al C che applica direttamente wrapping.

Differenti linguaggi adottano differenti approcci: se si può prevedere il valore massimo possibile in un programma allora è in generale facile evitare integer overflow; in caso contrario si possono utilizzare metodi dinamici come in Rust.

```rust
fn main() {
    let mut n = 1;
    let mut i = 1;

    while n > 0 {
        n = n*2;
        i = i+1;
        println!("{} bit architecture.", i);
    }
}
```

```
$  ./bin/intof_rs
2 bit architecture.
3 bit architecture.
        ...
30 bit architecture.
31 bit architecture.
thread 'main' panicked at 'attempt to multiply with overflow', intof.
    rs:6:13
note: run with `RUST_BACKTRACE=1` environment variable to display a
    backtrace.
```

Listing 3.4. Integer overflow in Rust

Nel listato 3.5 si prende un intero come primo argomento e una stringa - idealmente di lunghezza pari al primo argomento - come secondo. Il problema sorge nella conversione da integer a short, inserendo come primo argomento 65536 (2^{16} che non è rappresentabile con i soli 16 bit di uno short) si causa un buffer overflow, combinandone la pericolosità con la difficoltà di rivelamento dell'integer overflow.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3) return -1;
    i = atoi(argv[1]);
    s = i;

    if(s >= 80) return -1;

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
```

```
$ ./bin/width1 65536 hello
s = 0
[1]    20475 illegal hardware instruction  ./bin/width1 65536 hello
```

Listing 3.5. Integer overflow in C

3.3 Double-Free

Si ha un double free error [5, 10.4.4] quando si prova a liberare più volte la stessa zona di memoria e in Rust non è possibile grazie all'ownership, nel listato 3.6 si può vedere come `s1` perda validità dopo aver trasferito a `s2` il possesso della stringa cui puntava.

```rust
fn main() {
    let s1 = String::from("Primo!");
    let _s2 = s1;
    s1;
}
```

```
$ rustc doublefree.rs -o bin/doublefree_rs
error[E0382]: use of moved value: ‘s1‘
 --> doublefree.rs:4:5
  |
2 |     let s1 = String::from("Primo!");
  |         -- move occurs because ‘s1‘ has type ‘std::string::String
    ‘, which does not implement the ‘Copy‘ trait
```

```
3 |     let _s2 = s1;
  |               -- value moved here
4 |     s1;
  |     ^^ value used here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

Listing 3.6. Double free in Rust

In C invece si può osservare il codice nel listato 3.7: si immagini di avere un servizio ad iscrizione salvando ogni utente tramite una struct `User`: Guido libera la propria memoria che, disponibile, potrà memorizzare `Luisa` che si è appena iscritta. Il puntatore `Guido` però ora punta all'account di Luisa e con `free(Guido)` proprio l'account di Luisa viene eliminato con il risultato che non solo Luisa non ha più un account ma a quello di Carla possono accedere sia Guido che Luisa.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct User User;

struct User {
    User * self;
    char name[8];
};

int main(int argc, char** argv) {

    // (1) Guido si iscrive
    struct User * Guido = malloc(sizeof(struct User));
    Guido->self = Guido;
    strcpy(Guido->name, "Guido");

    printf(
        "Guido: [self: %p, name: %s]\n",
        Guido->self, Guido->name
    );

    // (2) Guido libera la sua memoria
    free(Guido);

    // (3) Luisa si iscrive
    struct User * Luisa = malloc(sizeof(struct User));
    Luisa->self = Luisa;
    strcpy(Luisa->name, "Luisa");

    printf(
        "Luisa: [self: %p, name: %s]\n",
        Luisa->self, Luisa->name
    );

    // (4) Guido libera la memoria di Luisa
    free(Guido);

    // (5) Carla si iscrive sulla memoria di Luisa
```

```
    struct User * Carla = malloc(sizeof(struct User));
    Carla->self = Carla;
    strcpy(Carla->name, "Carla");

    printf(
        "Carla: [self: %p, name: %s]\n",
        Carla->self, Carla->name
    );
    printf(
        "Luisa: [self: %p, name: %s]\n",
        Luisa->self, Luisa->name
    );
}
```

```
./bin/doublefree_c
Guido: [self: 0x7f9f72c05a30, name: Guido]
Luisa: [self: 0x7f9f72c05a30, name: Luisa]
Carla: [self: 0x7f9f72c05a30, name: Carla]
Luisa: [self: 0x7f9f72c05a30, name: Carla]
```

Listing 3.7. Double free in C

Il puntatore `Guido` è anche un **dangling pointer** (sezione 3.4).

3.4 Dangling References

Quando un oggetto viene eliminato ma il suo puntatore no si ha un dangling pointer che permettere di accedere a memoria cui non si dovrebbe.

Immaginiamo ora che per vendicarsi Luisa abbia scritto un semplice sistema di messaggistica per leggere la corrispondenza di Guido, la procedura `send_message` crea un puntatore di tipo `Message` che ha scope solo ed esclusivamente dentro `send_message` ma facendo riferimento esplicito a quella locazione Luisa riesce a leggere il messaggio di Guido anche dopo che questo non esiste più. Nel listato 3.8 l'indirizzo viene stampato a schermo e inserito dall'utente (indicato da ">") per semplicità.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Message {
  char from[8];
  char to[8];
  char text[32];
};

void send_message(char * from, char * to, char * text) {
  struct Message * mess = malloc(sizeof(struct Message));
  strcpy(mess->from, from);
  strcpy(mess->to, to);
  strcpy(mess->text, text);
  printf("%p\n", mess);
}

int main() {
```

```
  send_message("Guido", "Carla", "asa nisi masa");

  char * ptr;
  printf("> ");
  scanf("%p", &ptr);

  printf("[%s, %s, %s]\n", ptr, ptr+8, ptr+16);
}
```

```
$ ./bin/dangref_c
0x7fec9fe04d10
> 0x7fec9fe04d10
[Guido, Carla, asa nisi masa]
```

Listing 3.8. Dangling pointer in C

In Rust non possiamo usare i puntatori perché l'ownership ci obbliga ad avere un puntatore solo e una volta disabilitato non possiamo più accedere a quell'area di memoria né si può leggere la locazione perché non è permesso il raw dereferencing; possiamo usare però le reference, nel listato 3.9 `s1` è una shallow copy di `s2`, la prima però sopravvive alla seconda e il borrow checker interviene, non è possibile quindi avere dangling reference.

```
fn main() {
    let s1: &String;
    {
        let s2 = String::from("Hello");
        s1 = &s2;
    }
    println!("{}", s1);
}
```

```
$ rustc dangref.rs -o bin/dangref_rs
error[E0597]: `s2` does not live long enough
 --> dangref.rs:5:14
  |
5 |         s1 = &s2;
  |              ^^^ borrowed value does not live long enough
6 |     }
  |     - `s2` dropped here while still borrowed
7 |     println!("{}", s1);
  |                    -- borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.
```

Listing 3.9. Dangling reference in Rust

Capitolo 4

Type Safety

Ogni oggetto in matematica ha un **tipo**, basti pensare al fatto che non è possibile eseguire un'operazione come $\{13\} \wedge 10$ dato che $\{13\}$ è un insieme, 10 è un numero e $\wedge$ è un connettivo logico. In informatica un tipo è un vincolo che definisce un insieme definito di valori validi per una risorsa di un programma, in un linguaggio tipato l'insieme delle regole che assegna un tipo ad ogni espressione è detto semantica statica o **sistema dei tipi** ed è composto da quattro aspetti: **type safety**, **type casting**, **type initialization** e **immutability** [7].

Durante l'esecuzione di un programma si può incorrere in due tipi di errore: **trapped** e **untrapped**, i primi sono facilmente riconoscibili dato che interrompono l'esecuzione del programma, i secondi no e per questo sono molto più difficili da individuare. Un linguaggio in cui nessun programma ben tipato (definizione 4.1) genera errori trapped né untrapped si dice **sound**, se ne genera solo **untrapped** invece è **safe**, un linguaggio sound quindi è necessariamente type safe [3]. Una definizione analoga ma meno formale di Milner spesso adottata è la seguente.

well-typed programs cannot "go wrong" [9]

con programmi che "vanno male" si possono intendere programmi non sound o che generano undefined behavior (sezione 1).

Type checking

Definizione 4.1. La fase di verifica dei vincoli imposti dai tipi può avvenire a tempo di compilazione o a tempo di esecuzione e si parla rispettivamente di type checking statico/forte e dinamico/debole. Ad un controllo statico consegue un'ottimizzazione e per questo è desiderabile ma purtroppo non è sempre possibile verificare la correttezza di un programma a tempo di compilazione e di norma sono utilizzati sia static che dynamic type checker, in Java ad esempio i controlli sui metodi sono eseguiti dinamicamente.
Un programma che supera la fase di type checking è ben tipato.

4.1 Type casting

Quando si esegue un'operazione tra oggetti di tipo differente può rivelarsi necessaria una conversione di tipo (type casting) e può essere esplicita o implicita, nel secondo caso si parla di **coercion**. Tipicamente i linguaggi con type checking forte eseguono pochi casting impliciti e quelli con type checking dinamico molti, Rust è fortemente tipato e non c'è coercion invece in C che è debolmente tipato sì.

```c
#include <stdio.h>

int main() {
    int x = 12;
    printf("%f\n", x * 0.5);
}
```

```
$ gcc coercion.c -o ./bin/coercion_c
$ ./bin/coercion_c
6.000000
```

Listing 4.1. Casting implicito in C

```rust
fn main() {
    let x = 12;
    // println!("{}", x * 0.5); // ERRORE!
    println!("{}", x as f32 * 0.5);
}
```

```
$ rustc coercion.rs -o ./bin/coercion_rs
$ ./bin/coercion_rs
6
```

Listing 4.2. Casting in Rust

Rust non ha casting implicito perché può portare a vulnerabilità, un esempio di questa vulnerabilità è un bug del 2002 nel sistema operativo FreeBSD per cui in alcune chiamate di sistema si assumeva a priori che il valore in input fosse un intero positivo nonostante nella definizione della funzione il tipo fosse `int`. I programmatori data l'assunzione fatta non gestivano il caso di un intero negativo tramite boundary check, questo valore veniva poi passato a `memcpy()` che prende un `unsigned int` e a causa della coercion un valore come -1 veniva convertito in $2^{32} - 1$, valore che non avrebbe superato il boundary check. Inserendo numeri negativi abbastanza grandi era possibile copiare una porzione di memoria riservata al kernel [12]. Per chiarezza nel listato 4.1 è stato riproposto del codice a grandi linee, il problema è la conversione di `maxlen` che supera il controllo a riga 11 e viene convertita da intero a intero positivo a riga 12.

```c
#include <stdlib.h>

#define KSIZE 1024

char kbuf[KSIZE];

void *memcpy(void *dest, void *src, size_t n);

int copy_from_kernel(void *user_dest, int maxlen) {
```

```
    // len=min(KSIZE, maxlen)
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

4.2 Type initialization

In C quando si dichiara una variabile è buona norma inizializzarla perché il valore assegnatale è quello già presente nelle locazioni di memoria assegnate. Fortunatamente ad oggi C è un caso raro tra i linguaggi di programmazione e ad una variabile appena dichiarata viene assegnato un valore di default: 0 per i numeri, la stringa vuota per le stringhe e un valore speciale `NULL` per gli oggetti.

Il valore `NULL` fu introdotta da Tony Hoare nel 1964 e nel 2009 lui stesso lo definì il suo errore da un miliardo di dollari [6] perché ha portato a numerose sviste ed errori da parte dei programmatori: in C compilando con *gcc* un puntatore a `NULL` punta a una locazione 0 riservata (listato 4.3) e la sua dereferenza causa l'interruzione del programma.

```
#include <stdio.h>

int main() {
    void* ptr = NULL;
    printf("%p\n", ptr);
    printf("%d\n", * (int *) ptr);
}
```

```
$ gcc null.c -o ./bin/null_c
$ ./bin/null_c
0x0
[1]    89833 segmentation fault  ./bin/null_c
```

Listing 4.3. Null reference in C

In Rust non esiste `NULL` come in C ma si ha il tipo polimorfo `Option<T>` il quale può assumere i valori `Some<T>` e `None`. Nel listato 4.4 si può vedere come un `Box<i32>` (rappresenta un puntatore di un intero a 32 byte sull'heap) senza alcun valore punti ad un `None` di tipo `Option<Box<i32>>` e non ad un valore speciale `NULL` che si colloca meno coerentemente nel sistema dei tipi.

```
fn check_optional(optional: Option<Box<i32>>) {
    match optional {
        Some(p) => println!("Si ha il valore {}", p),
        None => println!("Non si ha alcun valore"),
    }
}

fn main() {
    let optional = None;
    check_optional(optional);
    let optional = Some(Box::new(42));
    check_optional(optional);
}
```

```
$ rustc null.rs -o ./bin/null_rs
$ ./bin/null_rs
Non si ha alcun valore
Si ha il valore 42
```

Listing 4.4. Null reference in Rust

4.3 Immutability

Un oggetto è immutabile quando durante l'esecuzione di un programma non può essere modificato, in Rust ogni oggetto è di default immutabile e lo si rende modificabile con la parola chiave `mut` coerentemente con il principio di least privilege.

In C non c'è una vera e propria immutabilità, ramite il modificatore `const` è possibile definire delle costanti che dovrebbero essere non modificabili, rimane comunque possibile dereferenziarle e tramite un puntatore modificarne il valore, l'immodificabilità assume particolare rilievo in programmi concorrenti. Nel listato 4.5 abbiamo:

1. `Misc`. Una struct composta da un intero, il cui valore non è rilevante, e un puntatore a un intero nel quale verrà salvato l'indirizzo di `accum`.

2. `accum`. Una variabile globale costante, il segmento in cui verrà allocata dipende dal compilatore ad esempio usando *gcc* è salvato nel text segment del processo insieme al codice ma in ogni caso è un'area in sola lettura.

3. `accum_mutex`.Un **mutex** o **lock** è un meccanismo di sincronizzazione per imporre la mutua esclusione ed evitare race condition, garantisce quindi che la stessa risorsa non venga modificata da più thread contemporaneamente. Il possesso e il rilascio di una forma sono effettuati mediante le due funzioni di lock e unlock.

4. `ths`. Un array di 20 thread ovvero sottoprocessi eseguiti parallelamente possibilmente condividendo delle risorse. Tramite la chiamata a `pthread_join()` si attende il termine del thread in input memorizzandone il risultato di ritorno in `ret`.

L'esecuzione del programma è interrotta dal sistema operativo perché `accum` è read-only, in C non c'è alcun controllo.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int v;
    int* ptr;
}Misc;

// global
const int accum = 0;
pthread_mutex_t accum_mutex = PTHREAD_MUTEX_INITIALIZER;

```

```c
// func
void* func(void* x) {
    Misc* xm = (Misc*) x;

    pthread_mutex_lock(&accum_mutex);
    *xm->ptr += xm->v;
    pthread_mutex_unlock(&accum_mutex);
    return NULL;
}

int main() {
    pthread_t ths[20];
    Misc misc[20];
    int i;

    for(i=0; i < 20; i++) {
        misc[i].v = i;
        misc[i].ptr = &accum;

        pthread_create(&ths[i], NULL, func, &misc[i]);
    }

    for(i=0; i < 20; i++) {
        void* res;
        pthread_join(ths[i], &res);
    }
    printf("accum = %d\n", accum);
}
```

```
$ gcc thread.c -o ./bin/thread_c
thread.c:19:10: warning: initializing 'int *' with an expression of
    type 'const int *' discards qualifiers [-Wincompatible-pointer-
    types-discards-qualifiers]
    int* ptr = &accum;
         ^     ~~~~~~
1 warning generated.
$ ./bin/thread_c
[1]    76864 bus error  ./bin/thread_c
```

Listing 4.5. Costanti e thread in C

Il programma nel listato 4.6 funziona perfettamente, in safe Rust non c'è il pericolo che un valore immutabile venga modificato da un riferimento perché non è possibile dereferenziare un puntatore, vale la pena però notare che in tutto il programma non compaia mai il modificatore `mut`. La funzione `thread::spawn()` prende in input una **closure**, è come una funzione anonima in Java e all'interno delle doppie pipe `||` si inseriscono i parametri in input, in questo caso nessuno. La parola chiave `move` dà alla closure l'ownership delle variabili che usa evitando così che un thread (`main` compreso) modifichi una risorsa usata da un altro oggetto, solo una variabile per volta potrà avere l'ownership della risorsa in questione, si garantisce la mutua esclusione e si evitano race condition a tempo di compilazione [1, 5.6]. Per la sincronizzazione tra thread si utilizzano le funzioni `channel`.

```rust
use std::thread;
use std::sync::mpsc;

```

```rust
fn main() {
    // tx is the transmitter or sender
    // rx is the receiver
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = i * i;

            tx.send(answer).unwrap();
        });
    }

    for _ in 0..10 {
        println!("{}", rx.recv().unwrap());
    }
}
```

```
$ rustc thread.rs -o ./bin/thread_rs
$ ./bin/thread_rs
0
1
4
9
16
25
36
49
81
64
```

Listing 4.6. Concorrenza in Rust

4.4 Rapporto con Memory Safety

Type safety e memory safety sono strettamente collegate tanto che a volte risulta davvero difficile distinguere tra le due e alcune vulnerabilità violano entrambe, ad esempio nel listato 4.7 si alloca un array di `short` (2 byte) sullo stack, considerando che il computer su cui ho eseguito il programma monta un processore *Intel* (little endian quindi la cifra più a destra è la meno significativa) si ha la seguente rappresentazione in memoria

1	1000000000000000
0	0000000000000000

il puntatore `p` punta al primo elemento ma essendo `int` legge 4 B

00000000000000001000000000000000

che in decimale equivale a $2^{16} = 65536$. È evidente che la causa di questa confusione sia la la lettura di un solo oggetto da 4 byte anziché due oggetti da 2, questo avviene

a causa della rappresentazione di due tipi differenti e del casting che C permette contrariamente a Rust.

```c
#include <stdio.h>

int main() {
  short arr[] = {0, 1};
  int* p = &arr[0];
  printf("%u", *p);
}
```

```
$ ./bin/lilliput_c
65536
```

Listing 4.7. Type confusion in C

Bibliografia

[1] The rust programming language. URL: https://doc.rust-lang.org/1.9.0/book/README.html.

[2] Michael Barr. Real men program in c, 2009. URL: https://www.embedded.com/real-men-program-in-c/.

[3] Pietro Cenciarelli. Dispense del corso linguaggi di programmazione, 2019. URL: http://wwwusers.di.uniroma1.it/~lpara/DISPENSE/note.pdf.

[4] Jean-Yves Girard. Linear logic. In *Theoretical Computer Science*, volume 50, pages 1–102. 1987. URL: http://girard.perso.math.cnrs.fr/linear.pdf.

[5] Dietere Gollmann. *Computer Security.* John Wiley & Sons Ltd, 3 edition, 2011.

[6] Tony Hoare. Null references: The billion dollar mistake. 2009. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.

[7] Amir A. Khwaja, Muniba Murtaza, and Hafiz F. Ahmed. A security feature framework for programming languages to minimize application layer vulnerabilities. *Security and Privacy*, 2020.

[8] Steve Klabnik and Carol Nichols. The rust programming language, 2020. URL: https://doc.rust-lang.org/book/title-page.html.

[9] Robin Milner. A theory of type polimorphism in programming. *Journal of Computer and Sciences*, 7:348–375, 4 1978.

[10] Jaime Niño. Implementing security via modern programming languages. Technical report, University of New Orleans.

[11] Aleph One. Smashing the stack for fun and profit. URL: https://insecure.org/stf/smashstack.html.

[12] Security Advisory The FreeBSD Project. Freebsd-sa-02:38.signed-error, 2002. URL: https://www.freebsd.org/security/advisories/FreeBSD-SA-02:38.signed-error.asc.

[13] Pypl popularity of programming language, 2020. URL: http://pypl.github.io/PYPL.html.

[14] Rustonomicon, 2020. URL: `https://doc.rust-lang.org/nomicon/index.html`.

[15] William Stallings. *Operating Systems: Internals and Design Principles*. Pearson Education, 7 edition, 2012.

[16] The rust reference, 2020. URL: `https://doc.rust-lang.org/reference/introduction.html`.

[17] Tiobe programming community index, 2020. URL: `https://www.tiobe.com/tiobe-index/`.

[18] David Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–43. MIT Press, 2002. URL: `https://mitpress-request.mit.edu/sites/default/files/titles/content/9780262162289_sch_0001.pdf`.