Rust Safeness

Edoardo De Matteis

4 giugno 2020

Indice

1 Introduzione

Definiamo un'entità software *memory-safe* quando non accede ad indirizzi di memoria fuori dal proprio address space né esegue istruzioni fuori dall'area assegnatale da compilatore e linker [2], un linguaggio che garantisce la memory-safety dei suoi programmi è detto memory-safe.

1.1 Configurazione della memoria

A tempo di compilazione avviene l'allocazione statica della memoria sul *call stack* che cresce verso il basso, l'allocazione dinamica invece avviene tramite l'*heap*: una zona di memoria che cresce verso l'altro in cui è possibile allocare dati la cui dimensione non è nota a tempo di compilazione [3]. Questo approccio consolidato è adottato da vari sistemi operativi UNIX e UNIX-like oltre che da compilatori di linguaggi quali C, Rust, Java e C++; quali strutture dati vengano allocate in quale area della memoria dipende dal linguaggio, è prassi che sullo stack si allochino le chiamate a funzione, i tipi primitivi e puntatori mentre sull'heap compound types.

2 Violazione della memory-safety

Se in un linguaggio la gestione della memoria è lasciata al programmatore il linguaggio è generalmente unsafe (come C o C++), altri linguaggi quali Java o C# risolvono questo problema nascondendo i puntatori al programmatore ed eseguendo automaticamente la gestione della memoria, obbligando il linguaggio ad eseguire i dovuti controlli. In Rust è possibile fare riferimento esplicito alla memoria tramite puntatori ma bisogna distinguere tra *Safe Rust* e *Unsafe Rust*, nel secondo è infatti presente l'aritmetica dei puntatori del C purché sia in un blocco `unsafe` (nel quale si possono usare anche funzioni e librerie scritte in C) mentre in Safe Rust esistono i raw pointer ma non è possibile dereferenziarli.

2.1 Politica di sicurezza

È molto difficile definire cosa sia la sicurezza in Rust anche solo per il fatto che non c'è una definizione precisa

"Unsafe operations are those that can potentially violate the memory-safety guarantees of Rust's static semantics." [7, 14]
"There is no formal model of Rust's semantics for what is and is not allowed in unsafe code" [7, 14.3]

La definizione più consistente è che Safe Rust non causi *undefined behavior*, purtroppo non si ha una definizione formale di undefined behavior bensì è una lista in continuo aggiornamento [7, 14.3] ma è decisamente incompleta dato che non compaiono *integer overflow* e *double free* che invece sono considerati errori rispettivamente in [4, 3.2] e [4, 4.1].

2.1.1 Buffer Overflow

Avviene quando in un buffer si possono inserire dati di dimensione maggiore della sua capacità sovrascrivendo così altre informazioni [8]. Un esempio esplicativo ed interessante di buffer overflow in C è nel listato 1 [6, 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 è vero perché si confrontano solo i primi 8 caratteri. In Rust questa cosa non è possibile, si veda nel listato 2 come il linguaggio esegua dei *boundary-check*, 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 1: Stack 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 2: Buffer overflow in Rust

2.1.2 Stack Overflow

Quando si chiama una funzione si salva lo stack frame sul call stack e può capitare che chiamate ricorsive facciano entrare il programma in uno stato di non terminazione, Rust prima identifica staticamente la ricorsione infinita e poi segnala anche dinamicamente l'overflow. In C l'errore che si vede è dato dal sistema operativo (macOS Catalina 10.15.3) perché con questa ricorsione infinita si prova ad accedere ad un segmento di memoria cui il programma non può.

```
void f() {f();}

int main() {
  f();
}
```

```
$ ./bin/segfault_c
[1]    22217 segmentation fault  ./bin/segfault_c
```

Listing 3: Stack overflow ricorsivo in C

```
fn main() { f();}

fn f() {
    f();
}
```

```
$ rustc segfault.rs -o bin/segfault_rs
warning: function cannot return without recursing
 --> segfault.rs:3:1
  |
3 | fn f() {
  | ^^^^^^ cannot return without recursing
4 |     f();
  |     --- recursive call site
  |
  = note: `#[warn(unconditional_recursion)]` on by default
  = help: a `loop` may express intention better if this is on
    purpose

$ ./bin/segfault_rs

thread 'main' has overflowed its stack
fatal runtime error: stack overflow
```

Listing 4: Stack overflow ricorsivo in Rust

2.1.3 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**. 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 in quanto esplicitamente richiesto dai progettisti per motivi di efficienza. Similmente in Rust compilando in *debug* mode si eseguono dei controlli dinamici per l'overflow 5, contrariamente al C che applica direttamente wrapping.
Differenti linguaggi adottano differenti approcci per limitare integer overflow, se si può prevedere il valore massimo possibile in un programma allora è in generale facile evitare overflow; in caso contrario si possono utilizzare metodi dinamici che sfruttano eccezioni 5.

```
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 5: Integer overflow in Rust

Nel listato 6 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.

```
#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 6: Integer overflow in C

2.1.4 Double-Free

Si ha un *double free error* [3, 10.4.4] quando si prova a liberare più volte la stessa zona di memoria, in Rust non è possibile grazie all'*ownership* che impedisce ad un'area di memoria di esser puntata da più di un puntatore, nel codice in 7 si può vedere come `s1` perda validità dopo aver trasferito a `s2`l'ownership della stringa cui puntava.

```
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 7: Ownership

In C invece si può osservare il codice nel listato 8: 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 8: Ownership

Il puntatore `Guido` è anche un *dangling pointer*.

2.1.5 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 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 9 l'indirizzo viene stampato a schermo e inserito dall'utente (indicato da >) per semplicità, lo si può rendere in modo più esplicito mantenendo una variabile per il puntatore, sia essa `mess` stessa o una ad hoc passata come argomento alla funzione.

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

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

void send_message(char *, char *, char *);
```

```
12
13 int main() {
14   send_message("Guido", "Carla", "asa nisi masa");
15
16   char * ptr;
17   printf("> ");
18   scanf("%p", &ptr);
19
20   printf("[%s, %s, %s]\n", ptr, ptr+8, ptr+16);
21 }
22
23 void send_message(char * from, char * to, char * text) {
24   struct Message * mess = malloc(sizeof(struct Message));
25   strcpy(mess->from, from);
26   strcpy(mess->to, to);
27   strcpy(mess->text, text);
28   printf("%p\n", mess);
29 }
```

```
1 $ ./bin/dangref_c
2 0x7fec9fe04d10
3 > 0x7fec9fe04d10
4 [Guido, Carla, asa nisi masa]
```

Listing 9: 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; esistonono però le reference [4, 4.2] che permettono di prendere in prestito (*borrowing*) una variabile senza esserne proprietari, nel codice in 10 `s` nella riga 5 è quindi una *shallow copy* di `s2`. Come nell'*ownership* però anche il *borrowing* perde validità oltre lo scope.

```
1 fn main() {
2     let s1: &String;
3     {
4         let s2 = String::from("Hello");
5         s1 = &s2;
6     }
7     println!("{}", s1);
8 }
```

```
1 $ rustc dangref.rs -o bin/dangref_rs
2 error[E0597]: ‘s2‘ does not live long enough
3  --> dangref.rs:5:14
4   |
5 |          s1 = &s2;
6   |               ^^^ borrowed value does not live long enough
7 |      }
8   |      - ‘s2‘ dropped here while still borrowed
9 |      println!("{}", s1);
10  |                     -- borrow later used here
11
12 error: aborting due to previous error
13
```

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

Listing 10: Dangling reference in Rust

2.2 Type Safety

Durante l'esecuzione di un programma si può incorrere in due tipi di errori: *trapped* e *untrapped*, i primi sono facilmente riconoscibili dato che provocano il fallimento dell'esecuzione di un programma, i secondi sono molto più difficili da individuare perché non interrompono il programma. Un linguaggio in cui nessun programma legale genera errori trapped né untrapped si dice *sound*, solo untrapped invece si dice *safe* (un linguaggio sound è necessariamente safe) [1]. Una definizione simile è quella di Milner per cui *well-type programs cannot "go wrong"* [5], con programmi che si "comportano male" si possono intendere programmi non sound o che generano undefined behaviour (sezione 2.1).

2.2.1 Type Confusion

Nel listato 11 si alloca un array di short (2 B) staticamente, andrà quindi sullo stack e considerando che il computer su cui ho eseguito il programma monta un processore Intel - little endian- 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 $2^{16} = 65536$.

```c
#include <stdio.h>

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

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

Listing 11: Type confusion in C

In Rust non si può proprio puntare ad una variabile con un tipo differente da quello della variabile stessa.

```rust
fn main() {
    let arr: [i16; 2] = [0, 1];
    let p = &arr as *const i32;
}
```

```
$ rustc lilliput.rs -o bin/lilliput_rs
error[E0308]: mismatched types
 --> lilliput.rs:3:13
  |
3 |     let p = &arr as *const i32;
  |             ^^^^^^^^^^^^^^^^^^ expected i16, found i32

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
```

Listing 12: Type confusion in Rust

3 Note al professore

Nella sezione 2.2.1 non sono riuscito a trovare esempi significativi di type confusion in C proprio grazie alla sua semplicità e assenza di classi quindi sembra più un'opacità semantica come ha detto lei.

Bibliografia

[1] Pietro Cenciarelli. Dispense del corso linguaggi di programmazione, 2019.

[2] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, 2003.

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

[4] Steve Klabnik and Carol Nichols. The rust programming language.

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

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

[7] The rust reference.

[8] Jansen Wayne, Theodore Winograd, and Karen Scarfone. Guidelines on active content and mobile code. Technical Report 800-28, NIST, 3 2008.