

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 }
```

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.

e se ne può derivare:

- ▶ Data una struct il lifetime associato ad un suo riferimento è sottoinsieme del lifetime del membro della struct.

Sicurezza di un linguaggio

La **trusted computed base (TCB)** è la totalità delle componenti che vanno a definire la sicurezza di un sistema, è composta da:

- ▶ Access control.
- ▶ Information flow.

Questi controlli vengono eseguiti da Rust a tempo di compilazione.

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.

Sistema di tipi substrutturale

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

- ▶ **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 logiche substrutturali permettono di ragionare in termini di consumo di risorse, in un sistema di tipi lineare valgono due invarianti:

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

Memory Safety

Un programma si dice memory safe quando nessuna entità esce dal suo **address space**.

L'allocazione di memoria può avvenire in più modi:

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

Buffer overflow

```
1 #include <stdio.h>
2
3 int main() {
4     int arr[] = {13, 10};
5     printf("%d\n", arr[4]);
6 }
```

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

Integer overflow

- ▶ Modular arithmetic.

$$w = n \bmod m$$

- ▶ Saturation arithmetic.

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

Double free

```
1 #include <stdlib.h>
2
3 int main() {
4     char *ptr = malloc(sizeof(char));
5     *ptr = 'a';
6     free(ptr);
7     free(ptr);
8 }
```

```
1 fn main() {
2     let s1 = Box::new(13);
3     drop(s1);
4     drop(s1);
5 }
```

Dangling reference

```
1 fn main() {  
2     let x;  
3     {  
4         let y = &13;  
5         x = y;  
6     }  
7     println!("{}", x);  
8 }
```

```
1 fn main() {  
2     let x: &Box<i32>;  
3     {  
4         let y = &Box::new(13);  
5         x = y;  
6     }  
7     println!("{}", x);  
8 }
```

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

```
1 #include <stdio.h>
2
3 int main() {
4     int x = 12;
5     printf("%f\n", x * 0.5);
6 }
```

```
1 fn main() {
2     let x = 12;
3     // println!("{}", x * 0.5); // ERRORE!
4     println!("{}", x as f32 * 0.5);
5 }
```


Type initialization

```
1 #include <stdio.h>
2
3 int main() {
4     void* ptr = NULL;
5     printf("%p\n", ptr);
6     printf("%d\n", * (int *) ptr);
7 }
```

```
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

Variabili

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

Costanti

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

Grazie