Is it safe to use unsafe?

How 10 open source projects manage `unsafe` code

Tim McNamara @timClicks

Introduction

About me

I tweet about Rust.

I tweet about Rust.

https://twitter.com/timClicks

I waste time on talking about Rust

I waste time on talking about Rust

https://reddit.com/u/timClicks

I do live coding in Rust

I do live coding in Rust

https://twitch.tv/timClicks

I make videos about Rust

I make videos about Rust

https://youtube.com/timClicks

I write books about Rust

I write books about Rust

https://rustinaction.com/

My goal

My goal

To shorten the time it takes for you to learn Rust by 100h.

Unsafe guidelines for the impatient

- Use `#[deny(unsafe_code)]`

- Use `#[deny(unsafe_code)]`
- Add comments to all `unsafe` blocks

- Use `#[deny(unsafe_code)]`
- Add comments to all `unsafe` blocks
- Ask someone who is not familiar with the code to review the comment. If they do not understand why the `unsafe` block is safe, then the comment (or perhaps the code) should be revised.

Objective: safety

Objective: safety

Remember: other people make mistakes

Objective: safety

Remember: other people make mistakes

Your job is to create a system that makes it very hard for a stressed, overworked and distracted staff members from doing the wrong thing.

Let’s learn about how other projects manage the risk of `unsafe` blocks.

Let's learn about how other projects manage the risk of `unsafe` blocks.

But first, a story.

Lemons and limes

Warning

Disallowing `unsafe` blocks in your project is insufficient

Using only safe Rust, it is possible to create code that is guaranteed to crash.

```rust
struct Container<T> {
    storage: Vec<T>,
    position: usize,
}

impl<T> Container<T> {
    fn get(&self) -> &T {
        &self.storage[self.position]
    }

    fn poison(&mut self) {
        self.position = self.storage.len() + 1;
    }
}
```

About the research

Rationale

Rationale

`actix-web`

Aims

Aims

- understand what Rust leaders are doing

Aims

- understand what Rust leaders are doing
- make a case for further research (community standards)

Methodology

Methodology

- *qualitative* research, rather than quantitative

Methodology

- *qualitative* research, rather than quantitative
- select a small sample of public open source projects, primarily from companies that should be doing the right thing and/or are high impact

Methodology

- *qualitative* research, rather than quantitative
- select a small sample of public open source projects, primarily from companies that should be doing the right thing and/or are high impact
- Look for documentation for contributors and code snippets

Projects

servo

Rust's foundational project.

servo

Rust's foundational project.

Really interesting look at how Rust emerged. Many of the documents in the wiki are quite old.

Strategy: deny unsafe

Servo requires `#[allow(unsafe_code)]` at the top of files that require `unsafe` blocks.

```
#[deny(unsafe_code)]
pub mod safe_module {

    #[allow(unsafe_code)]
    pub fn danger() -> i32 {
        unsafe { 42 }
    }

}
```

cargo-geiger

An extension to cargo.

cargo-geiger

An extension to `cargo`.

This project is designed to allow you to assess the use of `unsafe` in your dependencies. How do they manage `unsafe` themselves?

Strategy: forbid unsafe

They have added `#![forbid(unsafe_code]` to the root their crate.

Strategy: forbid unsafe

They have added `#![forbid(unsafe_code]` to the root their crate.

This instructs the compiler that `unsafe` is not legal.

Strategy: forbid unsafe

They have added `#![forbid(unsafe_code]` to the root their crate.

This instructs the compiler that `unsafe` is not legal.

Unlike `#![deny(unsafe_code)]`, it is impossible to opt-in to `unsafe` blocks with `#[allow(unsafe_code)]` later.

```rust
#[forbid(unsafe_code)]
pub mod safe_module {

    // #[allow(unsafe_code)]
    // pub fn danger() -> i32 {
    //     unsafe { 42 }
    // }

}
```

exa

exa is a replacement for `ls`. It is one of the Rust community's oldest utilities.

Why is unsafe needed?

`exa` talks to the file system via syscalls. In particular, it uses them to inspect files' extended attribute names via the `listxttr` syscall family on Linux and macOS.

Strategy: minimal wrappers

```rust
#[cfg(target_os = "macos")]
mod lister {
    use std::ffi::CString;
    use libc::{c_int, size_t, ssize_t, c_char, c_void,
        uint32_t};
    use std::ptr;

    extern "C" {
        fn listxattr(
            path: *const c_char, namebuf: *mut c_char,
            size: size_t, options: c_int
        ) -> ssize_t;
    }

    pub fn listxattr_first(&self, c_path: &CString) -> ssize_t {
```

```rust
#[cfg(target_os = "macos")]
mod lister {
    use std::ffi::CString;
    use libc::{c_int, size_t, ssize_t, c_char, c_void,
        uint32_t};
    use std::ptr;

    extern "C" {
        fn listxattr(
            path: *const c_char, namebuf: *mut c_char,
            size: size_t, options: c_int
        ) -> ssize_t;
    }

    pub fn listxattr_first(&self, c_path: &CString) -> ssize_t {
```

Example taken from `fs::feature::xttr`.

BLAKE3

BLAKE3 is a very fast cryptographic hash function. It is intended to replace SHA-256 and similar functions.

BLAKE3

BLAKE3 is a very fast cryptographic hash function. It is intended to replace SHA-256 and similar functions.

(You have stopped using MD5, right?)

Why is `unsafe` needed?

Why is `unsafe` needed?

BLAKE3 wants to perform at maximum performance.
It uses compiler intrinsics to access SIMD CPU instructions.

Strategy: Minimal wrappers

```rust
#[inline(always)]
unsafe fn add(a: __m256i, b: __m256i) -> __m256i {
    _mm256_add_epi32(a, b)
}

#[inline(always)]
unsafe fn xor(a: __m256i, b: __m256i) -> __m256i {
    _mm256_xor_si256(a, b)
}
```

Strategy: Comments around pointer use

```
#[inline(always)]
unsafe fn loadu(src: *const u8) -> __m256i {
    // This is an unaligned load, so the pointer cast is
        allowed.
    _mm256_loadu_si256(src as *const __m256i)
}

#[inline(always)]
unsafe fn storeu(src: __m256i, dest: *mut u8) {
    // This is an unaligned store, so the pointer cast is
        allowed.
    _mm256_storeu_si256(dest as *mut __m256i, src)
}
```

Firecracker

A new virtual machine manager written in Rust by Amazon Web Services. Powers AWS Lambda and AWS Fargate.

Why do they need `unsafe`?

Why do they need `unsafe`?

They’re interacting with a hypervisor.

Why do they need `unsafe`?

They’re interacting with a hypervisor.

They make heavy use of C++ code, from the Chrome OS project, for example.

Strategy: just blame `rust-bindgen`

```
/* automatically generated by rust-bindgen */
```

Aside: CONTRIBUTING.md

Contribution Quality Standards Most quality and style standards are enforced automatically during integration testing. Your contribution needs to meet the following standards:

- *Separate each logical change into its own commit.*
- *Each commit must pass all unit & code style tests, and the full pull request must pass all integration tests. See tests/README.md for*

tests. See tests/README.md for information on how to run tests.

- *Unit test coverage must increase the overall project code coverage.*
- *Include integration tests for any new functionality in your pull request.*
- *Document all your public functions.*
- *Add a descriptive message for each commit. Follow commit message best practices.*
- *Document your pull requests. Include the reasoning behind each change, and the testing done.*
- *Acknowledge Firecracker's Apache 2.0 license and certify that no part of*

2.0 license and certify that no part of your contribution contravenes this license by signing off on all your commits with git -s. Ensure that every file in your pull request has a header referring to the repository license file.

winrt-rs

A Rust interface (“language projection”) for the Windows Runtime, developed by Microsoft.

Why do they need `unsafe`?

`winrt-rs` interfaces with Windows APIs.

Strategy: short unsafe blocks

```rust
impl<T: RuntimeType> Array<T> {

    /// Creates an array of the given length with default
        values.
    pub fn with_len(len: usize) -> Self {
        assert!(len < std::u32::MAX as usize);

        // WinRT arrays must be allocated with CoTaskMemAlloc.
        let data = unsafe { CoTaskMemAlloc(len *
        std::mem::size_of::<T>()) as *mut T };

        if data.is_null() {
            panic!("Could not successfully allocate for Array");
        }
```

Aside: README.md

Safety

We believe that WinRT bindings can map to 100% safe Rust. However, often times WinRT APIs are implemented in non-memory safe languages (e.g., C++). […]

librsvg

An SVG renderer in Rust by the GNOME project.

Why do they need `unsafe`?

Why do they need `unsafe`?

`librsvg` talks to GLib.

Why do they need `unsafe`?

`librsvg` talks to GLib.

It also exposes a C API.

Strategy: culture of questioning unsafe

Can you please remind me why this is an `UnsafeCell`? The two places where it is accessed are like let `bg = unsafe { &*self.background_surface.get() };` - but I think this could be a safe `RefCell` instead?

https://gitlab.gnome.org/GNOME/librsvg/-/merge_requ

Strategy: minimal wrappers

as seen before

`std`

Rust's standard library.

Strategy: provide guidance for code reviewers

From the code review guidelines:

> *Unsafe code blocks in the standard library need a comment explaining why they're ok. There's a tidy lint that checks this. The unsafe code also needs to actually be ok.*

Strategy: seek input from experts

From the code review guidelines:

The rules around what's sound and what's not can be subtle. See the Unsafe Code Guidelines WG for current thinking, and consider pinging @rust-lang/libs, @rust-lang/lang, and/or somebody from the WG if you're in any doubt. We love debating the soundness of unsafe code, and the more eyes on it the better!

Strategy: Add safety section to unsafe functions

From `std::ptr::read`:

```
/// # Safety
///
/// Behavior is undefined if any of the following conditions are
        violated:
///
/// * `src` must be valid for reads.
///
/// * `src` must point to a properly initialized value of type
        `T`.
///
/// Like `read`, `read_unaligned` creates a bitwise copy of `T`,
        regardless of
/// whether `T` is `Copy`. If `T` is not `Copy`, using both the
        returned
/// value and the value at `*src` can violate memory safety
```

toolshed

An “arena” memory allocator. Typically faster than the system’s memory allocator, but can waste space.

Why do they need `unsafe`?

Lots of pointer manipulation.

Strategy: isolate `unsafe` within specific modules

All the public API of the `Arena` is safe, although the module that implements it makes heavy use of `unsafe` internally.

terminusdb

A new graph database that includes a Prolog reasoning engine.

Why do they need `unsafe`?

To interface with an external system, SWI-Prolog.

Strategy: isolate `unsafe` within a specific crate

The project team has decided to create a Rust interface to Prolog within a completely separate crate. This avoids introducing unsafe into the storage engine itself.

Strategy: add comments around unsafe blocks

Variable-length integer encoding.

```rust
/// Encodes a `u64` with a variable-byte encoding in a `Vec`.
///
/// The length of the resultant `Vec` is the encoding length of
        `num`.
pub fn encode_vec(num: u64) -> Vec<u8> {
    // Allocate a `Vec` of the right size.
    let mut vec = vec![0; encoding_len(num)];
    // Safety: We have created `vec` with the length of the
        encoded bytes of `num`.
    unsafe { encode_unchecked(&mut vec, num) };
    vec
}
```

Strategy: heavy commenting within unsafe functions

```rust
/// Encodes a `u64` by writing its variable-byte encoding to a
        slice.
///
/// Returns the encoding length.
///
/// This function does not ensure that `buf` is large enough to
        include the encoding length of the
/// number. In particular, there are no bounds checks on
        indexing. The caller of this function must
/// ensure that `buf` is large enough for the encoded `num`.
        This can be done, for example, by
/// using `MAX_ENCODING_LEN` to create the `buf` or by using
        `encoding_len` to validate the length
/// of `buf`.
unsafe fn encode_unchecked(buf: &mut [u8], mut num: u64) ->
```

Fuchsia OS

New operating system being developed by Google.

Why do they need `unsafe`?

Fuchsia's kernel (zircon) is not written in Rust.

Strategy: developer's individual responsibility

As a developer (“editor” in Fuchsia’s terminology), you have the primary responsibility for ensuring that you’re safe.

*When writing or reviewing unsafe code, it’s essential that **you**:*

- *clearly identify all of the assumptions and invariants required by every unsafe block;*
- *ensure that those assumptions are met;*
- *ensure that those assumptions will continue to be met.*

— *Unsafe code in Rust* *(emphasis added)*

Strategy: consider future needs

As a developer ("editor" in Fuschia's terminology), you're required to consider the ability for the codebase to maintain invariants into the future.

> *When writing or reviewing unsafe code, it's essential that you:*
>
> - *clearly identify all of the assumptions and invariants required by every unsafe block;*
> - *ensure that those assumptions are met;*
> - ***ensure that those assumptions will continue to be met**.*

— *Unsafe code in Rust* *(emphasis added)*

Strategy: document invariants

In order to ensure that unsafe invariants are not broken by future editors, each usage of unsafe must be accompanied by a clear, concise comment explaining what assumptions are being made.

— *Unsafe code in Rust*

Strategy: isolate unsafety

Where possible, package up unsafety into a single function or module which provides a safe abstraction to the outside world. FFI calls should usually be exposed through a safe function whose only purpose is to provide a safe wrapper around the function in question. These functions should contain a comment with the following information (if applicable):

- *Preconditions (e.g. what are the valid states of the arguments?)*
- *Failure handling (e.g. what values should be free'd? forgotten? invalidated?)*
- *Success handling (e.g. what values are created or consumed?)*

— *Unsafe code in Rust*

Strategy: highlight inherently unsafe types

Finally, struct definitions containing unsafe types such as `*const`, `*mut`, or `UnsafeCell` must include a comment explaining the internal representation invariants of the type.

There are further requirements to guard against specific problematic use cases:

Fuschia allows unsafe for either mutation or aliasing, but not both at the same time.

If the unsafe type is used to perform a mutation OR if it aliases with memory inside another type, there should be an explanation of how it upholds Rust's "aliasing XOR mutation" requirements.

If you've decided to not automatically derive traits *for safety reasons* explain that.

If any deriveable traits are purposefully omitted for safety reasons, a comment must be left to prevent future editors from adding the unsafe impls.

Resources

- Fuchsia OS Team “Unsafe code in Rust”
- Brian Anderson “Rust API Guidelines”
- Ralf Jung (2016) “The Scope of Unsafe”
- Ralf Jung (2018) “Two Kinds of Invariants: Safety and Validity”
- rust-unoffical/patterns contributors (2018) “Contain unsafety in small modules”
- Unsafe Working Group “Unsafe Code Guidelines Reference”

Thank you

// reveal.js plugins