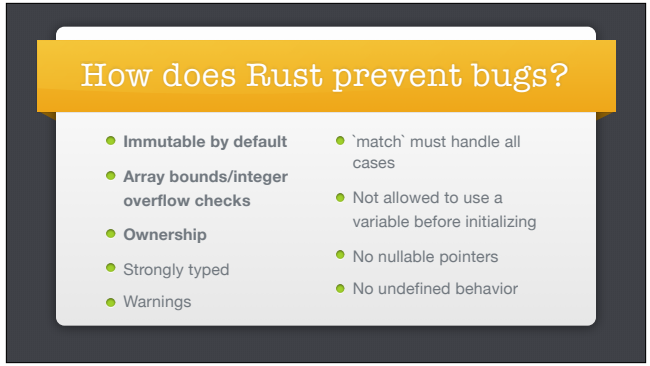
This is a talk I gave as a guest lecture at a software testing class at Pitt; updated to include a new case study.

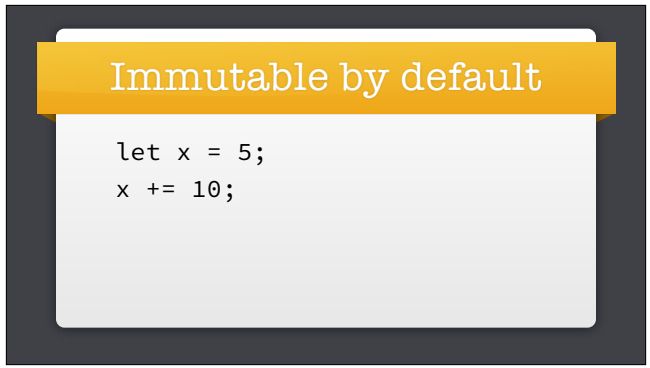
What languages are people coming from?
How long have people used Rust?

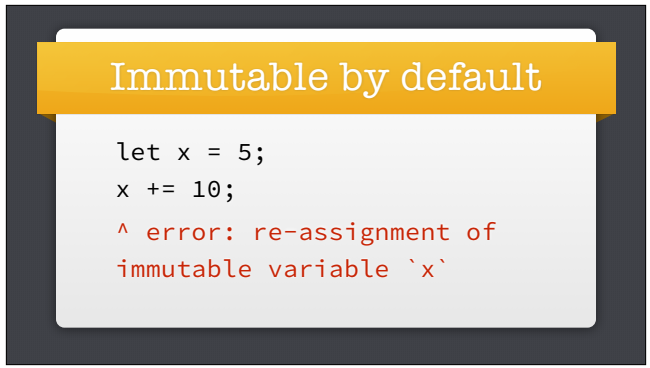
Our agenda for this evening



There are a lot of features that Rust has that have caught bugs that I personally was about to write into my code. Part of why I'm interested in Rust is I'm swinging away from Ruby— tired of getting undefined method on nil!

We're going to concentrate on these first three for tonight - 1st two quickly, more time on ownership.





Immutable by default

```
let mut x = 5;
x += 10;
```

Explicit about what might change, if mut isn't there, don't have to think about what might change that var
Errors if code tries to change something you thought shouldn’t change.
Encourages a functional programming style,
good for concurrency
if you need mutability for memory usage, speed, or convenience, it's there.

Array Bounds Checking

```
let buffer = [10, 20, 30];
println!("{}", buffer[100]);
```

Array Bounds Checking

```
let buffer = [10, 20, 30];
println!("{}", buffer[100]);
thread '<main>' panicked at
'index out of bounds: the len
is 3 but the index is 100'
```

This is a runtime error, since your array length and index could be dependent on user input and therefore couldn't be checked at compile time.
Prevents buffer over/under reads/writes
We'll see how this can prevent bugs in the case study section.

Integer Overflow Checking

```
let x = 5u8;
let y = 255u8;
println!("x + y is: {}", x + y);
```

u8 is an unsigned 8-bit number

255 is the max value that a u8 can hold

Integer Overflow Checking

```
let x = 5u8;
let y = 255u8;
println!("x + y is: {}", x + y);
```

```
thread '<main>' panicked at 'arithmetic
operation overflowed', src/main.rs:4
```

Also a runtime error.

It's assumed that the + sign will never overflow— can explicitly use wrapping, saturating, overflowing, or checked add if that's what you want

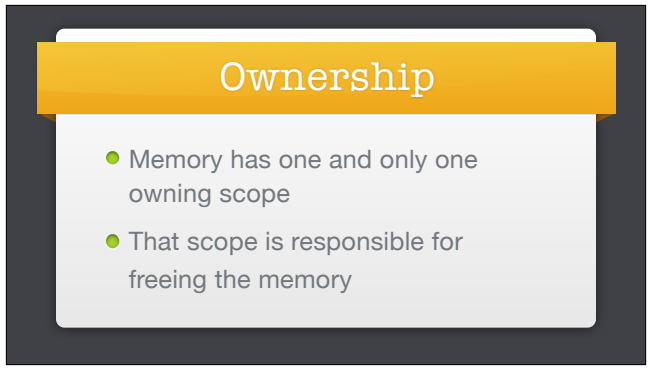
---

NOTE!

- Integer overflow checks are by default:
  - ON in debug mode
  - OFF in release mode
- Can change this to be on in release mode too

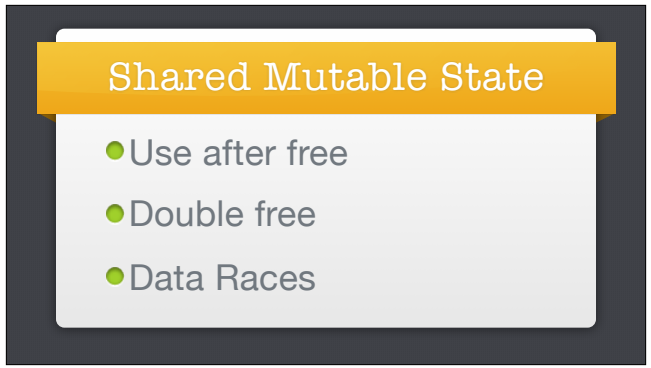
When these checks are off, array access is allowed and integers wrap. We'll see an example of changing this setting later.

Ownership is the most different concept in Rust, and one of the most important for ensuring safety as well as eliminating the need for garbage collection or explicit memory management.



This is one way of describing the ownership system.
When you allocate memory on the heap (new instances), gotta clean it up

---



One of the largest sources of bugs in programming in general. In languages like C where you have to manually manage your memory, dealing with shared mutable state can manifest itself in these ways

Lexical scoping

```
fn main() {
    let x = Thing::new();
    if x.floopy() {
        let y = OtherThing::new(x.florp());
        println!(“y is {}”, y);
    }
    // y is no longer valid here, x still is
}
```

x is in scope from where it's declared to the end of main, y is in scope until the end of the if block.

When x and y go out of scope, Rust automatically frees the memory.

What about calling functions with arguments

Move by default

```
fn main() {
    let x = Thing::new();
    whatevs(x);
    println!(“x is {}”, x);
}

fn whatevs(t: Thing) {
   println!(“I just want to print {}”, t);
}
```

```
fn main() {
    let x = Thing::new();
    whatevs(x);
    println!(“x is {}”, x);
}                       ^ error: use of moved
                        value: `x`
fn whatevs(t: Thing) {
    println!(“I just want to print {}”, t);
}
```

This is annoying and unexpected when people pick up Rust.

Borrowing

```
fn main() {
    let x = Thing::new();
    whatevs(&x);
    println!(“x is {}”, x); // cool.
}

fn whatevs(t: &Thing) {
    println!(“I just want to print {}”, t);
}
```

Difference: ampersand at caller and callee indicates borrow
whatevs returns thing, does not clean it up
main is still the owner and does clean up

```
fn whatevs(t: &Thing) {
    t.price += 50;
}
```

Borrows are immutable by default too. If instead of printing, we want to change thing

---

```
fn whatevs(t: &Thing) {
    t.price += 50;
}    ^ error: cannot assign to
     immutable field `t.price`
```

```
fn whatevs(t: &mut Thing) {
    t.price += 50;
}
```

so we can tell by looking at the signature which functions are going to change what.

```
fn main() {
    let mut x = Thing::new();
    whatevs(&mut x);
    println!("x is {}", x);
}

fn whatevs(t: &mut Thing) {
    t.price += 50;
}
```

Borrowing Rules

- Many immutable borrows
- Only one mutable borrow
  (and no immutable borrows)

```
let mut list = vec![1, 2, 3];
for i in &list {
    println!("i is {}", i);
    list.push(i + 1);
}
```

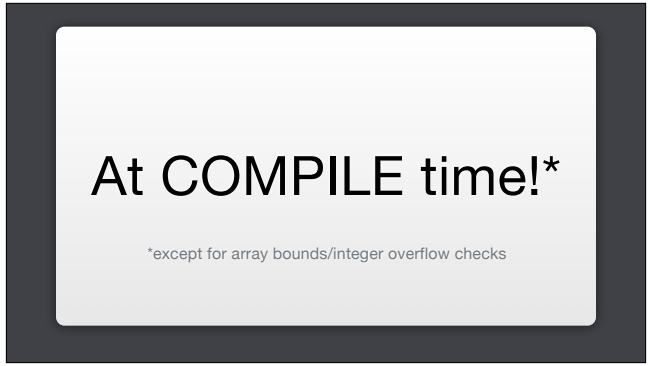
What is this code doing?

```
let mut list = vec![1, 2, 3];
for i in &list {
    println!("i is {}", i);
    list.push(i + 1);
}   ^ error: cannot borrow `list`
    as mutable because it is also
    borrowed as immutable
```

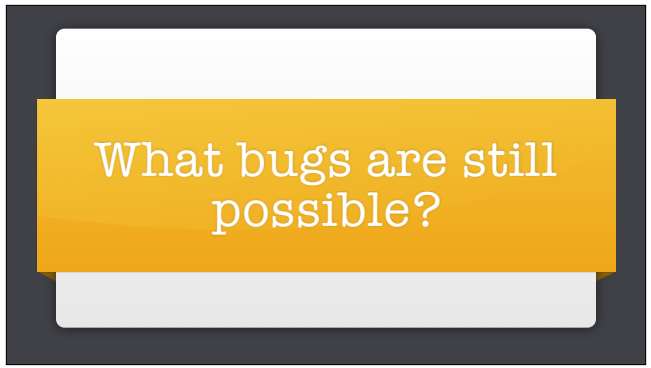
These rules apply when using multiple threads too, and make concurrency safe.

(example left as an exercise for the reader)

BUT WAIT!
THERE'S MORE!

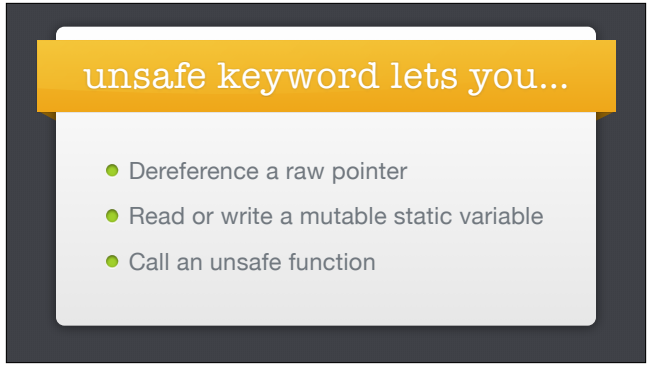


Why is this important?
- what affects the cost of fixing a bug
- what if these checks happened at runtime instead



Is Rust a silver bullet?

i kinda lied



Lets you opt out of the compiler's checks. A way to say "I, the human, have checked this code and found it to uphold the restrictions of safety, so you, the compiler, should let me run whatever is in here"

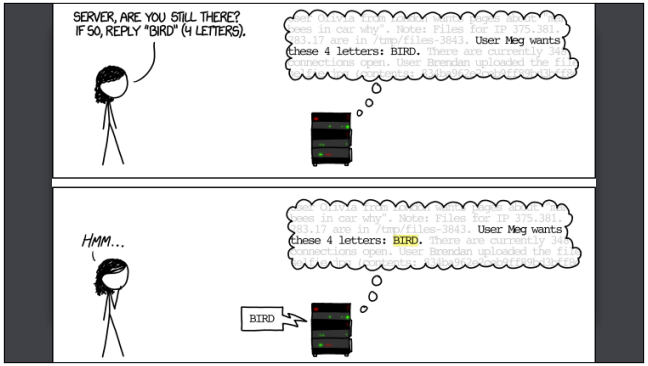
What's a big kind of bug that is still possible in Rust?
Hint: Would you still need to test Rust code?

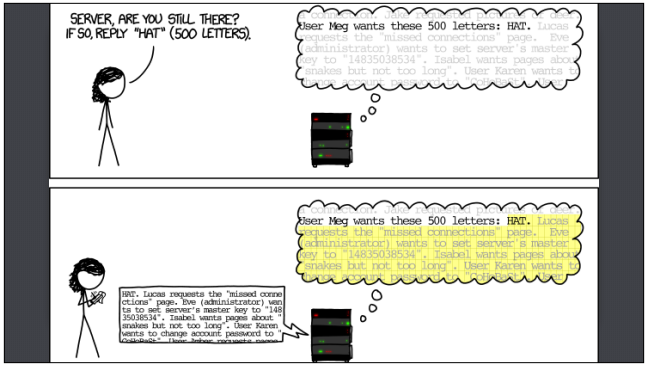
A compiler is never going to be able to check that you're doing the thing that you want to be doing.

You still need tests, you still need fuzzing

Case Studies

Heartbleed
April 2014

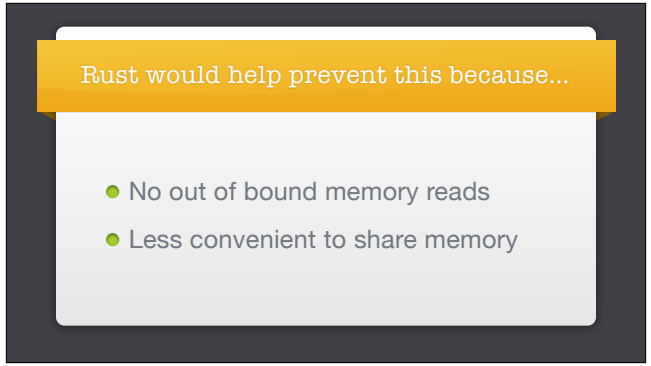


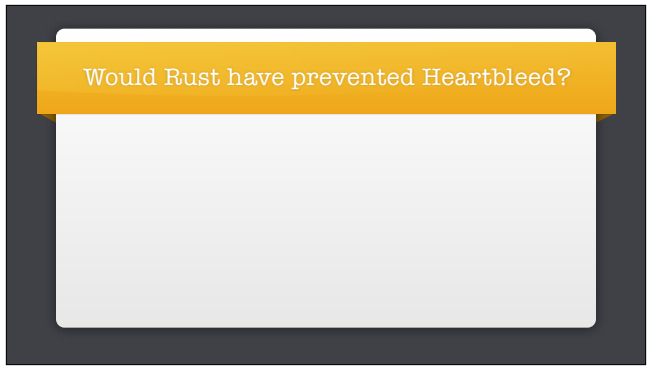


Root causes

- Buffer overread
- Did not validate user input
- Reusing a buffer
- Wrote their own memory allocator

- Not idiomatic C

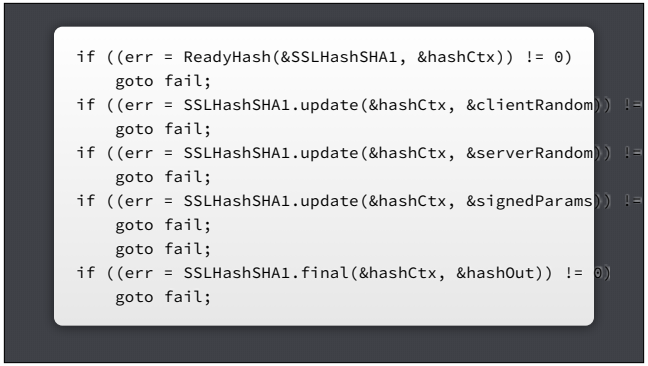




You can write crappy code in any language.

Does Rust make it less likely that you'll write bad code?

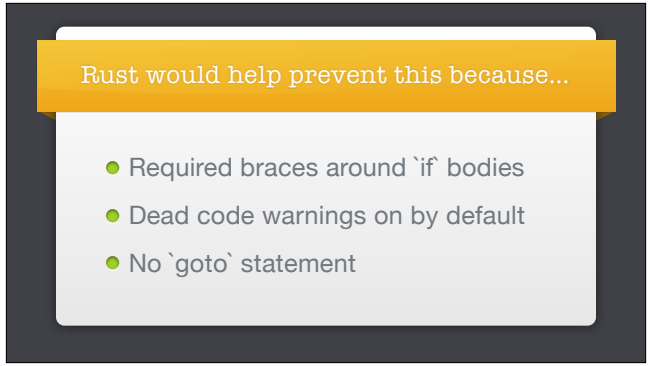
Verifying a signature of a handshake for a secure connection

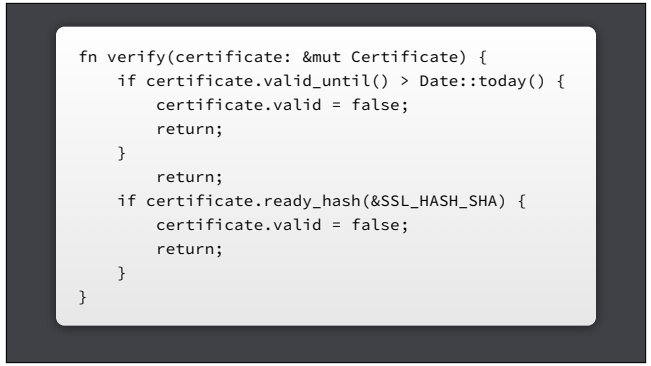


```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) !=
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) !=
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) !=
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

If something isn't right, we need to set `err` and THEN goto fail. Skipping some of these checks means invalid things might get through WITHOUT getting the value in `err` that they should have, and will look like success.

- bad alignment
- no curly braces
- `goto` makes it hard to follow code
- dead code

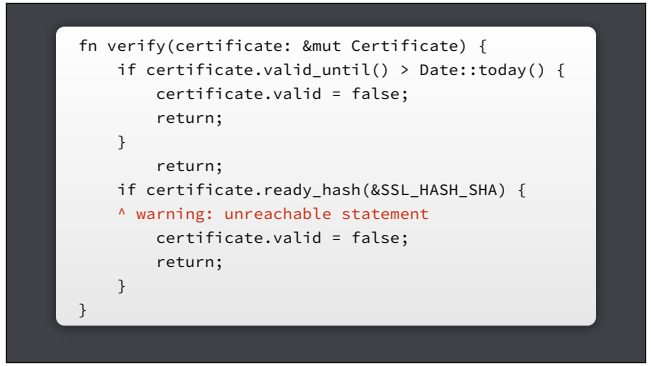




```
fn verify(certificate: &mut Certificate) {
    if certificate.valid_until() > Date::today() {
        certificate.valid = false;
        return;
    }
        return;
    if certificate.ready_hash(&SSL_HASH_SHA) {
        certificate.valid = false;
        return;
    }
}
```

This is my best attempt at reproducing goto fail in Rust; kinda cheating with bad indenting here

---



```
fn verify(certificate: &mut Certificate) {
    if certificate.valid_until() > Date::today() {
        certificate.valid = false;
        return;
    }
        return;
    if certificate.ready_hash(&SSL_HASH_SHA) {
    ^ warning: unreachable statement
        certificate.valid = false;
        return;
    }
}
```

Rust's warnings are generally helpful, I've been keeping them turned on and I resolve warnings as I go to be able to see new ones as I introduce them to help me catch bugs.

Would Rust have prevented `goto fail`?

Would Rust have prevented `goto fail`?

Probably.

---

libstagefright

libstagefright

- C++ library to parse of media metadata
  - From Android, also used in Firefox
- Arbitrary files from possibly-bad actors
- On Android, possibly parsed without any user action

– mention bug where firefox team was considering which media parser to use

---

```
uint8_t *buffer = new (std::nothrow)
    uint8_t[size + chunk_size];
                  ^ from the mp4
                    metadata
```

Allocate a buffer that is size + chunk_size big.

Later code assumes that the buffer is at least size big.

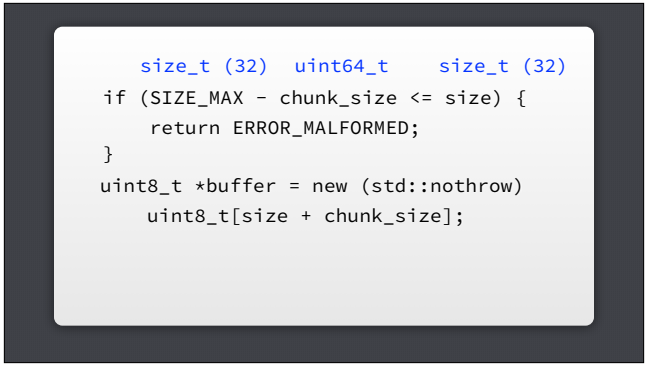
If size + chunk_size overflows, and thus wraps around, the buffer will be less than size and violate a variety of assumptions

---

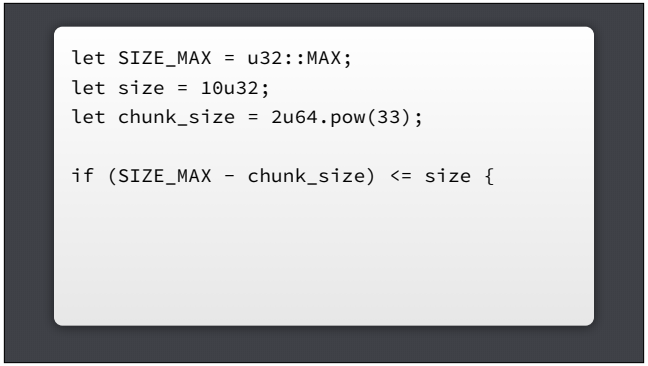
```
+ if (SIZE_MAX - chunk_size <= size) {
+     return ERROR_MALFORMED;
+ }
  uint8_t *buffer = new (std::nothrow)
      uint8_t[size + chunk_size];
```

Fix #1: check for this overflow condition and disallow it

But there's a problem. If you were reviewing this patch, with the threat of a security vulnerability being out there, would you see it?

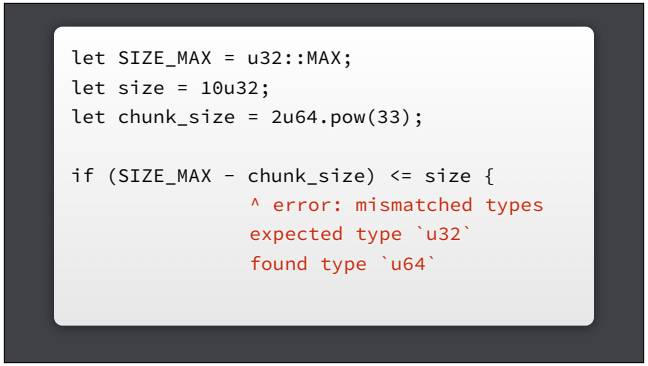


C lets you do arithmetic operations on integers of different types :(



If we try to translate this directly to Rust, even if there's lots of code in between the declarations and the use of these variables,

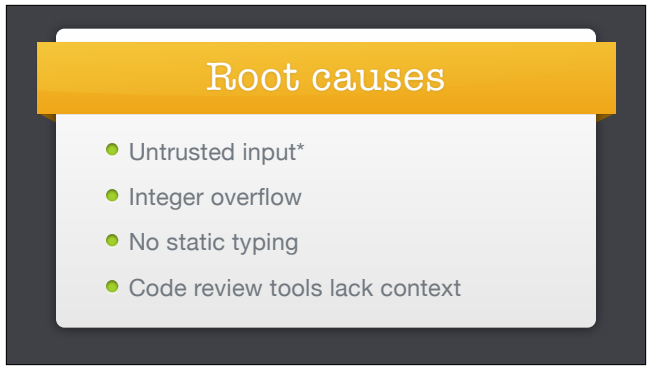
---



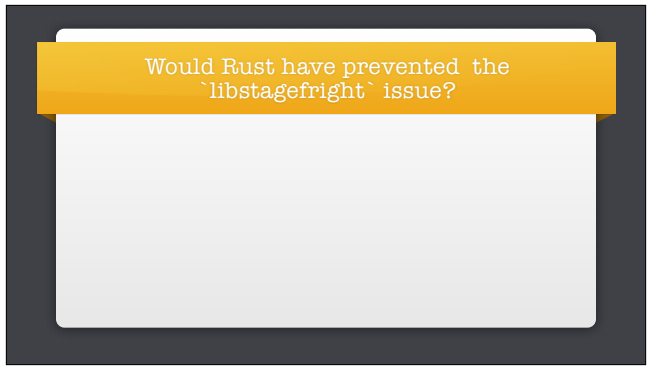
when we try to compile and use this code, Rust will complain.

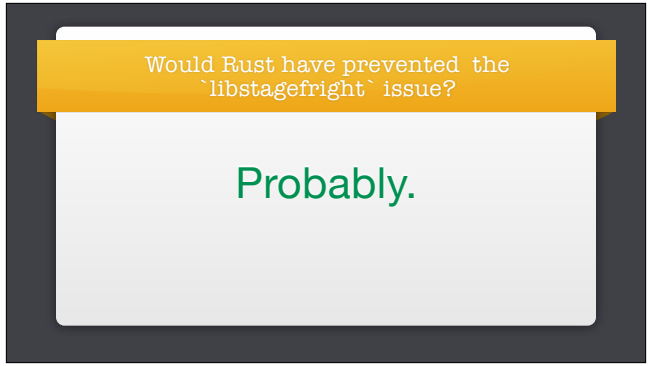
```
let SIZE_MAX = u32::MAX;
let size = 10u32;
let chunk_size = 2u64.pow(33);

if (SIZE_MAX - chunk_size as u32) <= size {
```

casting as u32 is.. kinda weird, but the logic works out
casting SIZE_MAX and size as u64 also works— gotta have overflow checks on.



- untrusted input comes with this territory





- Has integer overflow checks turned on in release mode
- They've done fuzzing
- I tried to find the part in their code like the bug, but it's not a translation, it's a reimplementation in idiomatic rust so looks pretty different. Or I don't understand the code well enough to spot it.

“Let them have knives!”

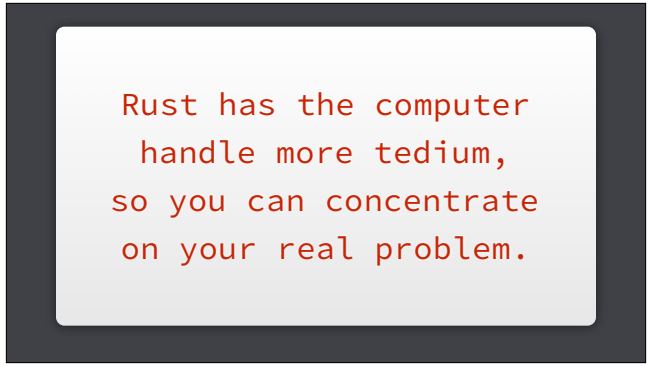
"The best way to prevent these kinds of attacks is either to use a higher level language, which manages memory for you (albeit with less performance), or to be very, very, very, very careful when coding. **More careful than the entirety of the Android security team, for sure.**"

xda-developers.com/a-demonstration-of-stagefright-like-mistakes/

Computers are good at tedium.

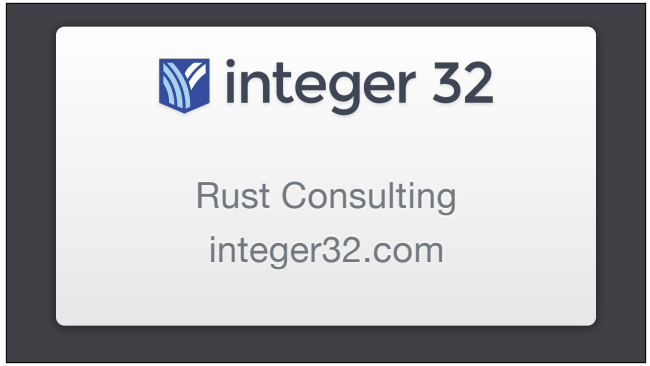
⚠Beep, boop. You forgot a semicolon in 23,982 places

Created by Dillon Arloff
from Noun Project



- Rust is not the only tool to do this – we NEED many different ways to make programming correctly easier
- I want to keep seeing innovation like this
- We don't have to keep creating the same bugs over and over


Rust Belt Rust Conference
Thurs Oct 27-Fri
016
rust-belt-rust.co
@rustbeltrust
ON SALE
NOW!!!


integer 32
Rust Consulting
integer32.com
