Bugs You'll Probably Only Have in Rust

Alexis Beingessner - June 14, 2017

> The source of this post is hosted on Github (https://github.com/Gankro/gankro.github.io/blob/master/blah/only-in-rust/index.md). If you find any errors, please file an issue or submit a PR! 😀

Recently, Ralf Jung found a bug in Rust's standard library (https://www.ralfj.de/blog/2017/06/09/mutexguard-sync.html). Congrats to him! 🎉

The bug was a missing annotation, and the result was that users of Rust's stdlib could compile some incorrect programs that violated memory safety. 🙀 Fortunately, those incorrect programs would be pretty surprising to see in the wild, so no one should be affected. Adding the annotation should fix the bug without anyone noticing.

On twitter an interesting question was raised: is this the first Rust-specific bug class?

Nope! There's actually a few of them, which is why I did my best to document them in The Rustonomicon (https://doc.rust-lang.org/nomicon/) (which I've started working on again, and hope to improve and finish). But the Rust-specific bugs documented in the nomicon are mixed in with lots of details and bugs that happen in other languages, so I thought it would be interesting to put all the Rust-specific ones together in one place.

What These Bugs Aren't

So just to be clear, there's lots of bugs that Rust code can have. Many of them you would expect to see in any language: integer overflow, index out of bounds, failing to sanitize input, etc.

Others are *kinda* Rust-specific, in that they're a consequence of certain features: exception safety (https://doc.rust-lang.org/nomicon/exception-safety.html), dangling raw pointers, inappropriately trusting interfaces (https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html#how-safe-and-unsafe-interact), etc. However we won't be focusing on these issues because they're just as relevant in Rust's close relatives: C++ and Swift. (Swift doesn't really have exception safety issues; and C++ doesn't really have trust issues, insofar as C++ doesn't really try to distinguish between "safe" and "unsafe" operations)

But the bugs I want to focus on I've only seen in Rust code, as a result of unique type-system interactions. I won't go so far as to say that no other language has these issues. I just don't know them, and I wouldn't expect to see them.

Also I should note that all the bugs I want to focus on will be related to writing unsafe code. So these aren't things you generally need to worry about day-to-day when using Rust. The reason they're related to unsafe code is because these bugs are generally places where we're doing clever/subtle things for you which are always correct *in safe code* but fall over once you start doing the things `unsafe` lets you.

Ralf's Bug: Oops Unsafe Autotraits

TL;DR: if you write `unsafe` you *must* closely audit `Send` and `Sync` for your types!

This one's going to need a fair bit of background, because it's the consequence of a several obscure Rust features.

There's a trait (interface) in Rust called Sync. It's a marker trait, which just means that the interface doesn't actually have any methods or types you need to provide to implement. It's basically a type-level boolean: either you implement it or you don't. Implementing Sync is a declaration that your type has a property: it's safe for two threads to access it concurrently. Generic APIs which want to share data between threads do that by requiring `T: Sync` so that they're safe.

Almost every type in Rust can implement Sync, because sharing in Rust means read-only access. There's no worries about two threads reading from an integer at the same time, so integers are Sync. Similarly a type made of integers is safe, and so on. The interesting Sync types come from concurrency primitives which actually let you mutate in shared contexts: Mutex, RwLock, AtomicUsize, etc. Very few types aren't Sync. The most notable one is probably Rc, which is a non-atomically reference counted pointer, because it lets you increment the count in shared contexts without proper synchronization.

There's two quirks about Sync that make it different from almost any other trait in Rust: it's unsafe, and it's an autotrait.

Being unsafe just means that when you implement Sync, you have to use the `unsafe` keyword. The benefit of this is purely a matter of rules lawyering: it must be impossible to violate memory safety with Rust and its standard library -- unless you wrote `unsafe`. We make Sync unsafe because, unlike most other traits, there is no "defense" that can be taken against a bad Sync implementation. If we get it wrong then we're letting our users write Data Races, which violate memory safety!

But stopping there would have been an unacceptable state of affairs. Making `unsafe` a big scary "all bets are off" button is only compelling if most of our users *don't need to use that button*. Rust is trying to be a language for writing concurrent applications, so sharing your type between threads requiring `unsafe` would be really bad.

This is solved by Sync being an autotrait. An autotrait, which is a feature that is currently unstable and basically only used by the standard library, is a trait that's structurally derived for all types in existence. So if your type contains only Sync types, you're Sync without you saying a word about it. So everyone gets to share their types across threads without writing `unsafe`!

For *almost* every type in existence, this works perfectly: if you're made up of things that can be safely shared between threads, then you can be safely shared between threads. If you contain anything that can't be, then you can't be.

For the few types where this system gets the wrong answer, Autotraits can be manually implemented or unimplemented. It is in these manual implementations that you must write `unsafe`, and accept the burden of memory safety. As a safety mechanism, several "fishy" types explicitly don't implement Sync, so that any type containing them has to explicitly declare that it's ok. The most notable example of this `*mut`, the raw pointer type.

And here we get to Ralf's Bug: a type (MutexGuard) autoderived Sync, when it shouldn't have. That's it. We should have been a bit more vigilant and noticed this, but we didn't. Dang. 🙈

But wait, how could this have happened? I just argued that it should be impossible to violate memory safety without writing `unsafe`, but our bug just violated memory safety, and it was because we *didn't* write anything!

The answer is that, in fact, we did write `unsafe` -- in the implementation of MutexGuard (https://github.com/rust-lang/rust/blob/995f741a0e3a57d4142c0590b3266514fa0a0e29/src/libstd/sync/mutex.rs#L407-L414). (score one for rules lawyers!) The only reason MutexGuard being Sync is problematic is because it has imbued itself with special semantics using unsafe code -- the ability to reach into a Mutex and access its data without any synchronization.

/

Usually this isn't a problem because our "fishy type" protection catches it. But in this case MutexGuard just holds a perfectly normal reference to a Mutex -- and Mutexes are Sync, so that should be fine!

So there's our first bug that You Should Only See In Rust: writing unsafe code, and failing to audit the stdlib's two unsafe autotraits for your types: Send and Sync.

And here's the fix (https://github.com/rust-lang/rust/pull/41624): properly overriding the Sync impl for MutexGuard... right next to the line where we had already done that for Send two years ago. 😭

Unbound Lifetimes

TL;DR: don't infer lifetimes that unsafe pointers produce!

So Rust has these lifetime things. They're type-level variables for talking about scopes/regions in your program. We attach them to references to indicate what scope of the program we can guarantee the reference will be valid for. Mostly, lifetimes only show up to create a relationship between two references. For example:

```
fn foo<'a>(input: &'a u32) -> &'a u32;
```

Here we're saying "I take in any reference to u32, and returns a reference to a u32 that lasts at least as long". Because this is an incredibly common idiom, we have a rule that lets you leave off the lifetimes here. So this says the same thing:

```
fn foo(input: &u32) -> &u32;
```

When a lifetime in the output also appears in the input (as is the case here), we say that it's *bound* by the input. This gives the compiler something to reason about. Specifically it determines that the output must, somehow, be derived from the input. So the caller of our function can make sure the input stays valid as long as we keep around the output:

```

fn main() {
    let x;
    {
        let y = 0;
        x = foo(&y);
    }
    println!("{}", x);
}
```

```
error: `y` does not live long enough
  --> <anon>:10:5
   |
9  |         x = foo(&y);
   |                  - borrow occurs here
10 |     }
   |     ^ `y` dropped here while still borrowed
11 |     println!("{}", x);
12 | }
   | - borrowed value needs to live until here
```

And the body of our function can make sure the output is derived from the input (or something that lives longer):

```
fn foo1(input: &u32) -> &u32 {
    return input;               // ok, output derived from input
}

static global_int: u32 = 0;
fn foo2(input: &u32) -> &u32 {
    return &global_int;         // ok, globals live longer than everything
}

fn foo3(input: &u32) -> &u32 {
    let temp = 0;               // ERROR: doesn't live long enough
    return &temp;
}
```

```
error: `temp` does not live long enough
  --> <anon>:12:13
   |
12 |     return &temp;
   |             ^^^^ does not live long enough
13 | }
   | - borrowed value only lives until here
   |
```

Ok, so what does the compiler do if you give it a lifetime that *isn't* bound -- an *unbound* lifetime?

```
fn foo1<'a>() -> &'a u32 {
    let temp = 0;
    return &temp;               // ERROR: doesn't live long enough
}

static global_int: u32 = 0;
fn foo2<'a>() -> &'a u32 {
    return &global_int;         // ok, globals live longer than everything
}
```

```
error: `temp` does not live long enough
 --> <anon>:3:13
  |
3 |     return &temp;
  |             ^^^^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
note: borrowed value must be valid for the lifetime 'a as defined on the body at 1:2
5...
 --> <anon>:1:26
  |
1 |   fn foo1<'a>() -> &'a u32 {
  |  __________________________^
2 | |     let temp = 0;
3 | |     return &temp;
4 | | }
  | |_^
```

It prevents us from using anything other than a reference to a global. Ok, that seems reasonable.

But what happens when we throw some unsafe code at the issue?

```
fn foo<'a>(input: *const u32) -> &'a u32 {
    unsafe {
        return &*input
    }
}

fn main() {
    let x;
    {
        let y = 7;
        x = foo(&y);
    }
    println!("hello: {}", x);
}
```

```
hello: 2384275376
```

UHHHH 🙀

It turns out that unbound lifetimes are really bad when unsafe code gets involved.

Basically, if you dereference a raw pointer, this produces an unbound lifetime. This is because we don't know what scope could possibly be right. Unbound lifetimes are basically the "yes men" of the type system -- they change shape to be whatever scope we ask them to fit. This is genuinely useful in the case of raw pointers, because there isn't a better answer. Normally it's *also* fine because we either discard the reference quickly, or we pass it somewhere that causes the lifetime to become bound and act reasonably (a struct field, or the return value of a function).

But if you mess up your signatures and leave an unbound lifetime in there... look out!

Note that our code only compiled because we used an explicit lifetime variable; this doesn't compile:

```
fn foo(input: *const u32) -> &u32 {
    unsafe {
        return &*input
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> <anon>:1:30
  |
1 | fn foo(input: *const u32) -> &u32 {
  |                              ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value with an elided lifet
ime, but the lifetime cannot be derived from the arguments
  = help: consider giving it an explicit bounded or 'static lifetime
```

So one way to keep yourself safe is to use lifetime elision as much as possible.

Here's a (pre-1.0) bug where we messed this up in the Rust standard library. (https://github.com/rust-lang/rust/issues/17500#issue-43719812)

Destructor Leaking - The Leakpocalypse

TL;DR: don't rely on borrows expiring meaning destructors ran!

Every language I know of that has destructors comes with a little asterisk: destructors aren't guaranteed to actually run. Or to put it another way:

> THIS SOFTWARE IS PROVIDED ''AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

In Rust, Swift, and C++, the standard example of this is stuffing a type-with-a-destructor inside a reference-counted cycle. Even when no more references to the data remain, the cycle will keep itself alive and "leak" the destructor forever.

In Swift and C++ this is kinda annoying, but not a big deal. In Rust, it's a bigger deal.

This is because in Rust, unlike the other two languages, "something is no longer alive" is an event that lets you do new things with your still-alive values -- it can free up borrows!

```
let x = 0;
{
    let y = &mut x;
    println!("{}", x); // ERROR: x is borrowed
}
println!("{}", x);     // OK
```

```
error[E0502]: cannot borrow `x` as immutable because it is also borrowed as mutable
 --> <anon>:5:24
  |
4 |         let y = &mut x;
  |                      - mutable borrow occurs here
5 |         println!("{}", x); // ERROR: x is borrowed
  |                        ^ immutable borrow occurs here
6 |     }
  |     - mutable borrow ends here
```

This is an important tool for API design in Rust. It's what lets us safely make iterators for collections without having to copy them or add invalidation checks. It lets us safely keep pointers to reference-counted data without actually touching the reference count. Being able to rely on "a borrow has expired" meaning the thing that held the borrow is gone forever is good and safe.

What you *can't* do is assume that the borrow expiring that was held by a type means that type had its destructor run. For instance, if you do the reference-counted cycle leak trick, Rust will correctly determine that the borrow can safely expire -- after all no one can ever access the reference that was leaked.

The Rust stdlib devs fell awry of this, and had developed two APIs that were relying on this fact. When this was pointed out (https://github.com/rust-lang/rust/issues/24292) there was a bit of a panic in the community, and thus the The Leakpocalypse (https://github.com/rust-lang/rfcs/pull/1066) occured.

In the end we declared that this sort of thing was just a fact of life. However while handling the fallout of this decision, we figured out a way to make this pattern work, sort of.

For example, `drain` was an experimental API that relied on destructors running. It was an iterator that moved the values out of a collection, without actually consuming the collection's allocation -- as if you called `pop` until there was nothing left. Of course, we wanted to do it without having to create any temporary allocations or keep the collection in a valid state.

We reasoned this was safe because the Drain iterator would hold a mutable borrow on the collection, statically preventing anyone from accessing it while removing elements. The destructor of the drain would empty out any elements that hadn't been read yet, and then update the collection's metadata to reflect its emptiness. Perfect!

Except, someone could leak the Drain's destructor, and then the collection would be accessible in a corrupt state.

The solution to this was a trick we called *leak amplification*, because it basically raised the stakes of a destructor leak: if you leak me, I'll leak you right back.

The trick was to change the code that constructed a Drain to tell the collection that it was now empty. So, if at any point the Drain was leaked, the collection would find itself in a perfectly valid state. However this would also result in a leak of all the destructors of all the unprocessed elements being leaked. You leak me, I'll leak you right back.

However if the Drain destructor was allowed to run, which is almost always the case, then it would do all the cleanup and no one would know the difference. Nice!

Sadly, some APIs don't have any way to apply leak amplification, as was the case for the original design of thread::scoped. For that API, we had to force the user of our API to run inside a closure we call, so that we could run any cleanup code we wanted after their closure terminated. This works, it's just less ergonomic and composable.

Honorable Mention: Zero Sized Types (ZSTs)

TL;DR: be sure that `size_of::<T>() == 0` is handled reasonably for allocations and offsets!

Ok so this one actually shows up in a few other languages (at very least Swift and Go), but it's really cool and I want to talk about it. To be fair, I *think* these problems are the worst in Rust.

Rust lets you create types that have nothing in them, and thus have zero size. For example, the empty tuple `()`. Instances of these types are completely useless -- after all they hold no state. However they serve two purposes:

1. As static encodings of properties/facts
2. As ways to "throw away" parts of a generic design

As an example of (1), the author of the Diesel ORM often brags that building up SQL queries in Diesel actually just builds up a really complicated zero-sized type, which the compiler then statically resolves for minimum runtime overhead.

As an example of (2), `HashSet<T>` is implemented as a trivial wrapper around `HashMap<T, ()>`. Because the language knows instances of `()` are useless, all the code in HashMap that manipulates values is completely eliminated, producing an optimized HashSet implementation. As a more extreme example, `Vec<()>` is basically just a glorified counter.

However, in low level (unsafe) code zero-sized types can be a bit hazardous, for two reasons:

- If you try to malloc some number of zero-sized types naively, you'll pass an allocation size of 0 to the allocator, which is implementation-defined at best. Also you want to avoid allocating these things anyway!
- If you try to offset a pointer to a zero-sized type naively, you will get nowhere. This breaks a C-style "two pointers moving towards each other" iterator.

I discuss how to handle these cases in the nomicion (https://doc.rust-lang.org/nomicon/vec-zsts.html).

To the best of my memory (which is bad), the stdlib has never *shipped* with ZST bugs in it, but I've certainly caught these bugs when reviewing patches. Subtle details about ZSTs and allocations were common enough issues in contributions that I ended up building the RawVec abstraction to handle all those details for everyone in the stdlib.

Since I mentioned Go and Swift, I'll briefly note that they do slightly different things here.

Swift decouples size from stride. ZSTs have a stride of 1 (byte), and allocations and offsets key off stride. So this means an Array of 100 empty tuples will allocate a buffer of 100 bytes that are never read/written. This is effectively the same solution that C++ has for empty types, but the decoupling of size and stride lets Swift stuff a bunch of zero-sized types in a struct and still declare that the struct is zero-sized with a stride of 1. I believe C++ would be forced to give every ZST in the struct its own byte.

It seems like Go is closer to Rust in legitimately optimizing away ZSTs, but I'm not super confident in the details. Because Go is garbage collected, it seems like it needs to handle a lot of the details here for you, so ZSTs aren't as big of a concern in practice.

This blog post (https://dave.cheney.net/2014/03/25/the-empty-struct) seems to have some interesting details about ZSTs in Go, but I don't know how accurate it is.

Honorable Mention: Variance

TL;DR: be sure to eat all of your PhantomDatas for breakfast!

Variance is the thing that makes it so that you can pass a Cat to a function that expects an Animal in an object-oriented language, but not an `ArrayList<Cat>` where an `ArrayList<Animal>` is expected. In Rust, variance is applied to lifetimes: you can pass a long-lived thing to a function that expects short-lived things.

Rust is already relatively unique in that it actually exposes type variance as something user-defined types can have (`&MyType<&'long T>` can potentially be passed where `&MyType<&'short T>` is expected). This is because variance is usually made unsound by shared mutability: if you turn an `ArrayList<Cat>` into an `ArrayList<Animal>`, and someone puts a `Dog` in, then anyone still holding the `ArrayList<Cat>` will interpret that Dog as a Cat! As it turns out, Rust's type system is incredibly good at managing aliasing and mutability, so it's easier for us to use variance in more places.

However Rust mostly handles variance in a similar manner to Sync: it's automatic and based on the values stored in the type. This leads to a similar scenario to Sync: once you start using unsafe code, it can get confused and give you variance when you shouldn't have it.

The solution to this is to, whenever your type stores values that Rust can't "see", use PhantomData. `PhantomData<T>` is a cool little language feature that lets you tell the compiler "look I'm not gonna directly store this type, but just pretend I did". It's zero-sized, so it has no impact on your type at runtime.

For instance:

```
struct MySmahtPointer<T> {
  allocation: *mut u8,       // Something complex we can't do in the type syst
em
  _boo: PhantomData<*mut T>, // But hey there's some mutably shared T's in the
re!
}
```

Pre-1.0, Rust actually used to have markers that explicitly requested a "kind" of variance, but it was deemed too confusing in practice. The PhantomData system was adopted because it let you just "say what you're doing" and have the compiler figure it out for you.

Note that in the above example, removing the PhantomData would be a compilation error, as we refuse to let you claim you're generic over a type without actually telling us how you're using it. However there are more subtle definitions which may lead to incorrect results, like this:

```
struct MySmahtPointer<T> {
  allocation: *mut u8,  // Something complex we can't do in the type system
  aux_value: T,         // Compiler thinks this is the only way we're using i
t,
                        // and gives us variance incorrectly.
}
```

I discuss this situation in more detail here (https://doc.rust-lang.org/nomicon/subtyping.html) and here (https://doc.rust-lang.org/nomicon/phantom-data.html).

That's All I Can Think Of!

Lemme know if you can think of any others!