Proprietà di sicurezza nel linguaggio Rust

Edoardo De Matteis

Dicembre 2020

Principi di sicurezza

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

Ownership

```
1  fn destroy(c: Box<i32>) {
2      println!("c: {}", c); // "c: 13"
3  } // c esce dal suo scope
4
5  fn main() {
6      let a = Box::new(13i32);
7      println!("a: {}", a); // "a: 13"
8
9      let b = a;
10     //println!("a: {}", a); // ERRORE! a non e' piu' valida
11     println!("b: {}", b); // "b: 13"
12
13     destroy(b);
14
15     //println!("b: {}", b); // ERRORE! b non e' piu' valida
16 }
```

Borrowing

```
1  fn borrow(c: &Box<i32>) {
2      println!("c: {}", c); // "c: 13"
3  } // c esce dal suo scope
4
5  fn main() {
6      let a = Box::new(13i32);
7      let b = &a;
8
9      println!("b: {}", b); // "b: 13"
10     borrow(b); // "c: 13"
11     println!("a: {}", a); // "a: 13"
12
13 }
```

Lifetime

- **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.

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

- **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

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

Memory Safety

- 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

- Modular arithmetic.

$$w = n \mod m$$

- Saturation arithmetic.

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

Double free

```c
#include <stdlib.h>

int main() {
    char *ptr = malloc(sizeof(char));
    *ptr = 'a';
    free(ptr);
    free(ptr);
}
```

Dangling reference

```
fn main() {
    let x;
    {
        let y = &13;
        x = y;
    }
    println!("{}", x);
}
```

```
fn main() {
    let x: &Box<i32>;
    {
        let y = &Box::new(13);
        x = y;
    }
    println!("{}", x);
}
```

Type safety

Gli errori si dividono in:

- **trapped**
- **untrapped**

Un programma **ben tipato** può essere:

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

Type casting

```c
#include <stdio.h>

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

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

Type initialization

```c
#include <stdio.h>

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

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

Immutability

Variabili

- Si dichiarano con `let`.
- Possono essere modificate.
- Lifetime generico.

Costanti

- Si dichiarano con `const`.
- Immutabili.
- Lifetime `'static`.

Grazie