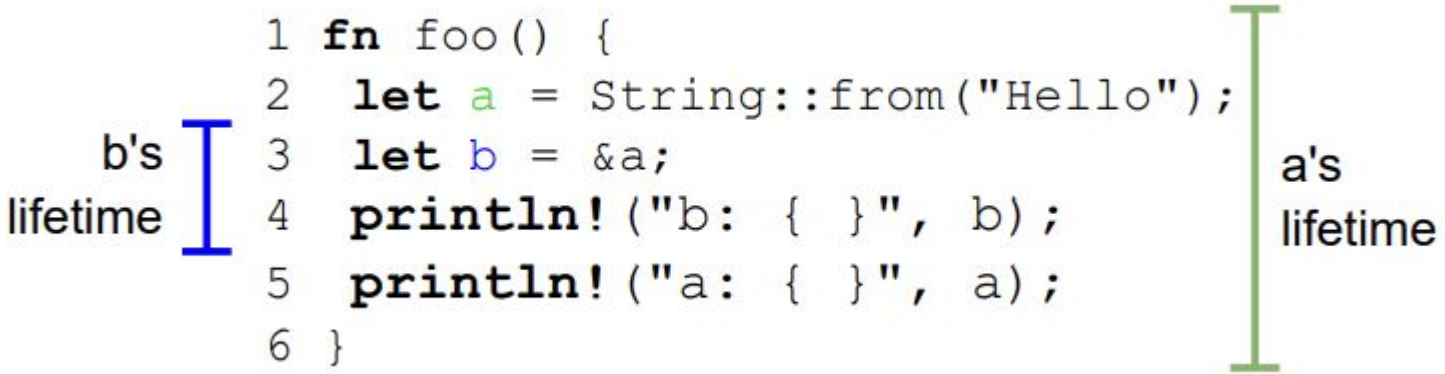
YUGA: Automatically Detecting Lifetime Annotation Bugs in the Rust Language

February 10, 2025

Muhammad Hassnain

Lifetime

Scope of a variable during which it is valid.



Lifetime Annotation

Syntactic representation of a concrete lifetime.

```
fn foo<'a,'b>(x: &'a String, y: &'b String) -> &'b String { y }
```

Here 'a and 'b are Lifetime annotations.

Some Rust Rules

- There can be either one one mutable or any number of immutable references at any point not both.
- The lifetime of the borrow must not extend beyond the lifetime of the borrows value.

Some Rust Rules (2)

Lifetimes in Rust ensure that references are valid as long as they're used. For example, the lifetime of a returned reference must match the lifetime of the input reference it borrows from.

```
fn longest(x: &str, y: &str) -> &str {
}

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

Some Rust Rules (3)

Raw pointers themselves don't have LAs, they can be contained within structs, and these structs can have associated LAs

```
struct Foo<'a> {
    x: *const String,
    ..
}
fn bar<'a,'b>(arg1: &'a String, arg2: &'b String)
        -> Foo<'a> {
    Foo{x: arg2 as *const String, ..}
}
```

Characterizing LA Bugs

These lifetime annotation bugs lead to:

- **Use-after-free:** attempt to access memory that has been freed.
- **Non exclusive Mutability:** two mutable borrows co-occur within same lifetime.

Characterizing LA Bugs

All lifetime annotation bugs fall into one of two categories:

- **Mismatched arg-arg**: Within a function, some values may be transferred between two arguments.
- **Mismatched arg-return**: Within a function, some values may be transferred between an argument and returned value.

Use-After-Free Bugs

-> self.head.next is a raw pointer with associated LA '_.

-> Returned Iter object has associated LA 'a.

(https://github.com/jeromefroe/lru-rs/issues/120)

Example 1. RUSTSEC-2021-0130: A **use-after-free** bug in the `lru` crate caused by assigning a value from an argument to return (**arg-return**)

```
1  pub struct LruEntry<K, V> {
2      next: *mut LruEntry<K, V>, ..
3  }
4  pub struct LruCache<K, V> {
5      head: *mut LruEntry<K, V>, ..
6  }
7  pub struct Iter<'a, K: 'a, V: 'a> {
8      ptr: *const LruEntry<K, V>, ..
9  }
10 impl<K, V, S> LruCache<K, V, S> {
11     ...
12   pub fn iter<'a>(&'_ self) -> Iter<'a, K, V> {
13         Iter {
14             ptr: unsafe { (*self.head).next },
15             ..
16         }
17     }
18 }
```

Use-After-Free Bugs

For every call of the update_hook function, hook must live at least as long as self.

Both hook and self have the same LA 'c, which means that they both live at least as long as 'c. But this does not mean that hook lives as long as self.

(https://github.com/rusqlite/rusqlite/issues/1048)

Example 2. RUSTSEC-2021-0128: A **use-after-free** bug in the `rusqlite` crate caused by assigning a value from one argument to another argument (**arg-arg**).

```
1  pub struct InnerConnection {
2    db: *mut ffi::sqlite3,
3    ..
4  }
5  pub struct Connection {
6    db: RefCell<InnerConnection>
7  }
8  impl Connection {
9      ..
10   pub fn update_hook<'c,F>(&'c self,hook:Option<F>)
11     where
12     F: .. + 'c
13   {
14     self.db.borrow_mut().update_hook(hook);
15   }
16 }
```

Non-Exclusive Mutability Bugs

On Line 9, raw_slice_mut, takes a mutable borrow to self.

raw_slice_mut gives mut access to some memory self.row.ptr.

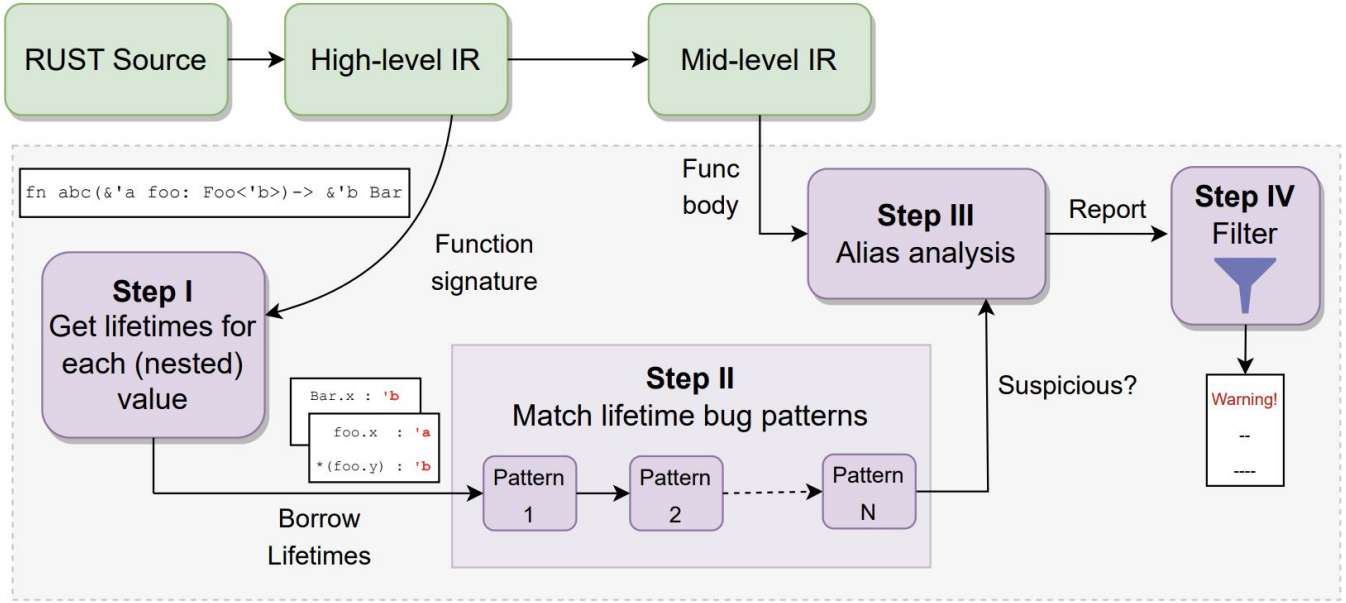
The function takes a lifetime '_ and returns a lifetime 'a.

It is possible for 'a to be longer than '_, so raw_slice_mut can be called multiple times to create multiple mutable borrows to the same *(self.row.ptr) value.

Example 3. RUSTSEC-2020-0023: A **non-exclusive mutability** bug in the `rulinalg` crate caused by assigning a value from an argument to return (**arg-return**)

```
1  pub struct MatrixSliceMut<'a, T: 'a> {
2      ptr: *mut T, ..
3  }
4  pub struct RowMut<'a, T: 'a> {
5      row: MatrixSliceMut<'a, T>
6  }
7  impl<'a, T: 'a> RowMut<'a, T> {
8    /// Returns the row as a slice.
9    pub fn raw_slice_mut(&'_ mut self)->&'a mut [T] {
10     unsafe { from_raw_parts_mut(self.row.ptr, ..) }
11   }
12 }
```

Yuga


RUST Source
High-level IR
Mid-level IR
fn abc(&'a foo: Foo<'b>)-> &'b Bar
Function signature
Func body
Step I
Get lifetimes for each (nested) value
Step II
Match lifetime bug patterns
Bar.x : 'b
foo.x : 'a
*(foo.y) : 'b
Pattern 1
Pattern 2
Pattern N
Borrow Lifetimes
Step III
Alias analysis
Report
Step IV
Filter
Suspicious?
Warning!
--

Borrow Lifetimes Rules

Treat a raw pointer as though it is “owned” by the structure that it is contained in.

- If a value is borrowed with LA’ b, then the value and all its fields must outlive ’b.
- If a value is pointed by a raw pointer and the raw pointer is contained inside a structure with LA’a, then ’a is the lifetime of the structure, and the value must outlive ’a.
- If a value is pointed by a raw pointer and the raw pointer is contained inside a structure with no LA, then this value must outlive any lifetime that the structure object must outlive.

Alias Analysis

Checking whether two pointers point to the same memory location. Their algorithm has following properties:

- **Intra-procedural:** Analysis on each function body individually, treating function calls as opaque.
- **Field-sensitive:** Treating each field of the struct separately.
- **Flow-sensitive:** Consider the program statements in order and compute ‘points-to’ sets at each program point.

Points to

They build up on this Anderson's Analysis (1994) but there algorithm is flow sensitive.

- a = b then we merge b's points-to set into a's.

Filter : Reducing False Positives

They focus on cases where:

- Raw pointer is assigned in both source and target structs and they have a custom drop implementation.
- There are three specific functions from trait implementations that they filter out - Iterator::next, Iterator::next_back, Clone::clone (they rely on semantic correctness to ensure safety?)

Implementation

- HIR -> Tree representation of program similar to AST.
- MIR -> Represents each function as CFG of basic-blocks.
- They use both and build on top of Rudra.

Evaluation

TABLE 2: Running YUGA on Rustsec vulnerabilities due to LA bugs

	RustSec ID	Error Type	#Vulnerable Funcs	#Detected	False Positives	False Negatives
1	2018-0020	Use-after-free	1 + 2*	2	0	1
2	2020-0023	Mutability	2	2	0	0
3	2020-0060	Use-after-free	1	0	0	1
4	2021-0128	Use-after-free	7	7	2	2
5	2021-0130	Use-after-free	2 + 3*	5	0	0
6	2022-0028	Use-after-free	2	0	0	2
7	2022-0040	Use-after-free	1	0	0	1
8	2022-0070	Use-after-free	1	0	0	1
9	2022-0078	Use-after-free	1	0	0	1
	Total		**23**	**16**	**2**	**9**

The asterisk * denotes newly exposed bugs that weren't part of the initial report.

Evaluation

TABLE 4: Results of running YUGA on real-world Rust crates

Category	Sub-category	#
Exploitable bugs		**3**
Code smells	Freed ptr but no deref	14
	User-implemented ref counting	30
	Memory copy, not alias	21
	Different field of struct	16
	Two lifetime annotations	1
Total		**85**

Evaluation

Tool	# Bugs	Tool detected	YUGA detected
Rudra	50	0	30
MirChecker	27	0	16
Miri (w/o exploit code)	27	0	16
Miri (w. exploit code)	27	27	16

TABLE 6: Comparison with existing tools

23 from Rust Sec and 27 Curated - why only Rudra evaluated on RustSec