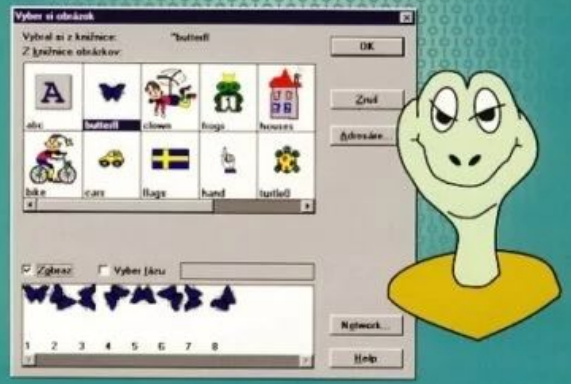
Unsafe Rust

$ whoami

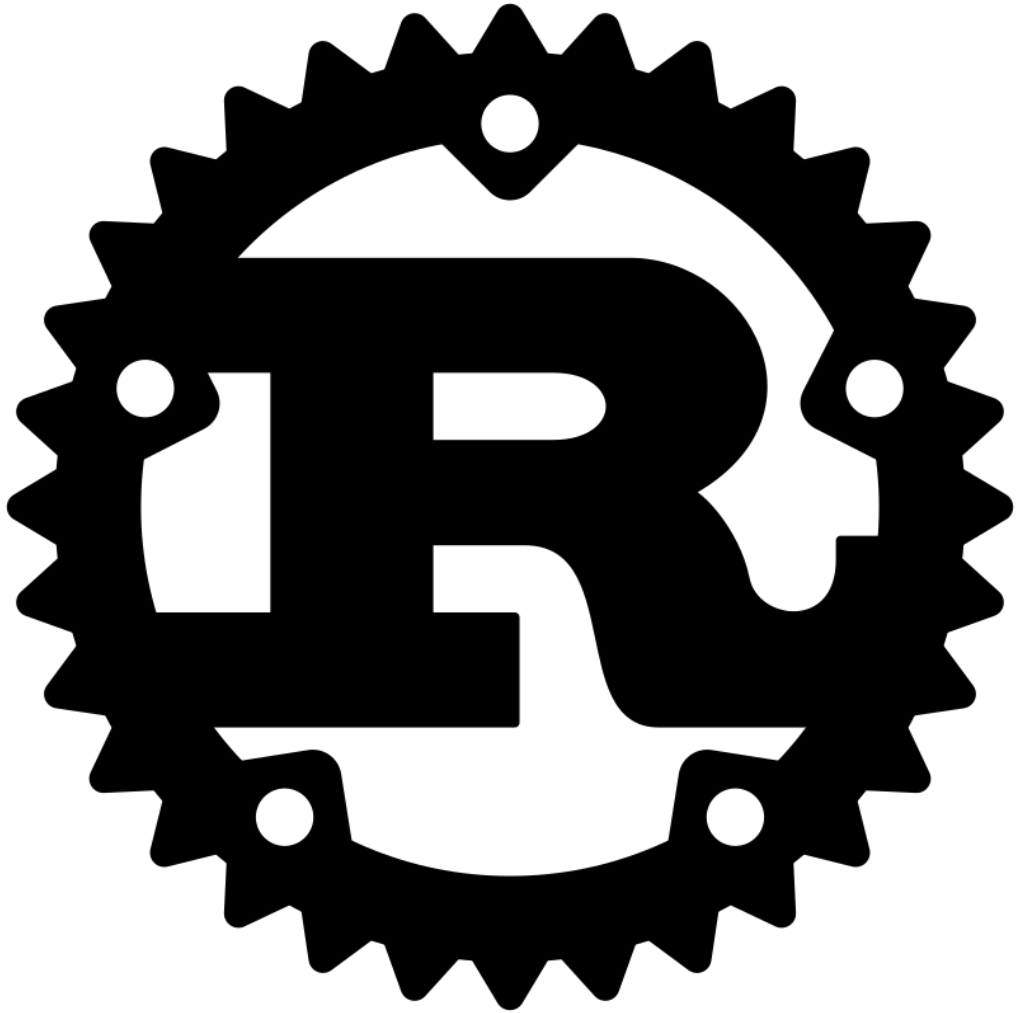
INFORMATIKA

PRE STREDNÉ ŠKOLY

Algoritmy s Logom



Slovenské pedagogické nakladateľstvo



← Warning: Rust Foundation trademark!

`preallocated_gen_new` is unsound #543

Closed **Kixunil** opened this issue on Nov 30, 2022 · 25 comments · Fixed by #548

Kixunil commented on Nov 30, 2022 • edited Contributor

Behold, use-after-free!

```
use secp256k1::Secp256k1;

fn make_bad_secp() -> Secp256k1::<secp256k1::AllPreallocated<'static>> {
    // in principle `Box` is not needed but if it's not here it won't show up as a crash.
    let mut array = Box::new([secp256k1::ffi::types::AlignedType::ZERO; 1024]);
    secp256k1::Secp256k1::<secp256k1::AllPreallocated<'static>>::preallocated_gen_new(&mut *array).unwrap()
}

fn main() {
    let secp = make_bad_secp();
    let secret = secp256k1::SecretKey::from_slice(b"release the nasal daemons!!!!!!!").unwrap();
    let pubkey = secp256k1::PublicKey::from_secret_key(&secp, &secret);
    println!("Dear compiler, do not optimize this computation {}", pubkey);
}
```

Yep, no `unsafe`. Curiously, this seems to corrupt the heap on my machine or something because it crashes *after* printing the key. Anyway, it's still UB so I'm lucky I didn't get nasal daemons. 🤷

But, but, but there's `C: 'buf` bound!

The problem is `C: 'buf` means `C` outlives `'buf`, so in our case `'static 'buf` which is trivially true for all lifetimes `'buf`. In theory the bound should've been inverted: `'buf: C`. But sadly, that's not a valid Rust syntax.

The bad news is I see no way of avoiding putting lifetimes everywhere with the current design.

Implementation of `BufMut` for `&mut [u8]` is unsound #328

Closed **Kixunil** opened this issue on Nov 27, 2019 · 19 comments

Kixunil commented on Nov 27, 2019

Example how safe code can cause UB using `BufMut`:

```
let mut buf = [42];

BufMut::bytes_mut(&mut buf)[0] = MaybeUninit::uninit();

println!("Reading this value is UB: {}", buf[0]);
```

As discussed at rust-lang/rust#66699 casting `&mut [T]` to `&mut MaybeUninit<T>` isn't safe. Thus `bytes_mut` must return a newtype that prevents overwriting value with invalid values. See how I addressed it in my crate `possibly_uninit`.

It'd be the best to share common newtype slice implementation to not fragment the ecosystem. You can expect full support from me when it comes to using my crate as a dependency. That includes making you maintainers, if you're interested.

👍 1

Maybe `BufMut` trait should be `unsafe`? #329

Closed **Kixunil** opened this issue on Nov 27, 2019 · 17 comments · Fixed by #432

Kixunil commented on Nov 27, 2019

I was thinking that it's likely that some `unsafe` code will rely on properties of `BufMut`. More specifically:

- That `remaining_mut` returns correct value
- `bytes_mut` always returns the same slice (apart from `advance()`)
- `has_remaining_mut` returns correct value
- `bytes_vectored` fills the correct data

Thus, I'd suggest making the trait `unsafe` to specify that `unsafe` code might rely on those properties and the implementors must ensure they hold.

carllerche commented on Oct 16, 2020 Member

@**Kixunil** You are correct that a caller of `BufMut` is not able to defend against a broken implementation. According to the snippet you pasted, `BufMut` should be `unsafe`.

👍 1

Safety hazard: thread::park() doesn't guarantee being unwind-free but sync::Once implicitly relies on it being unwind-free #89571

Closed **Kixunil** opened this issue on Oct 5, 2021 · 1 comment

Kixunil commented on Oct 5, 2021 Contributor

I noticed `Once` contains this code:

```
        // We have enqueued ourselves, now lets wait.
        // It is important not to return before being signaled, otherwise we
        // would drop our `Waiter` node and leave a hole in the linked list
        // (and a dangling reference). Guard against spurious wakeups by
        // reparking ourselves until we are signaled.
        while !node.signaled.load(Ordering::Acquire) {
            // If the managing thread happens to signal and unpark us before we
            // can park ourselves, the result could be this thread never gets
            // unparked. Luckily `park` comes with the guarantee that if it got
            // an `unpark` just before on an unparked thread it does not park.
            thread::park();
        }
```

As the comment correctly explains the while loop must **not** end prematurely. However if `thread::park()` unwinds it would break the loop.

Does `thread::park()` panic though?
I found at least two instances: 1 2 where it explicitly could and I didn't review all implementations, nor internals of `Mutex` and `CondVar`. I understand those are supposed to be unlikely but I guess they were intended to reduce unsafety but amusingly they increased it for `Once`.

I think it'd be safer to either:

- Guarantee that `park` can't unwind, change those panics to aborts and document it, ideally make this a public guarantee as well. (I noticed this issue because I'm writing something with structure similar to that piece in `Once`)
- Explicitly document that this is **not** guaranteed with a warning about this footgun and change `Once` to abort if `park` panics.

Assignees
No one assigned

Labels
A-atomic I-unsound T-libs

Projects
None yet

Milestone
No milestone

Development
No branches or pull requests

5 participants

What is unsafe?

Unsafe, unsound, undefined behavior…

Unsafe

Not checked by the compiler

Unsound API

A module or library API that allows **external safe** code to cause UB.

Undefined behavior

The compiler may arbitrarily mess up your code and you don’t know how

Vulnerability (regarding memory bugs)

The compiler managed to mess up the code such that an attacker can abuse it.

(AKA shit hit the fan)

When unsafe?

When unsafe?

- C FFI

When unsafe?

- C FFI
- Performance

Misconceptions

“Unsafe turns off the borrow checker”

```
1  fn main() {
2      let mut x = "foo".to_owned();
3      unsafe {
4          let y = &x;
5          let z = &mut x;
6          dbg!(y);
7          z.push('x');
8      }
9  }
10
```

```
error[E0502]: cannot borrow `x` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:13
  |
3 |     let y = &x;
  |             -- immutable borrow occurs here
4 |     let z = &mut x;
  |             ^^^^^^ mutable borrow occurs here
5 |     dbg!(y);
  |          - immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `playground` (bin "playground") due to previous error
```

```
1  fn main() {
2      let mut x = "foo".to_owned();
3      unsafe {
4          let y = &*(&x as *const String);
5          let z = &mut x;
6          dbg!(y);
7          z.push('x');
8          dbg!(y);
9      }
10 }
```

“You are allowed to break Rust borrow rules in unsafe blocks”

```
     Running `/playground/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/bin/cargo-miri runner target
error: Undefined Behavior: trying to retag from <2926> for SharedReadOnly permission at alloc1457[0x0], bu
 --> src/main.rs:6:9
  |
6 |         dbg!(y);
  |         ^^^^^^^
  |         |
  |         trying to retag from <2926> for SharedReadOnly permission at alloc1457[0x0], but that tag does
  |         this error occurs as part of retag at alloc1457[0x0..0x18]
  |
  = help: this indicates a potential bug in the program: it performed an invalid operation, but the Stacke
  = help: see https://github.com/rust-lang/unsafe-code-guidelines/blob/master/wip/stacked-borrows.md for f
help: <2926> was created by a SharedReadOnly retag at offsets [0x0..0x18]
```

“Unsound API is OK if it’s not abused”

Function totally_safe_transmute::totally_safe_transmute

source · [-]

```
pub fn totally_safe_transmute<T, U>(v: T) -> U
```

“UB is OK if the code works”

Falsehoods programmers believe about undefined behavior

November 27, 2022 compilers intro

Undefined behavior (UB) is a tricky concept in programming languages and compilers. Over the many years I've been an industry mentor for MIT's 6.172 Performance Engineering course,[1] I've heard many misconceptions about what the compiler guarantees in the presence of UB. This is unfortunate but not surprising!

https://predr.ag/blog/falsehoods-programmers-believe-about-undefined-behavior/

Footguns

Missing UnsafeCell

Variance

Out of bounds access

Alignment

Atomics?

Read the docs!

Patterns

Bindgen

Trivial functions

```
1  extern "C" {
2      fn write(fd: i32, buf: *const u8, len: usize) -> i32;
3  }
4
5  fn safe_write(fd: i32, bytes: &[u8]) -> i32 {
6      unsafe { write(fd, bytes.as_ptr(), bytes.len()) }
7  }
8
9  fn main() {
10     safe_write(1, "hello".as_bytes());
11 }
```

```rust
extern "C" {
    fn create_foo() -> *mut std::ffi::c_void;
    fn update_foo(foo: *mut std::ffi::c_void, num: u32);
    fn destroy_foo(foo: *mut std::ffi::c_void);
}

struct Foo(*mut std::ffi::c_void);

impl Foo {
    fn new() -> Self {
        unsafe { Foo(create_foo()) }
    }

    fn update(&mut self, num: u32) {
        unsafe { update_foo(self.0, num) }
    }
}

```

```
19 impl Drop for Foo {
20     fn drop(&mut self) {
21         unsafe { destroy_foo(self.0) };
22     }
23 }
24
25 fn main() {
26     let mut foo = Foo::new();
27     foo.update(42);
28 }
29
```

Copy std code!

Caveat

```
// FIXME:
// `Path::new` current implementation relies
// on `Path` being layout-compatible with `OsStr`.
// When attribute privacy is implemented, `Path` should be annotated as `#[repr(transparent)]`.
// Anyway, `Path` representation and layout are considered implementation detail, are
// not documented and must not be relied upon.
pub struct Path {
    inner: OsStr,
}
```

Search for a library!

Ask for help!

Unsafe is dangerous

But not evil