Rust Safeness

Edoardo De Matteis

19 marzo 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 [1], 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 [2]. 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."[5, 14]
"There is no formal model of Rust's semantics for what is and is not allowed in unsafe code"[5, 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 [5, 14.3] ma è decisamente incompleta dato che non compaiono *integer overflow* e *double free* che invece sono considerati errori rispettivamente in [3, 3.2] e [3, 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 [6]. Un esempio esplicativo ed interessante di buffer overflow in C è il seguente [4, 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

Dal momento che la memoria ha una dimensione limitata le operazioni vengono eseguite in aritmetica modulare e sommando oltre il limite superiore concesso dall'architettura si ha un overflow. Spesso ha causato problemi molto seri e il C non lo gestisce, lo stesso fa Rust se si compila in `release` mentre in `debug` interrompe l'esecuzione, il programma qui sotto ci rivela comunque l'architettura del sistema perché si stampa ad ogni iterazione ma non si può dire sia una violazione di sicurezza perché non potrebbe fare altrimenti, inoltre l'integer overflow diventa davvero interessante quando combinato ad altre vulnerabilità (i.e. buffer overflow) [2, 10.2.3].

```c
#include <stdio.h>

int main() {
  int n = 1;
  int i = 1;

  while(n > 0) {
    n *= 2;
    i++;
  }
  printf("%d bit architecture.\n", i);
}
```

```
$ ./bin/intof_c
32 bit architecture.
```

Listing 5: Integer overflow in C

```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 6: Integer overflow in Rust

2.1.4 Double-Free

Si ha un *double free error* [2, 10.4.4] quando si hanno due puntatori alla stessa locazione e si tenta di liberare entrambi. È una vulnerabilità perché la memoria è divisa in blocchi posizionati in double-linked list chiamate *bin*, quando un blocco viene liberato lo si unisce con i suoi vicini liberi in un unico bin; a questo punto la vulnerabilità si presenta se non si pone il puntatore appena liberato a `null` e si potrà accedere ad una zona di memoria addirittura maggiore di quella liberata in precedenza fino a che non si deciderà di liberare il secondo puntatore. In Rust questo non è possibile grazie all'*ownership* infatti una zona di memoria può avere al massimo un puntatore, se si esegue un assegnamento ad un nuovo puntatore il primo diventa invalido. A onor del vero in C tutto questo è possibile anche senza double free perché si può accedere a qualsiasi locazione, se un linguaggio lo vietasse però dovrebbe tener conto di questa vulnerabilità.

```
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

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. Nel listato 8 la variabile `p` riesce a puntare a `n` anche dopo che questa esce fuori dal suo scope, si può tentare di ricreare lo stesso comportamento in Rust con le reference [3, 4.2] ma viene intercettato dal compilatore.

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

int main() {
  long p;
  {
```

```
    int n = 13;
    p = (long) &n;
  }
  printf("%d\n", *(int *) p);
}
```

```
$ ./bin/scope_c
13
```

Listing 8: Dangling pointer in C

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

```
$ rustc dangref.rs -o bin/scope_rs
error[E0597]: `s2` does not live long enough
 --> dangref.rs:5:13
  |
5 |          s = & s2;
  |              ^^^^ borrowed value does not live long enough
6 |      }
  |      - `s2` dropped here while still borrowed
7 |      println!("{}", s);
  |                     - borrow later used here

error: aborting due to previous error

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

Listing 9: Dangling reference in Rust

2.2 Type Safety

La *type-safety* è spesso associata alla memory-safety perché si pensa alle violazioni in memoria sfruttando il sistema dei tipi e se ne ha un esempio nella sezione 2.2.1. Una definizione che distingue la type-safety dalla memory-safety si concentra sul fatto che nella semantica operazionale di un linguaggio type-safe ogni operazione se viene valutata allora termina [7].

2.2.1 Type Confusion

Nel listato 10 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 10: 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 11: Type confusion in Rust

3 Note al Professore

1. Nella sezione 2.1.4 non ho messo il codice in C perché non faceva nulla di interessante e il kernel interviene.

Bibliografia

[1] 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.

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

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

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

[5] The rust reference.

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

[7] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 11 1994.