Secure Software Development

Defensive III (Rust)

Lukas Prokop

03.12.2021

Winter 2021/22, www.iaik.tugraz.at/ssd

rust overview

1. Introduction to rust

   primitive types, functions, strings, structs, traits, modularization, references, arrays and slices, pattern matching and error handling, macros, typestate pattern

2. Memory safety

   Mutation xor aliasing, ownership model and borrowing, advanced types, unsafe superpowers, undefined behavior

3. Conclusion

   Preventing vulnerabilities, academic rust, resources

Introduction to rust

rust overview

- multi-paradigmatic
  (imperative, functional)
- systems programming language
  (easy interop with C, no GC)
- focus on memory safety and concurrency
- uses the LLVM infrastructure
- syntax similar to C++, immutability by default
- Modern competitors: Nim, Crystal, D, Zig, Go?
- 1.0 (May 2015), 1.57 (current), Rust 2021 edition

“Most loved programming language”
(Stack Overflow Developer Survey, 2016–2021)

Getting started

https://www.rust-lang.org/

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

cargo new --bin NAME create new project with a main function

cargo new --lib NAME create new project with a library

cargo test run testsuite

rustup update update your toolchain

rustup default nightly switch to nightly branch (as opposed to stable/beta)

rustup target add riscv32imac-unknown-none-elf add RISC-V backend

Playground: https://play.rust-lang.org/

Primitive types

```
u8    u16    u32    u64    u128
i8    i16    i32    i64    i128
 isize    usize    f32    f64
         bool    char
```

→ type suffix notation: `42u8`

```
42   42_000   0xFF   0o777   0b0010_1010   std::u32::MAX
1.   1e6   -4e-4f64   std::f64::INFINITY   std::f64::NAN
1usize   true   false   'c'
```

→ type inference to determine data type

→ type conversion with `as` keyword like `6u32 as u64`

Functions

```rust
fn square(arg: f64) -> f64 {
    arg * arg
}

fn main() {
    let a: f64 = 2.1;
    println!("the square of {} is {}", a, square(a));
}
```

Anonymous functions

```
fn named(name1: T1, name2: T2) -> T_RETURN {}
let unnamed = |name1: T1, name2: T2| -> T_RETURN { };
let short   = |name1    , name2    |             { };
```

Example usage:

```
let handler = std::thread::spawn(|| {
    println!("Hello World!");
});
handler.join().unwrap();
```

Strings

- Strings are encoded in UTF-8 in rust.
- `&str` is a pointer into the `.data` section

```
let lang: &str = "rust";
```

- `String` is a pointer to a string on the heap

```
let mut page: String = "rust".to_string();
page.push_str("-lang.org");
assert_eq!(page, "rust-lang.org");
```

- So what is a C-like string then?
  - `Vec<u8>`
  - `std::ffi::CStr`
  - `std::ffi::OsString` and `&std::ffi::OsStr`

Struct, enum, union

```rust
struct Student {
  id: u64,
  age: u8,
}

enum Participant {
  Lecturer,
  Student,
  TeachingAssistant,
}

#[repr(C)]
union MeasurementValue {
  int: u32,
  fp: f32,
}
```

```rust
impl Student {
  fn is_adolescent(&self) -> bool {
    self.age < 18
  }
}
```

```rust
trait Submission {
  fn len(&self) -> u32;
  fn summary(&self) -> String;
}
```

- `trait`s are inspired by Haskell typeclasses (no subtyping); like interfaces
- Nominal type system (like C++/Java/C#), not structural type system (like Go)
- default implementations and constant attributes can be provided
- `struct`s, `enum`s, and `union`s can implement traits
- first parameter is not `&self`? Then static method
- Implementing `std::ops::Add`? Enables `+` (operator overloading)

trait implementation

```rust
struct Talk {
  desc: String,
  duration: u16,
}
impl Submission for Talk {
  fn len(&self) -> u32 { self.duration as u32 }
  fn summary(&self) -> String {
    let dot = self.desc.find('.');
    match dot {
      Some(idx) => {
        let mut s = String::new();
        s.push_str(&self.desc[0..idx]);
        s.push_str(" …");
        s
      },
      None => self.desc.clone(),
} } }
```

```rust
use std::vec::Vec as V;
pub fn exclaim(x: &&str) -> String {
  let mut s = x.to_string();
  s.push_str("!");
  s
}
pub fn hash_map_example() {
  let calls: V<&str> = vec!["Hey", "You"];
  let shouts = calls.iter().map(exclaim);
  println!("He shouted: {}", shouts.collect::<V<String>>().join(" "));
}
```

1. **use** for import, **as** for renaming
2. functional elements like map, zip, filter
3. **pub** to expose functions publicly, modules are called crates

References

Shared references in rust

```
fn main() {
  let mut a = 42u32;
  let b: &u32 = &a;

  println!("value of ref: {}", *b);
}
```

- b is a [shared] reference (&u32) to a.
- Reference operator &
- Dereference operator *

Mutable references in rust

```rust
fn main() {
  let mut a = 42u32;
  let b: &mut u32 = &mut a;

  *b = 2;
  println!("value of ref: {}", *b);
}
```

- b is a [*mutable*] reference (`&u32`) to `a`.
- Reference operator `&mut`
- Dereference operator `*`

Auto-dereferencing in rust

```rust
fn main() {
  let mut a = 42u32;
  let b: &u32 = &a;

  println!("value of ref: {}", b);
}
```

Recognize that **b** does not need the dereference operator.

Rust implements *auto-dereferencing*.

Arrays and slices

Arrays

```rust
// declaration and initialization
let mut array: [u32; 3] = [0; 3]; // [{init-value}; {length}]

// indexing and assignment
array[1] = 1;
array[2] = 2;

// iterate over an array
for x in array.iter() { dbg!(&x); }
```

Arrays

- Arrays (like `[u8; 42]`) have a known, fixed size
- Arrays need to be initialized
  - compile time checks
  - exceptions via MaybeUninit
- Memory layout: consecutive memory segment
- Few API limitations for arrays of length >32

Slices

```
array[0..21]
```

- Slices (like `[u8]`) are memory views into an array
- Unknown size
- Memory layout: only a pointer
- Barely useful because they cannot be passed as fn argument or return value

References to slices

```
&array[0..21]
```

- References to slices (like `&[u8]`) are references to memory views into an array
- **known** size, see `len()` method
- Memory layout: pointer with length
- Similar performance characteristics like an array

Pattern matching and error handling

```rust
fn main() {
  let course = "ssd";
  println!("{} ({})",
    match course {
      "ssd" => "Secure Software Development",
      _ => "unknown"
    },
    course.to_uppercase()
  );
}
```

algebraic data types

*In computer programming, especially functional programming and type theory, an **algebraic data type** (ADT) is a kind of composite type, i.e., a type formed by combining other types.*
—*Wikipedia*

```rust
enum List {
  Nil,
  Cons(Box<List>, u32),
}
```

Algebraic data types in rust

```rust
enum List {
  Nil,
  Cons(Box<List>, u32),
}
```

- Boxing? Avoids `recursive type `List` has infinite size`.
- Article: Algebraic data types in four languages (namely Haskell, Scala, rust, and TypeScript)

Algebraic data types in rust

```
enum List {
  Nil,
  Cons(Box<List>, u32),
}
```

- Boxing? Avoids `recursive type `List` has infinite size`.
- Article: Algebraic data types in four languages (namely Haskell, Scala, rust, and TypeScript)

```
enum Tree {
  Empty,
  Leaf(u32),
  Node(Box<Tree>, Box<Tree>),
}
```

```rust
impl fmt::Display for List {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    match self {
      List::Cons(inner, item)
        => write!(f, "(cons {} {})", item, inner),
      List::Nil
        => write!(f, "nil"),
    }
  }
}
```

Recognize that `Cons` is addressed by `List::Cons`.

Contrived error type

```
enum Result {
  Okay(Digest),
  Error(String),
}
```

Contrived error type

```
enum Result {
  Okay(Digest),
  Error(String),
}

fn generate_digest() -> Result {
  Result::Okay([42u8; 32])
}
```

Contrived error type

```
enum Result {
  Okay(Digest),
  Error(String),
}

fn generate_digest() -> Result {
  Result::Okay([42u8; 32])
}

fn main() {
  match generate_digest() {
    Result::Okay(d) => {
      for byte in d.iter() { print!("{:02X}", byte); }
      println!("");
    },
    Result::Error(msg) => eprintln!("error: {}", msg),
  }
}
```

Error handling in rust

`std::result::Result<T, E>`

- `Ok(T)`
- `Err(E)`

No exceptions, no error codes.

Error handling in rust

`std::result::Result<T, E>`

- `Ok(T)`
- `Err(E)`

No exceptions, no error codes.

`std::option::Option<T>`

- `None`
- `Some(T)`

If we fetch one element from a container data structure,
we get *some* value or *none*.

Error handling in rust

```
// Example for Result
match File::open("foo.txt") {
  Ok(fd) => { /* ... */ },
  Err(e) => panic!(e),
}

// Example for Some
let fetched = Some(value);
fetched.unwrap();  // return Some value or panic
fetched.unwrap_or(default_value);  // … or default
```

You usually implement error types (SyntaxError, InvalidArgError, …) on your own in your library.

Error handling in rust

```rust
// Result API (excerpt)
fn is_ok(&self) -> bool;
fn is_err(&self) -> bool;
fn ok(self) -> Option<T>;
fn err(self) -> Option<E>;
fn and_then<U, F>(self, op: F) -> Result<U, E>;

// Option API (excerpt)
fn is_some(&self) -> bool;
fn is_none(&self) -> bool;
fn unwrap(self) -> T;
fn unwrap_or(self, default: T) -> T;
fn ok_or<E>(self, err: E) -> Result<T, E>;
```

A unique error handling operator

Question mark operator

The question mark operator exits early in case of `Err` or returns the value otherwise.

```
fn compile(src: &str) -> Result<(), Error> {
  let tokens = tokenize(&src)?;
  let ast = parse(&tokens)?;
  // ...
  Ok(())
}
```

Return type of function must be a corresponding `Result`.

Question mark operator rewritten

It is can be rewritten with a match expression:

```rust
fn compile(src: &str) -> Result<(), Error> {
  let tokens = match tokenize(&src) {
    Err(E) => return Err(E),
    Ok(ts) => ts,
  };
  let ast = parse(&tokens)?;
  // ...
  Ok(())
}
```

Macros

- Three kinds of macros
  1. function-like macros (`println!("hi")`)
  2. derive macros (`derive(Debug)`)
  3. attribute-like macros (`cfg(target_arch = "x86")`)

Important differences from C:

- They operate on tokens, not the lexical level
- Macro hygiene (variables are not visible outside)

Macros

```
macro_rules! shake {
  (update $base:ident with $($elem:expr, )*)
    => { $( $base.update($elem); )* };
}
```

Input:

```
  shake!(update h with &data, &[b' '], &data2,);
```

Output:

```
  h.update(&data);
  h.update(&[b' ']);
  h.update(&data2);
```

The typestate pattern

Idea: Encode the state in the type

Example:

- `fopen` returns `FileOpened`
- `fwrite` returns `FileNonEmpty`
- `fclose` returns `FileClosed`

More details, for example, in The Typestate Pattern in Rust (blog post).

Why? Can increase security.

> *Both from a design point of view as from an implementation perspective the entire scope can be considered of exceptionally high standard. Using the type system to* ***statically encode properties such as the TLS state transition function*** *is one just one example of great defense-in-depth design decisions.*
>
> —*rustls formal audit report*

Memory safety

A program execution is memory safe if the following things do not occur:

- access errors
  - buffer overflow/over-read
  - invalid pointer
  - race condition
  - use after free
- uninitialized variables
  - null pointer access
  - uninitialized pointer access
- memory leaks
  - stack/heap overflow
  - invalid free
  - unwanted aliasing

References

Rules:

- one or more *shared* references (`&T`) to a resource
- exactly one *mutable* reference (`&mut T`)
- either or, not both! ("aliasing xor mutation")

Benefits of reference limitations for memory safety:

- one writer XOR n readers in concurrent context
- prevents data races

C++ uses the notion of RAII:

```
void WriteToFile(const std::string& message) {
  static std::mutex mutex;
  std::lock_guard<std::mutex> lock(mutex);
  std::ofstream file("example.txt");
  if (!file.is_open()) {
    throw std::runtime_error("unable to open file");
  }
  file << message << std::endl;
}
```

Ownership

- Each value in Rust has a variable that's called its *owner*
- There can only be one owner at a time
- Ownership can *move* from one variable to another
- When the owner goes out of scope, the value will be "dropped"

Ownership example

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(mut s: Stats) {
  s.score += 1;
}

fn main() {
  let a = Stats { score: 8 };
  sub(a);

}
```

Ownership example

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(mut s: Stats) {
  s.score += 1;
}

fn main() {
  let a = Stats { score: 8 };
  sub(a);
  println!("{:?}", a);
}
```

Ownership example

```
error[E0382]: borrow of moved value: `a`
  --> src/main.rs:10:20
   |
8  |     let a = Stats { score: 8 };
   |         - move occurs because `a` has type `Stats`,
   |           which does not implement the `Copy` trait
9  |     sub(a);
   |         - value moved here
10 |     println!("{}", a);
   |                    ^
   |         value borrowed here after move
```

Ownership example

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(mut s: Stats) {
  // owner of Stats instance = `s`
  s.score += 1;
  // `s` goes out of scope → Stats instance is dropped
}

fn main() {
  let a = Stats { score: 8 };
  // owner of Stats instance = `a`
  sub(a); // move Stats instance: `a` → `s`
  println!("{:?}", a); // has been dropped already → error
}
```

Ownership example

Solutions:

- Use `#[derive(Debug,Copy,Clone)]`. Then `sub` uses copied instance. Results in `Stats { score: 8 }`
- Return `Stats` instance and assign it again in `main`.
- Use references (*borrowing* ownership)

Benefits of ownership for memory safety:

- we can pin-point when a variable is dropped (across threads!)

Ownership example with borrowing

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(s: &mut Stats) {
  s.score += 1;
}

fn main() {
  let mut a = Stats { score: 8 };
  // ownership of `a` is borrowed to `s`
  sub(&mut a);
  // ownership of `s` is returned back to `a`
  println!("{:?}", a);
}
```

Advanced types

Box<T> value on the heap (implies ownership)

Rc<T> reference-counted value on the heap (weakly & strongly counted)

Arc<T> Rc with atomic counters

Cell<T> aliasing XOR mutation at runtime for copyable types (creates Ref<T> and RefMut<T>)

RefCell<T> same for non-copyable types

Cheatsheet: Memory layout of types by Raph Levien

unsafe

```
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn rdtscp() -> (u64, u32) {
  let (mut eax, mut ecx, mut edx) = (0, 0, 0);
  {
    unsafe {
      asm!(
        "rdtscp",
        lateout("eax") eax,
        lateout("ecx") ecx,
        lateout("edx") edx,
        options(nomem, nostack)
      );
    }
} }
```

Blog article: Intel's RDTSC instruction with rust's RFC-2873 asm! macro

Superpowers:

1. Dereference a raw pointer (`const *`)
2. Call an `unsafe` function or method
3. Access or modify a mutable static variable
4. Implement an `unsafe` trait
5. Access fields of `unions`

Abusing unsafe

```
fn get_mutable_ref(val: &u32) -> &mut u32 {
  let ptr: *const u32 = val;
  let ptr_mut: *mut u32 = ptr as *mut u32;
  let ref_mut: &mut u32 = unsafe { &mut *ptr_mut };
  ref_mut
}
fn demo_two_mutable_refs() {
  let v: u32 = 42;
  let ref1: &mut u32 = get_mutable_ref(&v);
  let ref2: &mut u32 = get_mutable_ref(&v);

  *ref1 = 13;
  assert_eq!(*ref2, 13);
  *ref2 = 7;
  assert_eq!(*ref1, 7);
}
```

UB in rust

- Not all bugs can be caught with the type system
- A type system needs to be relaxed to be pragmatic
- A type system needs to be strict to be able to reason about it

Does undefined behavior (UB) exist in rust? Yes.

- See Behavior considered undefined for a non-exhaustive list
- Corner cases are still subject to academic debate

Overflow snippet

The following snippet can trigger an overflow. Where?

```
char buffer[128];
int bytesToCopy = packet.length;
if (bytesToCopy < 128) {
    strncpy(buffer, packet.data, bytesToCopy);
}

```

Example via CS 110L, Ryan Eberhardt and Armin Namavari

Overflow snippet

The following snippet can trigger an overflow. Where?

```
char buffer[128];
int bytesToCopy = packet.length;
if (bytesToCopy < 128) {
    strncpy(buffer, packet.data, bytesToCopy);
}

```

Example via CS 110L, Ryan Eberhardt and Armin Namavari

- Proper bounds check (yay!)
- `strncpy`, not `strcpy` (yay!)

Overflow snippet solved

The issue:

1. As declared, `bytesToCopy` is an `int`
2. Third argument of `strncpy` is a `size_t`
3. `bytesToCopy < 128` is true if `bytesToCopy` is negative
4. `bytesToCopy` is cast to an unsigned type and becomes huge

How is this prevented in rust?

- Types contain length (`String` is `Vec<u8>`, a `Vec` carries a `len`)
- No implicit casts (explicit casts via `as`)
- Bounds checks per default

Conclusion

Preventing vulnerabilities

off-by-one loops	`for elem in coll.iter() { /* … */ }`
buffer overflow	Checks happen. `Option<Item>`
integer overflow	*Debug mode?* Panic. *Release mode?* Bug with mod 2^n. *mod 2^n intended?* Use `(255u8).wrapping_add(1)`
type confusion	Explicit casts, From/Into/TryFrom/TryInto traits
use-after-free/double free	ownership model
format string exploits	C#-style format string, not C-style
TOCTTOU bugs	inevitable. Use AFFNP. Use `Mutex<Data>`.
uninitialized memory read	all memory is initialized
overlapping memory reads	mutation XOR aliasing
macro confusion	macros work on token-level

Academic rust

- RustBelt: academic project for formal verification of the Rust compiler
- Selected papers:
    - "RustBelt: Securing the Foundations of the Rust Programming Language" (paper)
    - "Stacked Borrows: An Aliasing Model for Rust" (paper)
    - "GhostCell: Separating Permissions from Data in Rust" (paper)
- "How Usable Are Rust Cryptography APIs?" (paper)
- "Is Rust Used Safely by Software Developers?" (paper)
- "Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs" (paper)

Rust at university

University courses on Rust:

- Rust course by Lukas Kalbertodt [DE]
- CS196 at Illinois
- CS110L at Stanford: Safety in Systems Programming

I mostly used the rust book.

- Learning Rust via Advent of Code
- Small exercises to get you used to reading and writing Rust code
- Rust by example
- Rust official doc
- stdlib
- rustlings
- Idiomatic rust
- A half hour to learn rust

Tools

clippy detects common mistakes and unidiomatic code

rustfmt allows you to reformat/normalize rust source code

There are many UNIX utilities rewritten in rust (xsv, ripgrep, etc.)

Thank you! Q/A?