Preventing RUST Programs from becoming C Programs

Redhat Research Day Presentation

Vikram Nitin, Baishakhi Ray

Columbia University

May 19, 2022

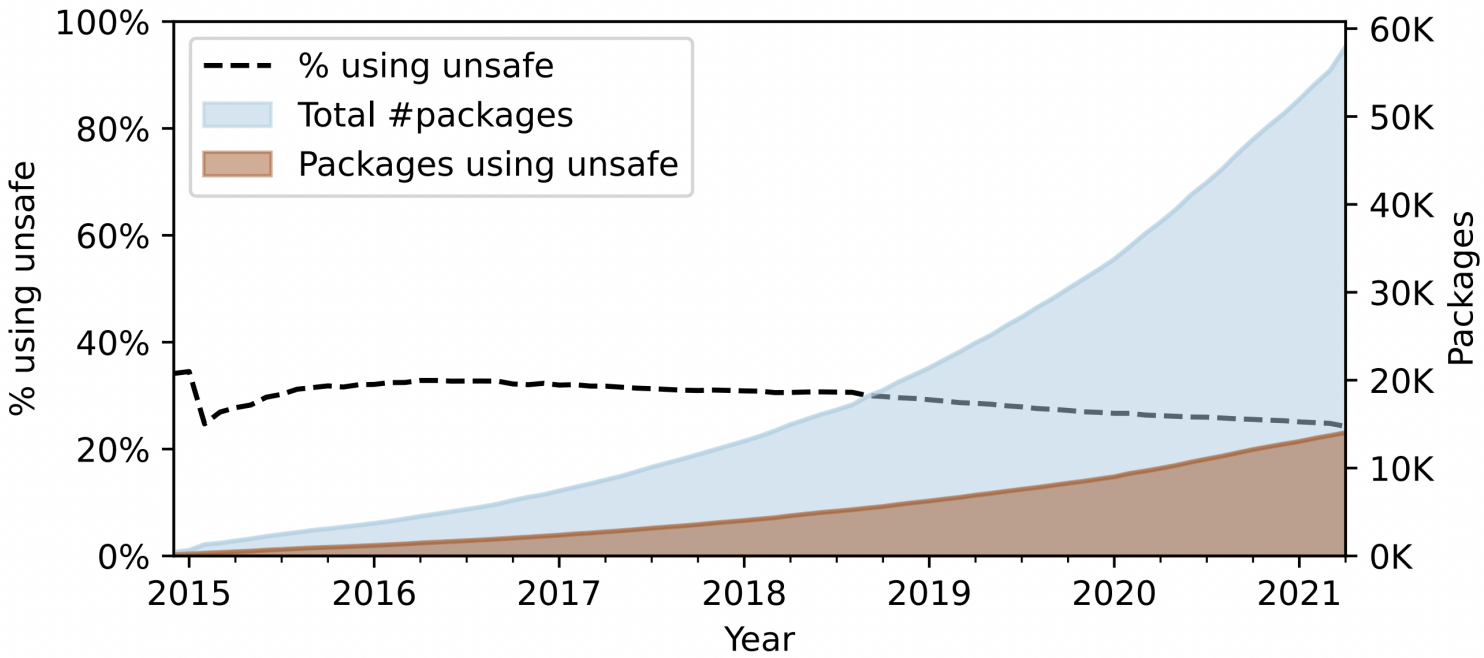
Outline

1. Vulnerabilities in RUST
2. Features of RUST
3. Lifetime Annotation Bugs
4. Algorithm to Detect Lifetime Bugs

Vulnerabilities in RUST

Use of `unsafe`

The percentage of packages using `unsafe` has not decreased significantly with time. [1]



[1] Figure taken from `https://taesoo.kim/pubs/2021/bae:rudra.pdf`

Vulnerabilities in RUST

RUSTSEC Security Advisory Database [2]

Tracks security advisories filed against RUST packages (or "crates").

Number of advisories has been steadily growing :

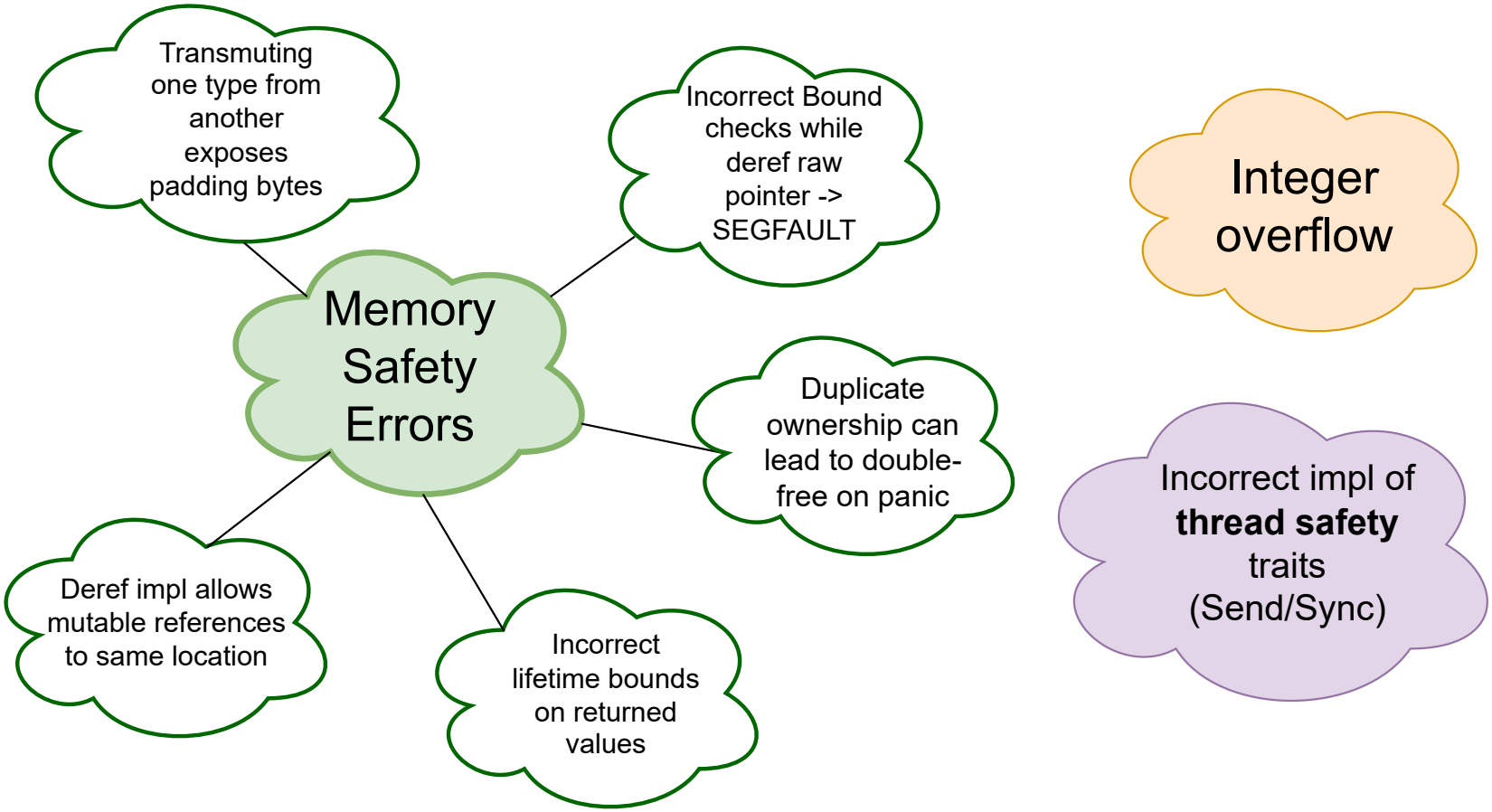
- 2015: 1
- 2016: 6
- 2017: 8
- 2018: 26
- 2019: 40
- 2020: 165
- 2021: 141

[2] https://rustsec.org/advisories/

Vulnerabilities in RUST

Types of Vulnerabilities

We did an empirical study of 50 RUSTSEC advisories. Here are some of the types of vulnerabilities we found :



Vulnerabilities in RUST

With all these vulnerabilities, a natural question is :

Why use RUST?

The "pure" RUST language, without unsafe, provides very strong memory safety guarantees.

Outline

1. Vulnerabilities in RUST
2. Features of RUST
3. Lifetime Annotation Bugs
4. Algorithm to Detect Lifetime Bugs

Features of RUST

Ownership

Ownership is arguably the most central and unique feature of RUST.

- Each allocated heap value is "owned" by a variable.
- Only one owner at a time!
- Once the variable goes out of scope, the heap memory is freed, or "dropped".

Features of RUST

Ownership

- Heap memory is "owned" by the variable that points to it
- "Dropped" when owning variable goes out of scope.

```
{
    let s = String::from("Hello"); //Heap
} // Memory dropped here
```

```
let s1 = String::from("Hello");
let s2 = s1;
println!("{}", s1); //Does not compile
```

```
let x = 5; //Stack
let y = x;
println!("{}", x); // Fine
```

Features of RUST

Ownership and Functions

```
let s = String::from("Hello");
foo(s);
println!("{}", s); // No

fn foo(s: String) {
    // do something
}
```

```
let x = 5;
foo(x);
println!("{}", x); // Yes

fn foo(x : i32) {
// do something
}
```

Features of RUST

Ownership and Functions (cont.)

```
let s = String::from("Hello");
foo(&s);
println!("{}", s); // Yes

fn foo(s: &String) {
// do something
}
```

```
let s = String::from("Hello");
foo(&s);
println!("{}", s); // Yes

fn foo(s: &String) {
s.push_str(" world"); //No
}
```

Features of RUST

Mutable References

```
let s = String::from("Hello");
foo(&mut s);
println!("{}", s); // Yes

fn foo(s: &mut String) {
    s.push_str(" world");//Yes
}
```

Only one at a time!

```
let s = String::from("Hello");
let r1 = &mut s;
let r2 = &mut s; // No
let r3 = &s; // No
```

Features of RUST

Great! So what's the catch?

Features of RUST

We may need a finer level of control sometimes. RUST compiler might reject valid programs! Example :

```
let slice : [i32; 6] = [1, 2, 3, 4, 5, 6];
let first_half = &mut slice[..3];
let second_half = &mut slice[3..];
// Do something with the sub-arrays
```

Error - cannot have two mutable references to ‘slice‘

Solution: “Raw pointers”

Features of RUST

Raw pointers

Raw pointers don't have the same guarantees that normal references do.

- No automatic cleanup/drop
- They aren't guaranteed to point to valid memory
- Can point to "null"
- You can have two mutable raw pointers to the same location

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Features of RUST

Unsafe RUST

- RUST with some "Superpowers"
- Allows you to dereference raw pointers! **gasp**

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("{}{}", *r1, *r2);
}
```

Features of RUST

Unsafe RUST (cont.)

Unsafe code can be used in a function in two ways :

```
fn foo() { // ``Encapsulating'' the unsafety
    let mut num = 5;
    let r1 = &num as *const i32;
    unsafe { println!("{}", *r1); }
}
```

```
unsafe fn foo() { // ``Exposing'' the unsafety
    let mut num = 5;
    let r1 = &num as *const i32;
    println!("{}", *r1);
}
```

Features of RUST

Unsafe RUST (cont.)

If a function has encapsulated unsafe code, then it should behave like a safe function to its callers!

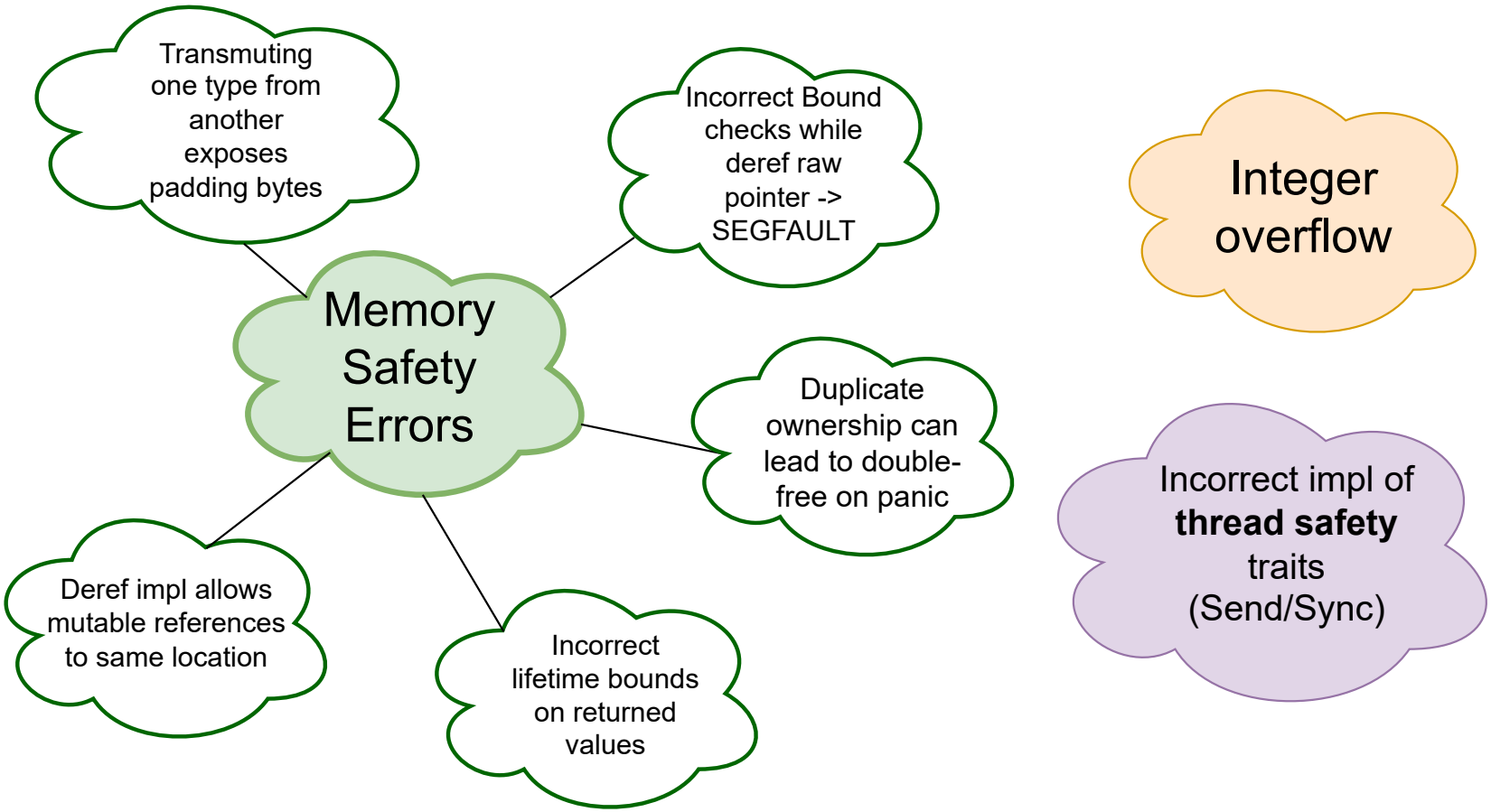
Features of RUST

Unsafe RUST (cont.)

If a function doesn't encapsulate its unsafety correctly, then it transitively compromises the safety of every function that relies on it!

Vulnerabilities in RUST

Types of Vulnerabilities



Outline

1. Vulnerabilities in RUST
2. Features of RUST
3. Lifetime Annotation Bugs
4. Algorithm to Detect Lifetime Bugs

Understanding Lifetimes

Definition

Lifetime - a construct that the RUST borrow-checker uses to ensure that all borrows are valid. Similar to scope.

Fundamental rule : A borrow is valid if the lifetime of the borrow ends before the lifetime of the lender. That is, the variable being pointed to must outlive the pointer.

Understanding Lifetimes

Example

In the example below, the lifetime of `x` ends before the lifetime of `r`. So this will not compile.

```
{
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("{}", r);  // Invalid
}
```

Understanding Lifetimes

Lifetime Annotation

- Each identifier can be annotated with a lifetime parameter, denoted with an apostrophe. Eg - `'a`, `'b`, `'foo`.
- Each lifetime parameter corresponds to a concrete scope or statically defined region in the program.
- The variable can live for at most as long as the scope corresponding to the lifetime parameter.

Understanding Lifetimes

Lifetime Annotation (cont.)

A variable with `'a` can live for at most as long as `'a`.

Conversely...

If a *reference* has lifetime `'a`, then the variable it points to must live at least as long as `'a`.

Understanding Lifetimes

Lifetime Annotation (cont.)

Example of lifetime annotations :

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

What does this mean?

Understanding Lifetimes

Types of Bugs

Raw pointers don't have lifetime annotations! Rust doesn't track lifetimes of the objects that they point to.

We define 3 types of bugs involving raw pointers.

Lifetime Annotation Bugs

Type 1

```
struct Record {
    age: *const u32, ... // Raw pointer
}
impl Record {
    fn get_age<'a,'b>(&'a mut self) -> &'b mut u32 {
        unsafe{&mut *(self.age)}
    }
}
```

Problem - the returned reference now points to `self.age` (dataflow). So the lifetime `'b` should be at least as long as the lifetime `'a`.

Real-world example : `https://rustsec.org/advisories/RUSTSEC-2020-0023.html`

Lifetime Annotation Bugs

Type 2

```
struct Record {
    age: *const u32, ... // Raw pointer
}
impl Record {
    fn set_age<'a>(&'a mut self, age: &'a u32) {
        self.age = age;
    }
}
```

Problem - `self.age` now points to `age` (dataflow). So the lifetime `'a` should be at least as long as the lifetime of the struct object `self`.

Real-world example : `https://rustsec.org/advisories/RUSTSEC-2021-0128.html`

Lifetime Annotation Bugs

Type 3

```
struct Record<'a> {
    age: *const u32, ... // Raw pointer
}
fn create_record<'a, 'b>(x : &'b u32) -> Record<'a>{
    Record{age: x, ...}
}
```

Problem - `Record` is borrowing `x` (dataflow). So the lifetime `'b` should be at least as long as the lifetime `'a`.

Real-world example : `https://rustsec.org/advisories/RUSTSEC-2021-0130.html`

Outline

1. Vulnerabilities in RUST
2. Features of RUST
3. Lifetime Annotation Bugs
4. Algorithm to Detect Lifetime Bugs

Algorithm to Detect Lifetime Bugs

We implement our analysis by building on the `Rudra` framework [3].

The outline is as follows :

- Step 1: Find all structures with raw pointers
- Step 2: Find all functions where references to these structures are arguments, or returned.
- Step 3: Look for dataflows to/from the raw pointers inside.
- Step 4: Check for Type 1, Type 2, Type 3 lifetime bugs.

It is possible to have *false posititives* with this approach, but that is acceptable.

[3] Bae, Yechan, et al. "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale."

Toy Program

Has Type 1, Type 2 and Type 3 bugs

```
pub struct Record<'a> {
        age: *mut i32, ...
}
impl<'a> Record<'a>{
        fn set_age(&mut self, val: &mut i32) {
                self.age = val;
        }
        fn get_age<'b>(&mut self) -> &'b mut i32 {
                unsafe {&mut (*self.age)}
        }
}
fn create<'a>(age: &'_ mut i32) -> Record<'a> {
        return Record {age : age, ...}
}
```

Algorithm to Detect Lifetime Bugs

Output

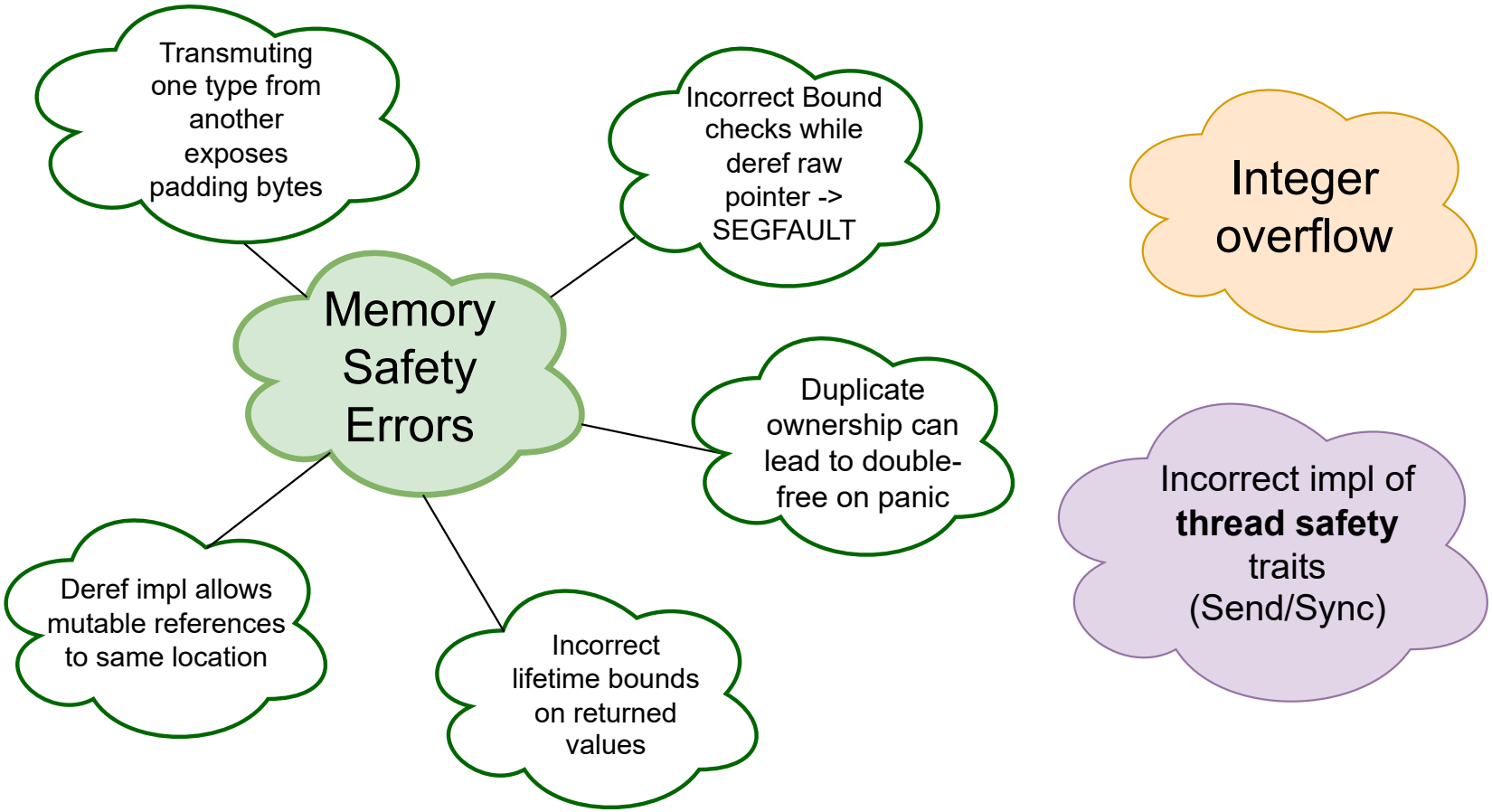
```
Potential memory safety error!
Func "set_age" takes a reference to self, and the
lifetime of the structure object containing a raw
pointer is Some(Param(Plain('a#0))). But it also
takes a reference with a different lifetime Implicit.

Potential memory safety error!
Func "get_age" takes a reference to self with
lifetime Implicit. But it returns a reference with a
different lifetime Some(Param(Plain('a#0))).

Potential memory safety error!
Func "create" returns a struct containing a raw
pointer with lifetime Some(Param(Plain('a#0))).
But it takes one or more input references, none
of which has the same lifetime.
```

Future steps

This is a work in progress. Dataflow analysis isn't implemented yet :

- Step 1: Find all structures with raw pointers
- Step 2: Find all functions where references to these structures are arguments, or returned.
- ~~Step 3: Look for dataflows to/from the raw pointers inside.~~
- Step 4: Check for Type 1, Type 2, Type 3 lifetime bugs.

Future Steps



Thank You!

Questions?