Rust: Towards Better Code Security

GDR Sécurité / GT SSLR

Pierre Chifflier

pierre.chifflier@ssi.gouv.fr

Agence Nationale de la Sécurité
des Systèmes d'Information

27 novembre 2019

Introduction

Who

Who

- Pierre Chifflier
- Head of the Detection Research lab (LED) at ANSSI
- Security, ML, compilers and languages
- Rust evangelist (parse all the things!)

Outline

- Rust Language Properties
- The Rust Ecosystem
- Foreign Function Interface (FFI)
- Feedback: Suricata

This is not a Rust tutorial. For learning resources, see Rust by Example[1] or The Rust Book[2]

[1] Rust by Example. https://doc.rust-lang.org/rust-by-example/.
[2] The Rust Programming Language. https://doc.rust-lang.org/book/.

Rust Language Properties

The Beginning

Personal (and maybe unpopular) opinion:

To create a secure program in C you need an almost perfect developer, aware of all language/compiler gotchas, undefined behaviors, etc.

To create a formal proof, you need an expert in formal methods. Usually lots of efforts even for small applications, and very far from implementation.

How to reach **other** developers?

Overview

From the official website (http://rust-lang.org):

- Rust is a system programming language barely on hardware.
- No Runtime requirement
- Automatic yet deterministic memory allocation/destruction
- Guarantees memory safety

History

- First developed to address memory leakage and corruption bugs in Firefox
- First stable release in 2015
- Now used in many major projects
  - Firefox, Suricata, DropBox, …
- And being evaluated for others
  - Microsoft, Linux Kernel, …

General Properties

- Low-level
- Performance, similar to C
- Zero-cost abstraction
- Low overhead
- Strict Type checking
- Ownership, borrowing and lifetimes concepts
- Combines a static analyzer and a compiler

- But at a (cognitive) cost for developers

What is *not* in Rust

- No GC
  - Precise memory control
  - No latency
- No Runtime
  - Runs fast
- No exceptions
  - More predictable control path

This makes Rust usable for embedded systems, for ex.

Rustc is based on LLVM

The main compiler is `rustc`

- Intermediate IRs: HIR, MIR
- Compiles to LLVM IR
- Uses `lld` by default (LTO!)
- Lots of optimizations (and inlining)

Consequence: usual C tools (`gdb`, `valgrind`, `perf`, etc.) all work!

Rust Types

- Primitives types
  - `u8, i8, u16, usize,`…
  - `char` (4-byte unicode)
  - Pointers and references (cannot be null)
  - Specify sign and size
  - Prevents bugs due to **unexpected promotion/coercion/rounding**
- Separate `bool` type
  - No automatic conversion from/to integer
- Enums, Structs, Generic Types
- Strict separation of bytes and strings (only valid unicode)
- Strict type checking
- Immutable by default

Arrays

- Arrays `[T; N]` are stored with their length
- Fixed-sized arrays and variable-sized arrays
- Boths compile-time and runtime checks on access using `[]`
- Program is killed (`panic`) on violations

```
thread 'main' panicked at 'index out of bounds:
the len is 3 but the index is 4', src/main.rs:5:13
note: run with `RUST_BACKTRACE=1` environment variable
to display a backtrace.
```

Bounds Checking

- Adds overhead for every access
- Using iterators is strongly advised
- Compiler can sometimes remove extra checks, for ex:
  - When able to infer size
  - Or, on redundant tests
- Unsafe direct access is possible using `get_unchecked`

Mutability

- Variables *must* be initialized before use
- By default, variables are immutable
  - Checked by compiler
- The `mut` keyword is used to declare a mutable variable

```rust
let a: u8 = 0;
a = 1;
```

```
2 |     let a: u8 = 0;
  |         -
  |         |
  |         first assignment to ‘a‘
  |         help: make this binding mutable: ‘mut a‘
3 |     a = 1;
  |     ^^^^^ cannot assign twice to immutable variable
```

Type Conversions

- No aliasing
- Casts are allowed (between compatible types)
  - Using the `from` method

```rust
let a = u8::from(256u32);
```

  - Will refuse to build if types are not compatible

```
error[E0277]: the trait bound ‘u8: std::convert::From<u32>‘
is not satisfied
 --> src/main.rs:2:13
  |
2 |     let a = u8::from(256u32);
```

Type Conversions (2)

- Lossy casts using the `as` keyword

```rust
let a = 256 as u8;
```

- Only available for primitive types
- Compiler still checks what it can

```
error: literal out of range for ‘u8‘
 --> src/main.rs:2:13
  |
2 |     let a = 256 as u8;
  |             ^^^
```

Integer Overflows/Underflows

- Overflows/Underflows can be detected

```rust
let mut a: u8 = 255;
a = a + 1;
```

```
thread 'main' panicked at 'attempt to add with overflow'
```

- By default, only debug mode
- Or using explicit methods (e.g `checked_add`, `overflowing_add`, `wrapping_add`)
- Often mistaken (believed to be undefined[3])

```rust
match a.checked_add(1) {
  Some(result) => result,
  None { return Err("overflow") }
}
```

[3] Myths and Legends about Integer Overflow in Rust. http://huonw.github.io/blog/2016/04/myths-and-legends-about-integer-overflow-in-rust/.

Traits

Traits describe functionalities a type must provide

- Similar to *interfaces* in OOP
- Used to constrain types in generic functions
- Also used to allow/forbid core functions
  - `Clone, Copy, Eq, PartialEq, ...`
  - Prevents **type/semantic errors** (e.g copying a type which should not)

```
5 | let d = Dummy{ a:0, b:0 };
  |       - move occurs because 'd' has type 'Dummy',
          which does not implement the 'Copy' trait
6 | f(d);
  |     - value moved here
7 | let x = d.a;
  |           ^^^ value used here after move
```

Rust's Borrowing and Ownership

Compiler enforced:

- Every resource has a *unique* owner
- Other can borrow (*i.e* create an alias) with restrictions
- Owner cannot change or delete its resource while it is borrowed
- When the owner goes out of *scope*, the value is dropped

⇒ No runtime

⇒ Memory safe

⇒ Thread safe

Borrowing restrictions

The 4 rules of borrowing:

- You cannot borrow a *mutable* reference from an *immutable* object
- You cannot borrow *more than one mutable reference*
  - You can borrow multiple immutable references
- A *mutable* and an *immutable* reference cannot exist simultaneously
- The lifetime of a borrowed reference must end before the lifetime from the owner object

These rules prevent:

- **Side-effects** (esp. when calling functions)
- **Race conditions**
- **Use-after-free**

Lifetimes

- The Lifetime is the length of time a variable is usable
  - Checked by the compiler
  - Infered when possible, but often has to be explicit specified
- Lifetimes can be *anonymous* or *named*
- Allocation and destruction are inserted by compiler
  - No runtime (except allocation/destruction)
- Usually similar to the variable *scope*
  - Rust 1.36 introduced Non-Lexical Lifetimes (NLL)

```
{
    let o = f();        // Introduce scoped value: 'o'.
    ...
}                       // 'o' goes out of scope and is dropped.
```

Lifetime and References

- *Lifetimes* prevents dangling pointers/references

```
let r;                  // Introduce reference: ‘r‘.
{
    let i = 1;          // Introduce scoped value: ‘i‘.
    r = &i;             // Store reference of ‘i‘ in ‘r‘.
}                       // ‘i‘ goes out of scope and is dropped.

println!("{}", r);      // ‘r‘ still refers to ‘i‘.
```

```
5 |     r = &i;          // Store reference of ‘i‘ in ‘r‘.
  |     ^^^^^^ borrowed value does not live long enough
6 | }                    // ‘i‘ goes out of scope and is dropped.
  | - ‘i‘ dropped here while still borrowed
7 |
8 | println!("{}", r);  // ‘r‘ still refers to ‘i‘.
  |                - borrow later used here
```

Lifetime and References (2)

- *Lifetimes* also indicates (polymorphic) constraints between objects

```
struct UserInfo<'a> {
    name: &'a str
}
```

- The `'a` is the *name* of the lifetime
- This tells the compiler that `name` cannot be freed before `UserInfo`
  - Each instance of `UserInfo` will have its own lifetime
  - This prevents **dangling pointers** and **memory leaks**
- Objects can have multiple lifetime declarations (adding constraints)

```
struct UserInfo2<'a, 'b> {
    name: &'a str,
    address: &'b str
}
```

Ownership and Lifetimes

- Assignment changes ownership
  - For ex. function calls

```rust
struct Dummy{ a: i32, b: i32 }

fn take(arg: Dummy) { }

fn foo() {
    let mut res = Dummy {a: 0, b: 0};
    take(res); // res is moved here
    println!("res.a = {}", res.a); // COMPILE ERROR
}
```

- Ownership is moved from `res` to `arg`
- Additionally, `arg` is freed at end of function
- This is required for **thread safety**

Concurrency and Thread Safety

```
struct Dummy{ a: i32, b: i32 }

fn foo() {
    let mut res = Dummy {a: 0, b: 0};
    std::thread::spawn(move || { // Spawn a new thread
        let borrower = &mut res; // Mutably borrow res
        borrower.a += 1;
    });
    res.a += 1; // Error: res is borrowed
}
```

- Borrowing and ownership are the foundations of **thread safety**
- Some other restrictions apply
  - Moved items must be `Send + Sync`
  - Known non-thread-safe items can be marked `!Send`

The `unsafe` keyword

- Some operations are forbidden, except in a function or block marked `unsafe`
  - Foreign Function Calls (e.g `libc` calls)
  - Assembly
  - Raw pointer dereference
- This allows violating some security properties
  - But not *all* of them (e.g types and lifetimes are checked, etc.)
- Better code auditability
- Can be forbidden using `#![forbid(unsafe_code)]`

```rust
fn say_hello() {
  let msg = b"Hello, world!\n";
  unsafe{
    write(1, &msg[0], msg.len());
  }
}
```

Common Problems (1)

Rust is evolving fast

- Versions in Linux distributions are often outdated
  - `rustup` is often mandatory
- Some features are only in the *nightly* version

Most tools require Internet access

- Even for simple operations (creating a project, building it)
- Having a mirror is required for offline development

Common Problems (2)

Hidden calls to `panic`

- Many functions can hide calls to `panic`
  - Many published libraries
  - Even from `std`, for ex `Duration::Add`
  - Some core operators like `[]`
- Ensuring code cannot panic is very hard

Checking for `unsafe` code

- It can be prevented in *your* crate[4]
- But is harder to check in dependencies

[4] A crate is a code package, for ex. a library or binary

Lack of formal verification tools

- Rust was made from ideas of many languages
  - It was not designed from a global grammar
- Formal reasoning/verification tools do not yet exist
  - They will require models for complex properties (lifetimes, borrowing, ownership)
  - See Oxide[5], Rustbelt[6] and Prusti[7]

[5] Aaron Weiss et al. Oxide: The Essence of Rust. 2019. arXiv: 1903.00982 [cs.PL].

[6] Ralf Jung et al. "RustBelt: securing the foundations of the Rust programming language." In: 2.POPL (Jan. 2018), 66:1–66:?? ISSN: 2475-1421. DOI: https://doi.org/10.1145/3158154.

[7] A static verifier for Rust, based on the Viper verification infrastructure. http://prusti.ethz.ch.

Summary: Key Security Properties

Property	Threat Covered
Bounds Checking	OOB access
Checked Arithmetic	Integer underflows/overflows
Mandatory Initialization	Use of uninitialized memory
Format String Types	Format String errors
Lifetimes	Memory Leaks, Use-After-Free
Borrowing,Ownership	Memory errors
Ownership	Data races
`unsafe`[8]	Unintended dangerous operations

[8] `unsafe` can break all of the above properties!

The Rust Ecosystem

Cargo (1)

`cargo` is the main Rust tool

- Handles all tasks: building, checking dependencies, running tests, publishing crates, …
- Based on subtools
- Extensible

⚠ Assumes an internet connection

Cargo (2)

`cargo` encourages good practises[9]

- Unit tests (`cargo test`)
  - Can be inline (unit tests) or in separate tree (integration tests)
  - Can also be in documentation
- Documentation (`cargo doc`)
  - Inline documentation
  - `pragma` can require doc for exported functions
- Benchmarks (`cargo bench`)
  - Performance measure

These are part of the core tools

[9] Good practises are not security properties, but contributes to security and helps finding regressions/breaking changes

`crates.io`

Main crates repository: https://crates.io

- Similar to `opam`, `pip` and other repositories
- Anybody can upload a crate
  - No review process
  - No validation (e.g License compatibility)
- Quality/maintenance may vary

Clippy

- Lints/Common Mistakes/Idiomatic checks in categories:
  - Correctness
  - Style
  - Complexity
  - Performances
  - …
- Easily integrated into QA
- Can be extended with custom checks

Other Tools of Interest (for security)

- `audit`: check dependencies for crates with security vulnerabilities
- `crev`: collaborative code review system
- `fuzz`: integration with `libFuzzer`
- `geiger`: find usages of unsafe Rust code
  - Including in dependencies
- `miri`: find certain undefined behaviors
- `outdated`: find out of date dependencies

These tools are not part of the core distribution

cargo fuzz (1)

Fuzzing Rust Code

- Write a fuzzer (call function)

```
#[export_name="rust_fuzzer_test_input"]
pub extern fn go(data: &[u8]) {
    let _ = der_parser::parse_der(data);
}
```

- Call libFuzzer

```
$ cargo +nightly fuzz run --jobs 24 fuzzer_parse_der
...
[2] #1188      NEW    cov: 1106 ft: 6985 corp: 576/91Kb lim: 42560
  exec/s: 1188 rss: 66Mb L: 15/3674 MS: 4
  CopyPart-EraseBytes-ChangeByte-ChangeBit-
```

- Uses a corpus by default

cargo fuzz (2)

Can be combined with coverage

- For ex. with `kcov`

```
$ kcov --include-path .,.. ./cov \
  ./target/debug/fuzzer_parse_der corpus/fuzzer_parse_der/*
```

- Shameless citation of author's blog[10]

Filename	Coverage percent	C
[...]/RUST/der-parser/src/lib.rs	0.0%	
[...]/RUST/der-parser/src/der/parser.rs	98.6%	
[...]/RUST/der-parser/src/ber/parser.rs	99.1%	
[...]/RUST/der-parser/src/ber/ber.rs	100.0%	
[...]/RUST/der-parser/src/oid.rs	100.0%	
[...]/RUST/der-parser/fuzz/fuzzers/fuzzer_parse_der.rs	100.0%	

[10] Fuzzing Rust code: cargo-fuzz and honggfuzz. `https://www.wzdftpd.net/blog/rust-fuzzers.html`.

Foreign Function Interface (FFI)

Interoperability

Foreign Function Interface

- Rust is designed to be interoperable with other languages
  - Calling functions
  - Accessing foreign objects
  - Exposing objects/functions
- All of this requires `unsafe` code

Goals

- Wrap C libraries and create safe abstractions
- Create "safe zones" inside programs
  - Perform dangerous operations safely
  - Exposed as C modules
- Use libraries
- Access hardware

Some General Points on FFI

- Rust is based on LLVM
  - This simplifies interoperability
- However, Rust has its own memory model
- Extra care must be take to
  - Access or expose data properly
  - Avoid making the memory model angry
  - Handle lifetimes of foreign objects
  - Ensure a robust interface (e.g handling unwinding)

Data Layout

- Rust types use a specific representation
  - For simple types, layout can be predicted
  - Alignment and padding may differ from C
  - Layout can change with compiler versions
- Some types can use C representation `repr(C)`
  - Tells the compiler to use the exact C layout
  - Can be coupled with `bindgen` or `cbindgen` to generate headers
- Other representations exist (`transparent, packed, u16, ...`)
- Not all types have a defined C representation (e.g `enums`)

Function Calls

- Rust has its own ABI
  - Name mangling
  - Hash added for specialization/versioning
- Some functions can be marked `extern "C"`
- Input arguments are trusted by the compiler
  - Values must be verified
  - Type coercions must be applied
  - Lifetimes must be added (or removed) manually

C Types

`::std::ffi` and `::std::os::raw` contain FFI types

Rust	Wrapped C Type	C
`String`	`CString`	`char *`†
`&str`	`CStr`	`char *`†
`void`	`c_void`	`void *`
...	...	...

†Only if valid UTF-8, else mapped to `&[u8]`

FFI Good Practises

- Write **minimal** unsafe layer (or generate it)
  - Test input values
  - Build Rust objects
  - Call safe code
  - Extract result, convert it back to C
- Unwinding panics *must* be caught
- Use opaque types when possible
  - Memory from language *x* should (must) be freed in language *x*

The Rustonomicon

The Dark Arts of Unsafe Rust[11] book covers

- Safe/Unsafe calls, and how to create safe abstractions
- Types, memory representation and coercions
- Exception safety
- Uninitialized memory
- Concurrency
- ...

[11] Rustonomicon. https://doc.rust-lang.org/nomicon/.

Feedback: Suricata

Suricata

Suricata[12] is a Network Intrusion Detection system. It has to

- Parse **untrusted data**
- Containing **complex protocols**
- And apply **lots of detection rules**
- At **very high speed**

This is the **perfect** candidate!

[12] Suricata: Open Source IDS / IPS / NSM engine. `https://suricata-ids.org/`.

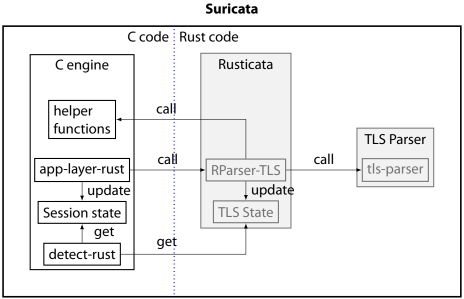
Codebase in 2016

- Open Source
- ~400 000 lines of C
- Many parsers
  - Low-level network layers (IP, TCP, …)
  - Application layers (HTTP, TLS, …)
- Heavily multithreaded

Hardening Suricata

Rusticata (shameless citation #2):

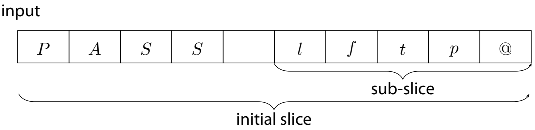
- Proof of concept code
- Presented at Suricon 2016[13]
- Integration of Rust into the detection engine



[13] Pierre Chifflier. Securing Security Tools. https://suricon.net/highlights-suricon-2016/. Suricon. 2016.

Rust Parsers

- Mostly based on Nom[14]
- Parser Combinators very easy to map in Rust
  - Descending parsing
  - Slices of decreasing length
  - Length tests everywhere




```
tag!("PASS") >> multispace1 >> rest
```

[14] Nom: Rust parser combinator framework. https://github.com/Geal/nom.

Architecture

- Code separation
  - Parsers (pure Rust)
  - Interface/helpers (FFI)

Suricata & Rust

- Rust support added in 4.0 (August 1, 2017)
- Not using Rusticata, but inspired from
  - Core team had to control tightly the implementation
- Shipped with new Rust parsers
  - SMB, NFS, NTP
- Rust support marked as experimental

Changes: Build System

- The Rust code is compiled to an archive file (`.a`)
  - Exposing a C ABI
  - Linked into the resulting binary
- Lack of runtime is a key advantage
- Rust not easily usable from `autotools+make`
  - Compiler could be called in `Makefile`,
  - But dependencies would have to be resolved manually
  - Choice: `cargo` is used from `autotools`

Changes: Distributing code

Difficulties: package manager *vs* distributing sources

- `cargo` uses internet
  - breaks offline builds
- `cargo` fetches dependencies for every build
  - breaks reproducible builds

Solution: distributing dependencies (vendoring, `cargo vendor`)

Changes: Linux Distributions

- Rust & `cargo` not shipped in Linux distros (or outdated)
  - Many features not usable in practice
  - Forced targeting a minimum version
  - With time, situation improved

Changes: Performances

- Benchmarks by Brad Woodberg in 2017 and 2019
- Rust overhead: between 5% and 10%
- May not be an entirely fair comparison ☺
  - More parsers and features when Rust is enabled
- Considered as acceptable by the core team

Rust & Suricata: 2 years later (2019)

- Rust support now mandatory
  - Especially for new parsers
- Many included (complex) parsers
  - SNMP, Kerberos, SIP, FTP, …
  - Several externally contributed
- 5.5% of total lines of code
- May replace complex parts in the future
  - For ex. the DER parser (X.509 certificates)

Team Feedback

- Overall: very good
- Macros: hard to understand
- Code review: less doubts and dangers
- Required some experience in the language
- Some parsers would not have been added if written in C

Problems not solved yet

- Lots of code duplication for C interface
- C unit tests *vs* Rust unit tests
- Doc generation: separate tools

Conclusion

Summary

- Modern Language (steep learning curve), good for security
- Both a Static Analyzer[15] [16] and a Compiler
- Enforces good practices and checks them
- Huge improvement over C

[15] It will yell at you until your code is acceptable

[16] Hard time for average C developers

Conclusion

Rust & Security

- Rust is a modern language
  - Built with security in mind
  - Based on new concepts
- Lacks some tools
  - But is evolving fast
- ANSSI Recommendations[17]

[17] ANSSI Recommendations for secure applications development with Rust. https://github.com/ANSSI-FR/rust-guide.