Proprietà di sicurezza nel linguaggio Rust

Edoardo De Matteis

Dicembre 2020

Problema attuale

Il 98% dei sistemi ICT sono integrati e richiedono un controllo a basso livello delle risorse e negli anni a venire questa tendenza non diminuirà. Il linguaggio più usato è C ma Rust si propone come linguaggio sicuro per embedded system.

Principi di sicurezza

Si indica con **soggetto** qualsiasi entità che richieda di accedere ad un **oggetto**. La **trusted computed base (TCB)** è la totalità delle componenti che vanno a definire la sicurezza di un sistema

Principi di sicurezza

- Least privilege.
- Fail-safe default.
- Economy of mechanisms.
- Complete mediation.
- Open design.
- Separation of privilege.
- Least common mechanism.
- Psychological acceptability.

Rust

Rust è un linguaggio di programmazione focalizzato sulla velocità d'esecuzione e sulla sicurezza dei suoi programmi tramite **ownership**, **borrowing** e **lifetime**.

Ownership

L'ownership rappresenta il possesso di un **right value** da parte di un **left value**, ad un variabile corrisponde esattamente un valore e ad un valore una sola variabile. Un assegnamento invalida l'accesso da parte della variabile precedente, se presente.
L'ownership permette di avere una gestione automatica della memoria senza garbage collector.

Borrowing

Il borrowing permette di creare molteplici riferimenti in lettura ad un dato, tramite il modificatore `mut` è possibile definire un unico riferimento modificabile. Il **borrow checker** garantisce a tempo di compilazione tramite i **lifetime** che un oggetto non venga distrutto se ne esistono dei riferimenti.

Lifetime

Un lifetime è una regione di codice nella quale ogni riferimento deve essere valido, al momento della definizione di una variabile le viene associato un lifetime che verrà poi distrutto con la distruzione della variabile.
Ad un lifetime possono essere associati più scope e ad uno scope possono essere associati più lifetime.

Lifetime

In Rust valgono le tre seguenti regole assiomatiche:

- **Association**. Lo scope di un riferimento è sottoinsieme del suo lifetime.
- **Reference**. Il lifetime associato ad un riferimento è sottoinsieme dello scope dell'oggetto cui fa riferimento.
- **Assignment**. Il lifetime associato ad un riferimento è sottoinsieme del lifetime dell'oggetto cui fa riferimento.

Sicurezza di un linguaggio

La TCB può essere scomposta in:

- Access control.
- Information flow.

Il sistema operativo esegue questi controlli e vengono bloccati alcuni programmi che in C sono legali. Internet ha reso semplice la condivisione dello stesso codice su macchine con sistemi operativi differenti, per questo la sicurezza è stata implementata direttamente nel linguaggio, come in Rust.

Security policy

La definizione di sicurezza non è assoluta ma dipende dalla security policy, in Rust consideriamo un programma sicuro se

- Non presenta codice `unsafe`.
- Non genera **undefined behavior**.
- Non è possibile dereferenziare un **raw pointer**.
- Non è possibile accedere al campo di un' `union` se non per l'inizializzazione.

Sicurezza in Rust

1. I meccanismi di sicurezza in Rust sono modellati sulla **logica lineare**.
2. La sicurezza in Rust è divisa concettualmente in **memory safety** e **type safety**.

Sistema di tipi substrutturale

Un sistema di tipi è detto **substrutturale** quando almeno una delle tre regole **strutturali** non è valida:

- **Exchange**. L'ordine degli elementi in un'ipotesi o conclusione è irrilevante.
- **Weakening**. Ipotesi e conclusione possono essere estese con affermazioni superflue.
- **Contraction**. In ipotesi e conclusione è possibile unificare due elementi unificabili.

Sistema di tipi lineare

Un sistema si dice lineare quando è valida solo exchange, valgono due invarianti:

- Le variabili lineari sono usate esattamente una volta.
- Espressioni meno restrittive non possono contenere espressioni più restrittive.

Memory Safety

Si parla di memory safety quando in un programma nessuna entità esce fuori dal proprio **address space**. È definito dall'allocazione, ne considereremo tre tipi:

- Allocazione statica.
- Allocazione dinamica.
- Allocazione automatica.

Buffer overflow

```c
#include <stdio.h>

int main() {
  int arr[] = {13, 10};
  printf("%d\n", arr[4]);
}
```

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

Integer overflow

Gli approcci principali adoperati di fronte ad un integer overflow sono i seguenti:

- Modular arithmetic.

$$w = n \mod m$$

- Saturation arithmetic.

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

Double free

Quando si libera due volte la stessa memoria si incorre in un double free error e in C è possibile avere due puntatori che puntano alla stessa locazione.
In Rust l'ownership ci impedisce a priori di avere due puntatori sullo stesso dato.

Dangling reference

Quando un oggetto viene eliminato senza liberare i suoi puntatori si ha un dangling pointer che può essere usato per accedere a memoria cui non si dovrebbe.
Rust tramite il borrow checker impedisce di avere puntatori dopo la distruzione di un oggetto e in ogni caso non è possibile dereferenziare un raw pointer.

Type safety

Gli errori si dividono in **trapped** e **untrapped** e un programma **ben tipato** può essere:

- **Sound**. Non viene generato alcun errore.
- **Safe**. Non vengono generati errori untrapped.

Si possono identificare quattro caratteristiche per definire la sicurezza di un linguaggio: **type safety**, **type casting**, **type initialization**, **immutability**.

Type casting

Una conversione di tipo può essere esplicita o implicita (**coercion**), tipicamente i linguaggi con type checking forte hanno pochi casting impliciti e quelli con type checking debole ne hanno molti. Il casting implicito può causare undefined behaviour e per questo in Rust non è presente.

Type initialization

In C una variabile non inizializzata assume il valore contenuto nel suo address space, in altri linguaggi si usa un valore speciale `NULL`. In *GCC* un puntatore a `NULL` punta alla locazione 0 che è però riservata, per questo in Rust non si ha il tipo polimorfo `Option<T>` che può assumere `Some<T>` o `None`.

```
1 fn check_optional(optional: Option<Box<i32>>) {
2     match optional {
3         Some(p) => println!("Si ha il valore {}", p),
4         None => println!("Non si ha alcun valore"),
5     }
6 }
```

Immutability

In Rust è possibile definire sia delle costanti tramite la parola chiave `const` che delle variabili, di default immutabili.
Anche in C esistono le costanti ma è possibile modificarne il valore dereferenziandone un puntatore all'interno di programmi concorrenti. Un programma del genere di norma viene interrotto dal sistema operativo.

Grazie