PROGRAMMING RULES TO DEVELOP SECURE APPLICATIONS WITH RUST

ANSSI GUIDELINES

ANSSI-PA-074-EN
06/09/2020

TARGETED AUDIENCE:

Developers Administrators IT security managers IT managers Users

Information

Warning

This document, written by the ANSSI, presents the **"Programming Rules to Develop Secure Applications with Rust"**. It is freely available at www.ssi.gouv.fr/en/. It is an original creation from the ANSSI and it is placed under the "Open Licence" published by the Etalab mission (www.etalab.gouv.fr). Consequently, its diffusion is unlimited and unrestricted.
This document is a courtesy translation of the initial French document **"Titre non défini"**, available at www.ssi.gouv.fr. In case of conflicts between these two documents, the latter is considered as the only reference.
These recommendations are provided as is and are related to threats known at the publication time. Considering the information systems diversity, the ANSSI cannot guarantee direct application of these recommendations on targeted information systems. Applying the following recommendations shall be, at first, validated by IT administrators and/or IT security managers.

Document changelog:

VERSION	DATE	CHANGELOG
1.0	06/09/2020	Initial version

Contents

1
Introduction

Rust is a multi-paradigm language with a focus on memory safety.

It aims to be system programming oriented, allowing fine-grained memory management without garbage collection but also without tedious and error-prone manual memory allocations and deallocations. It achieves this goal by means of its ownership system (mostly related to variable aliasing). At any point of a Rust program, the compiler tracks how many variables refer to a given data, and enforces a set of rules which enable automatic memory management, memory safety and data-race free programs.

The language also focuses on performance, with powerful compilation optimizations and language constructs that allow writing zero-cost abstraction code.

Moreover, the Rust language provides some high-level programming features. Thanks to higher-order functions, closures, iterators, etc., it allows to write program parts in the same vein as in functional programming languages. Besides, static typing discipline, type inference, and ad hoc polymorphism (in the form of traits) are other ways Rust provides to build libraries and programs in a safe manner.

Nevertheless, due to its versatility, the language possibly offers some constructions that, if not used properly, can introduce security problems, by making code misinterpreted by the programmer or a reviewer. In addition, as for every tool in the compilation or software verification field, the tools used to develop, compile and execute programs can expose certain features or configurations that, if misused, may lead to vulnerabilities.

Thus, the object of this document is to compile hints and recommendations to stay in a safe zone for secure applications development while taking advantage of the range of possibilities the Rust language can offer.

1.1 Target Audience

The guide intents to group recommendations that should be applied for application development with strong security level requirements. Anyway, it can be followed by everyone who wants to ensure that guarantees offered by the Rust platform are not invalidated due to unsafe, misleading or unclear feature usage.

It is not intended to be a course on how to write Rust programs, there are already plenty of good learning resources for this purpose (see for instance the Rust documentation main page). The purpose is rather to guide the programmer and inform him about some pitfalls he may encounter.

These recommendations form a complement to the good level of trust the Rust language already provides. That said, recalls are sometimes necessary for clarity, and the experienced Rust programmer may rely solely on highlighted inserts (*Rule*, *Recommendation*, *Warning*, etc.).

1.2 Contributions

This guide is written in a collaborative and open manner, via the GitHub platform (`https://github.com/ANSSI-FR/rust-guide`). All contributions for future versions are welcome, whether in the form of direct propositions (*pull requests*) or in the form of suggestions and discussions (*issues*).

1.3 Structure of the Document

This document considers separately different phases of a typical (and simplified) development process. Firstly, we provide some advices on how to take advantage of using tools of the Rust ecosystem for secure development. A following chapter focuses on precautions to take when choosing and using external libraries. Then, recommendations about the Rust language constructs are exposed. A summary of recommendations presented throughout the document is listed at the end of this guide.

2
Development environment

2.1 Rustup

Rustup is the Rust toolchain installer. Among other things, it enables switching between different flavors of the toolchain (stable, beta, nightly), managing additional components installation and keeping them up to date.

Warning

From a security perspective, `rustup` does perform all downloads over HTTPS, but does not yet validate signatures of downloads. Protection against downgrade attacks, certificate pinning, validation of signatures are still works in progress. In some cases, it may be preferable to opt for an alternative installation method listed in the *Install* section of the official Rust website.

2.1.1 Rust Editions

Several flavors, called *editions*, of the Rust language coexist.

The concept of editions has been introduced to clarify new features implementation and to make them incremental. A new edition will be produced every two or three years, as stated in the Edition Guide, but this doesn't mean that new features and improvements will only be shipped in a new edition.

Some editions bring new keywords and language constructs. Recommendations for secure applications development then remain closely linked to features of the language, that are used in such applications, rather than to Rust editions. In the rest of this guide, best effort will be made to highlight constructions and language features that are specific to a particular Rust edition.

Note

No specific edition is recommended, as long as users follow the recommendations related to the features offered by the edition that has been chosen.

2.1.2 Stable, nightly and beta toolchains

Orthogonally to editions that allow one to select a flavor (a set of features) of the Rust language, the Rust toolchain is provided in three different versions, called *release channels*:

- *nightly* releases are created once a day,

- *beta* releases are created every six weeks, from promoted *nightly* releases,
- *stable* releases are created every six weeks, from promoted *beta* releases.

When playing with different toolchains, it is important to check not only what the default toolchain is, but also if overrides are currently set for some directories.

Console

```
$ pwd
/tmp/foo
$ rustup toolchain list
stable-x86_64-unknown-linux-gnu (default)
beta-x86_64-unknown-linux-gnu
nightly-x86_64-unknown-linux-gnu
$ rustup override list
/tmp/foo                                        nightly-x86_64-unknown-linux-gnu
$
```

R1

RULE - Use a stable compilation toolchain

Development of a secure application must be done using a fully stable toolchain, for limiting potential compiler, runtime or tool bugs.

When using a specific `cargo` subcommand that requires a nightly component, it is preferable to run it by switching the toolchain only locally, instead of explicitly switching the complete toolchain. For example, to run the (nightly) latest `rustfmt`:

Console

```
$ rustup toolchain list
stable-x86_64-unknown-linux-gnu (default)
beta-x86_64-unknown-linux-gnu
nightly-x86_64-unknown-linux-gnu
$ rustup run nightly cargo fmt
$ # or
$ cargo +nightly fmt
$
```

2.2 Cargo

Once Rustup has set up the appropriate Rust toolchain, Cargo is available through the command line program `cargo`. Cargo is the Rust package manager. It has a fundamental role in most Rust development:

- It structures project by providing the project skeleton (`cargo new`),
- It compiles the project (`cargo build`),
- It generates the project's documentation (`cargo doc`),
- It runs tests (`cargo test`) and benchmarks (`cargo bench`),
- It manages and download dependencies,
- It makes packages distributable and publishes them on crates.io,

- It's also a front-end to run complementary tools such as those that are described below, in the form of sub-commands.

Warning

Like `rustup`, `cargo` does perform all downloads over HTTPS, but does not validate the registry index. Ongoing discussions occur on how to best protect and verify crates. For now, the security relies on the good security of the website crates.io and the GitHub hosted repository containing the registry index. In some cases, it may be preferable to opt for an alternative installation method for dependencies.

Cargo proposes many different commands and options to adapt the build process to your project needs, mainly through the manifest file `Cargo.toml`. For a complete presentation, see The Cargo Book.

During the development of a secure application, some of the options may require some attention. The `[profile.*]` sections allow configuring how the compiler is invoked. For example:

- the `debug-assertions` variable controls whether debug assertions are enabled,
- the `overflow-checks` variable controls whether overflows are checked for integer arithmetic.

Overriding the default options may cause bugs not being detected, even when using the debug profile that normally enables runtime checks (for example checks for integer overflows, see 4.3).

R2

RULE - Keep default values for critical variables in cargo profiles

The variables `debug-assertions` and `overflow-checks` must not be overridden in development profiles sections (`[profile.dev]` and `[profile.test]`).

Cargo proposes other ways to setup its configuration and change its behavior on a given system. This can be very useful, but it may also be difficult to know and remember at a given time all the options that are effectively used, and in particular passed to the compiler. At the end, this can affect the confidence and robustness of the build process. It is preferable to centralize compiler options and flags in the configuration file `Cargo.toml`. For the case of environment variable `RUSTC_WRAPPER`, for example, that may be used to generate part of code or to run external tools before Rust compilation, it is preferable to use the Cargo build scripts feature.

R3

RULE - Keep default values for compiler environment variables when running cargo

The environment variables `RUSTC`, `RUSTC_WRAPPER` and `RUSTFLAGS` must not be overriden when using Cargo to build the project.

2.2.1 Clippy

Clippy is a tool that provides and checks many lints (bugs, styling, performance issues, etc.). Since version 1.29, `clippy` can be used within the stable `rustup` environment. It is recommended to

install `clippy` as a component (`rustup component add clippy`) in the stable toolchain instead of installing it as a project dependency.

The tool comes with some lint categories regarding the kind of issues it aims to detect. The warnings should be re-checked by the programmer before committing the fix that is suggested by `clippy`, especially in the case of lints of the category `clippy::nursery` since those hints are still under development.

R4

RULE - Use linter regularly

A linter, such as `clippy`, must be used regularly during the development of a secure application.

2.2.2 Rustfmt

Rustfmt is a tool that formats your code according to style guidelines. The documentation of the tool states some limitations, among others partial support of macro declarations and uses. One should use the `--check` option that prints proposed changes, review these changes, and finally apply them if the code readability is not affected.

So, to launch it:

Console

```
$ cargo fmt -- --check
$ # review of the changes
$ cargo fmt
```

These guidelines can be customized to your needs by creating a `rustfmt.toml` or `.rustfmt.toml` file at the root of your project. It will be used to override the default settings, for instance:

Toml

```
# Set the maximum line width to 120
max_width = 120
# Maximum line length for single line if-else expressions
single_line_if_else_max_width = 40
```

For more information about the guidelines that `rustfmt` will check, have a look at the Rust Style Guide.

R5

RULE - Use Rust formatter (rustfmt)

The tool `rustfmt` can be used to ensure that the codebase respects style guidelines (as described in `rustfmt.toml` file), with `--check` option and manual review.

2.2.3 Rustfix

Included with Rust, since the end of 2018, Rustfix is a tool dedicated in fixing compiler warnings as well as easing transitions between editions.

Console

```
$ cargo fix
```

To prepare a Rust 2015 project to transition to Rust 2018, one can run:

Console

```
$ cargo fix --edition
```

Rustfix will either fix the code to be compatible with Rust 2018 or print a warning that explains the problem. This problem will have to be fixed manually. By running the command (and possibly fixing manually some issues) until there is no warning, one can ensure the code is compatible with both Rust 2015 and Rust 2018.

To switch definitely to Rust 2018, one may run:

Console

```
$ cargo fix --edition-idioms
```

Be advised that this tool provides few guarantees on the soundness of the proposed fixes. In particular mode, some corrections (such as some of those provided with the `--edition-idioms`) are known to break the compilation or change the program semantics in some case.

R6

RULE - Manually check automatic fixes

In a secure Rust development, any automatic fix (for instance, provided by `rustfix`) must be verified by the developer.

2.2.4 Others

There exist other useful tools or `cargo` subcommands for enforcing program security whether by searching for specific code patterns or by providing convenient commands for testing or fuzzing. They are discussed in the following chapters, according to their goals.

3
Libraries

In addition to the standard library, Rust provides an easy way to import other libraries in a project, thanks to `cargo`. The libraries, known as *crates* in the Rust ecosystem, are imported from the open-source components central repository crates.io.

It should be noticed that the quality (in terms of security, performances, readability, etc.) of the published crates is very variable. Moreover, their maintenance can be irregular or interrupted. The usage of each component from this repository should be justified, and the developer should validate the correct application of rules from the current guide in its code. Several tools can aid in that task.

3.1 Cargo-outdated

Cargo-outdated tool allows one to easily manage dependencies versions.

For a given crate, it lists current dependencies versions (using its `Cargo.toml`), and checks latest compatible version and also latest general version.

R7

RULE - Check for outdated dependencies versions (cargo-outdated)

The `cargo-outdated` tool must be used to check dependencies status. Then, each outdated dependency must be updated or the choice of the version must be justified.

3.2 Cargo-audit

Cargo-audit tool allows one to easily check for security vulnerabilities reported to the RustSec Advisory Database.

R8

RULE - Check for security vulnerabilities report on dependencies (cargo-audit)

The `cargo-audit` tool must be used to check for known vulnerabilities in dependencies.

4
Language generalities

4.1 Naming

As of now, the standard library is the de facto standard for naming things in the Rust world. However, an effort has been made to formalize it, first in RFC 430, then in the Rust API Guidelines.

The basic rule consists in using :

- `UpperCamelCase` for types, traits, enum variants,
- `snake_case` for functions, methods, macros, variables and modules,
- `SCREAMING_SNAKE_CASE` for statics and constants,
- `'lowercase` for lifetimes.

The Rust API Guidelines also prescribes more precise naming conventions for some particular constructions:

- (C-CONV) for conversion methods (`as_`, `to_`, `into_`),
- (C-GETTER) for getters,
- (C-ITER) for iterator-producing methods,
- (C-ITER-TY) for iterator types,
- (C-FEATURE) for feature naming,
- (C-WORD-ORDER) for word order consistency.

R9

RULE - Respect naming conventions

Development of a secure application must follow the naming conventions outlined in the Rust API Guidelines.

4.2 Unsafe code

The joint utilization of the type system and the ownership system aims to enforce safety regarding memory management in Rust's programs. So the language aims to avoid memory overflows, null or invalid pointer constructions, and data races. To perform risky actions such as system calls, type coercions, or direct manipulations of memory pointers, the language provides the `unsafe` keyword.

R10

RULE - Don't use unsafe blocks

For a secured development, the `unsafe` blocks must be avoided. Afterward, we list the only cases where `unsafe` may be used, provided that they come with a proper justification:

- The Foreign Function Interface (FFI) of Rust allows for describing functions whose implementations are written in C, using the `extern "C"` prefix. To use such a function, the `unsafe` keyword is required. “Safe” wrapper shall be defined to safely and seamlessly call C code.
- For embedded device programming, registers and various other resources are often accessed through a fixed memory address. In this case, `unsafe` blocks are required to initialize and dereference those particular pointers in Rust. In order to minimize the number of unsafe accesses in the code and to allow easier identification of them by a programmer, a proper abstraction (data structure or module) shall be provided.
- A function can be marked unsafe globally (by prefixing its declaration with the `unsafe` keyword) when it may exhibit unsafe behaviors based on its arguments, that are unavoidable. For instance, this happens when a function tries to dereference a pointer passed as an argument.

With the exception of these cases, `#[forbid(unsafe_code)]` must appear in `main.rs` to generate compilation errors if `unsafe` is used in the code base.

4.3 Integer overflows

Although some verification is performed by Rust regarding potential integer overflows, precautions should be taken when executing arithmetic operations on integers.

In particular, it should be noted that using debug or release compilation profile changes integer overflow behavior. In debug configuration, overflow cause the termination of the program (`panic`), whereas in the release configuration the computed value silently wraps around the maximum value that can be stored.

This last behavior can be made explicit by using the `Wrapping` generic type, or the `overflowing_<op>` and `wrapping_<op>` operations on integers (the `<op>` part being `add`, `mul`, `sub`, `shr`, etc.).

Rust

```
use std::num::Wrapping;

let x: u8 = 242;

println!("{}", x + 50);     // panics in debug, prints 36 in release.
println!("{}", x.overflowing_add(50).0);     // always prints 36.
println!("{}", x.wrapping_add(50));          // always prints 36.
println!("{}", Wrapping(x) + Wrapping(50));  // always prints 36.

// always panics:
let (res, c) = x.overflowing_add(50);
if c { panic!("custom error"); }
```

```
else { println!("{}", res); }
```

R11

RULE - Use appropriate arithmetic operations regarding potential overflows

When assuming that an arithmetic operation can produce an overflow, the specialized functions `overflowing_<op>`, `wrapping_<op>`, or the `Wrapping` type must be used.

4.4 Error handling

The `Result` type is the preferred way of handling functions that can fail. A `Result` object must be tested, and never ignored.

R12

RECO - Implement custom `Error` type wrapping all possible errors

A crate can implement its own `Error` type, wrapping all possible errors. It must be careful to make this type exception-safe (RFC 1236), and implement `Error + Send + Sync + 'static` as well as `Display`.

R13

RECO - Use the `?` operator and do not use the `try!` macro

The `?` operator should be used to improve readability of code. The `try!` macro should not be used.

Third-party crates may be used to facilitate error handling. Most of them (notably failure, snafu, thiserror) address the creation of new custom error types that implement the necessary traits and allow wrapping other errors.

Another approach (notably proposed in the anyhow crate) consists in an automatic wrapping of errors into a single universal error type. Such wrappers should not be used in libraries and complex systems because they do not allow developers to provide context to the wrapped error.

4.4.1 Panics

Explicit error handling (`Result`) should always be preferred instead of calling `panic`. The cause of the error should be available, and generic errors should be avoided.

Crates providing libraries should never use functions or instructions that can fail and cause the code to panic.

Common patterns that can cause panics are:

- using `unwrap` or `expect`,
- using `assert`,
- an unchecked access to an array,
- integer overflow (in debug mode),

- division by zero,
- large allocations,
- string formatting using `format!`.

R14

RULE - Don't use functions that can cause `panic!`

Functions or instructions that can cause the code to panic at runtime must not be used.

R15

RULE - Test properly array indexing or use the `get` method

Array indexing must be properly tested, or the `get` method should be used to return an `Option`.

4.4.2 FFI and panics

When calling Rust code from another language (for ex. C), the Rust code must be careful to never panic. Stack unwinding from Rust code into foreign code results in undefined behavior.

R16

RULE - Handle correctly `panic!` in FFI

Rust code called from FFI must either ensure the function cannot panic, or use `catch_unwind` or the `std::panic` module to ensure the rust code will not abort or return in an unstable state.

Note that `catch_unwind` will only catch unwinding panics, not those that abort the process.

5
Memory management

5.1 Forget and memory leaks

While the usual way for memory to be reclaimed is for a variable to go out of scope, Rust provides special functions to manually reclaim memory: `forget` and `drop` of the `std::mem` module (or `core::mem`). While `drop` simply triggers an early memory reclamation that calls associated destructors when needed, `forget` skips any call to the destructors.

Rust

```
let pair = ('*', 0xBADD_CAFEu32);
drop(pair); // here `forget` would be equivalent (no destructor to call)
```

Both functions are **memory safe** in Rust. However, `forget` will make any resource managed by the value unreachable and unclaimed.

Rust

```
let s = String::from("Hello");
forget(s); // Leak memory
```

In particular, using `forget` may result in not releasing critical resources leading to deadlocks or not erasing sensitive data from the memory. That is why, `forget` is **unsecure**.

R17

RULE - Do not use `forget`

In a secure Rust development, the `forget` function of `std::mem` (`core::mem`) must not be used.

R18

RECO - Use clippy lint to detect use of `forget`

The lint `mem_forget` of Clippy may be used to automatically detect any use of `forget`. To enforce the absence of `forget` in a crate, add the following line at the top of the root file (usually `src/lib.rs` or `src/main.rs`):

Rust

```
#![deny(clippy::mem_forget)]
```

The standard library includes other way to *forget* dropping values:

- `Box::leak` to leak a resource,
- `Box::into_raw` to exploit the value in some unsafe code, notably in FFI,
- `ManuallyDrop` (in `std::mem` or `core::mem`) to enforce manual release of some value.

Those alternatives may lead to the same security issue but they have the additional benefit of making their goal obvious.

R19

RULE - Do not leak memory

In a secure Rust development, the code must not leak memory or resource in particular via `Box::leak`.

`ManuallyDrop` and `Box::into_raw` shift the release responsibility from the compiler to the developer.

R20

RULE - Do release value wrapped in `ManuallyDrop`

In a secure Rust development, any value wrapped in `ManuallyDrop` must be unwrapped to allow for automatic release (`ManuallyDrop::into_inner`) or manually released (unsafe `ManuallyDrop::drop`).

R21

RULE - Always call `from_raw` on `into_raw`ed value

In a secure Rust development, any pointer created with a call to `into_raw` (or `into_raw_nonnull`) from one of the following types:

- `std::boxed::Box` (or `alloc::boxed::Box`),
- `std::rc::Rc` (or `alloc::rc::Rc`),
- `std::rc::Weak` (or `alloc::rc::Weak`),
- `std::sync::Arc` (or `alloc::sync::Arc`),
- `std::sync::Weak` (or `alloc::sync::Weak`),
- `std::ffi::CString`,
- `std::ffi::OsString`,

must eventually be transformed into a value with a call to the respective `from_raw` to allow for their reclamation.

Rust

```
let boxed = Box::new(String::from("Crab"));
let raw_ptr = unsafe { Box::into_raw(boxed) };
let _ = unsafe { Box::from_raw(raw_ptr) }; // will be freed
```

Note

In the case of `Box::into_raw`, manual cleanup is possible but a lot more complicated than re-boxing the raw pointer and should be avoided:

Rust

```
// Excerpt from the standard library documentation
use std::alloc::{dealloc, Layout};
use std::ptr;

let x = Box::new(String::from("Hello"));
let p = Box::into_raw(x);
unsafe {
    ptr::drop_in_place(p);
    dealloc(p as *mut u8, Layout::new::<String>());
}
```

Because the other types (`Rc` and `Arc`) are opaque and more complex, manual cleanup is not possible.

5.2 Uninitialized memory

By default, Rust forces all values to be initialized, preventing the use of uninitialized memory (except if using `std::mem::uninitialized` or `std::mem::MaybeUninit`).

R22

RULE - Do not use uninitialized memory

The `std::mem::uninitialized` function (deprecated 1.38) or the `std::mem::MaybeUninit` type (stabilized 1.36) must not be used, or explicitly justified when necessary.

The use of uninitialized memory may result in two distinct security issues:

- drop of uninitialized memory (also a memory safety issue),
- non-drop of initialized memory.

Note

`std::mem::MaybeUninit` is an improvement over `std::mem::uninitialized`. Indeed, it makes dropping uninitialized values a lot more difficult. However, it does not change the second issue: the non-drop of an initialized memory is as much likely. It is problematic, in particular when considering the use of `Drop` to erase sensitive memory.

5.3 Secure memory zeroing for sensitive information

Zeroing memory is useful for sensitive variables, especially if the Rust code is used through FFI.

R23

RULE - Zero out memory of sensitive data after use

Variables containing sensitive data must be zeroed out after use, using functions that will not be removed by the compiler optimizations, like `std::ptr::write_volatile` or the `zeroize` crate.

The following code shows how to define an integer type that will be set to 0 when freed, using the `Drop` trait:

Rust

```
/// Example: u32 newtype, set to 0 when freed
pub struct ZU32(pub u32);

impl Drop for ZU32 {
    fn drop(&mut self) {
        println!("zeroing memory");
        unsafe{ ::std::ptr::write_volatile(&mut self.0, 0) };
    }
}

{
    let i = ZU32(42);
    // ...
} // i is freed here
```

6
Type system

6.1 Standard library traits

6.1.1 `Drop` trait, the destructor

Types implement the trait `std::ops::Drop` to perform some operations when the memory associated with a value of this type is to be reclaimed. `Drop` is the Rust equivalent of a destructor in C++ or a finalizer in Java.

Dropping is done recursively from the outer value to the inner values. When a value goes out of scope (or is explicitly dropped with `std::mem::drop`), the value is dropped in two steps. The first step happens only if the type of this value implements `Drop`. It consists in calling the `drop` method on it. The second step consists in repeating the dropping process recursively on any field the value contains. Note that a `Drop` implementation is **only responsible for the outer value**.

First and foremost, implementing `Drop` should not be systematic. It is only needed if the type requires some destructor logic. In fact, `Drop` is typically used to release external resources (network connections, files, etc.) or to release memory (e.g. in smart pointers such as `Box` or `Rc`). As a result, `Drop` trait implementations are likely to contain `unsafe` code blocks as well as other security-critical operations.

R24

RECO - Justify `Drop` implementation

In a Rust secure development, the implementation of the `std::ops::Drop` trait should be justified, documented and peer-reviewed.

Second, Rust type system only ensures memory safety and, from the type system's standpoint, missing drops is allowed. In fact, several things may lead to missing drops, such as:

- a reference cycle (for instance, with `Rc` or `Arc`),
- an explicit call to `std::mem::forget` (or `core::mem::forget`) (see section 5.1),
- a panic in drop,
- program aborts (and panics when abort-on-panic is on).

And missing drops may lead to exposing sensitive data or to lock limited resources leading to unavailability issues.

R25

RULE - Do not panic in `Drop` implementation

In a Rust secure development, the implementation of the `std::ops::Drop` trait must not panic.

Beside panics, secure-critical drop should be protected.

R26

RULE - Do not allow cycles of reference-counted `Drop`

Value whose type implements `Drop` must not be embedded directly or indirectly in a cycle of reference-counted references.

R27

RECO - Do not rely only on `Drop` to ensure security

Ensuring security operations at the end of some treatment (such as key erasure at the end of a cryptographic encryption) should not rely only on the `Drop` trait implementation.

6.1.2 `Send` and `Sync` traits

The `Send` and `Sync` traits (defined in `std::marker` or `core::marker`) are marker traits used to ensure the safety of concurrency in Rust. When implemented correctly, they allow the Rust compiler to guarantee the absence of data races. Their semantics is as follows:

- A type is `Send` if it is safe to send (move) it to another thread.
- A type is `Sync` if it is safe to share a immutable reference to it with another thread.

Both traits are *unsafe traits*, i.e., the Rust compiler does not verify in any way that they are implemented correctly. The danger is real: an incorrect implementation may lead to **undefined behavior**.

Fortunately, in most cases, one does not need to implement it. In Rust, almost all primitive types are `Send` and `Sync`, and for most compound types the implementation is automatically provided by the Rust compiler. Notable exceptions are:

- Raw pointers are neither `Send` nor `Sync` because they offer no safety guards.
- `UnsafeCell` is not `Sync` (and as a result `Cell` and `RefCell` aren't either) because they offer interior mutability (mutably shared value).
- `Rc` is neither `Send` nor `Sync` because the reference counter is shared and unsynchronized.

Automatic implementation of `Send` (resp. `Sync`) occurs for a compound type (structure or enumeration) when all fields have `Send` types (resp. `Sync` types). Using an unstable feature (as of Rust 1.37.0), one can block the automatic implementation of those traits with a manual *negative implementation*:

Rust

```
#![feature(option_builtin_traits)]

struct SpecialType(u8);
impl !Send for SpecialType {}
impl !Sync for SpecialType {}
```

The negative implementation of `Send` or `Sync` are also used in the standard library for the exceptions, and are automatically implemented when appropriate. As a result, the generated documentation is always explicit: a type implements either `Send` or `!Send` (resp. `Sync` or `!Sync`).

As a stable alternative to negative implementation, one can use a `PhantomData` field:

Rust

```
struct SpecialType(u8, PhantomData<*const ()>);
```

R28

RECO - Justify `Send` and `Sync` implementation

In a Rust secure development, the manual implementation of the `Send` and `Sync` traits should be avoided and, if necessary, should be justified, documented and peer-reviewed.

6.1.3 Comparison traits (`PartialEq`, `Eq`, `PartialOrd`, `Ord`)

Comparisons (==, !=, <, <=, >, >=) in Rust relies on four standard traits available in `std::cmp` (or `core::cmp` for `no_std` compilation):

- `PartialEq<Rhs>` that defines a partial equivalence between objects of types `Self` and `Rhs`,
- `PartialOrd<Rhs>` that defines a partial order between objects of types `Self` and `Rhs`,
- `Eq` that defines a total equivalence between objects of the same type. It is only a marker trait that requires `PartialEq<Self>`!
- `Ord` that defines the total order between objects of the same type. It requires that `PartialOrd<Self>` is implemented.

As documented in the standard library, Rust assumes **a lot of invariants** about the implementations of those traits:

- For `PartialEq`
  - *Internal consistency*: `a.ne(b)` is equivalent to `!a.eq(b)`, i.e., `ne` is the strict inverse of `eq`. The default implementation of `ne` is precisely that.
  - *Symmetry*: `a.eq(b)` and `b.eq(a)`, are equivalent. From the developer's point of view, it means:
    - `PartialEq<B>` is implemented for type `A` (noted `A: PartialEq<B>`),

    - `PartialEq<A>` is implemented for type `B` (noted `B: PartialEq<A>`),
    - both implementations are consistent with each other.
  - *Transitivity*: `a.eq(b)` and `b.eq(c)` implies `a.eq(c)`. It means that:
    - `A: PartialEq<B>`,
    - `B: PartialEq<C>`,
    - `A: PartialEq<C>`,
    - the three implementations are consistent with each other (and their symmetric implementations).
- For `Eq`
  - `PartialEq<Self>` is implemented.
  - *Reflexivity*: `a.eq(a)`. This stands for `PartialEq<Self>` (`Eq` does not provide any method).
- For `PartialOrd`
  - *Equality consistency*: `a.eq(b)` is equivalent to `a.partial_cmp(b) == Some(std::ordering::Eq)`.
  - *Internal consistency*:
    - `a.lt(b)` iff `a.partial_cmp(b) == Some(std::ordering::Less)`,
    - `a.gt(b)` iff `a.partial_cmp(b) == Some(std::ordering::Greater)`,
    - `a.le(b)` iff `a.lt(b) || a.eq(b)`,
    - `a.ge(b)` iff `a.gt(b) || a.eq(b)`.

    Note that by only defining `partial_cmp`, the internal consistency is guaranteed by the default implementation of `lt`, `le`, `gt`, and `ge`.
  - *Antisymmetry*: `a.lt(b)` (respectively `a.gt(b)`) implies `b.gt(a)` (respectively, `b.lt(b)`). From the developer's standpoint, it also means:
    - `A: PartialOrd<B>`,
    - `B: PartialOrd<A>`,
    - both implementations are consistent with each other.
  - *Transitivity*: `a.lt(b)` and `b.lt(c)` implies `a.lt(c)` (also with `gt`, `le` and `ge`). It also means:
    - `A: PartialOrd<B>`,
    - `B: PartialOrd<C>`,
    - `A: PartialOrd<C>`,
    - the implementations are consistent with each other (and their symmetric).
- For `Ord`
  - `PartialOrd<Self>`
  - *Totality*: `a.partial_cmp(b) != None` always. In other words, exactly one of `a.eq(b)`, `a.lt(b)`, `a.gt(b)` is true.
  - *Consistency with `PartialOrd<Self>`*: `Some(a.cmp(b)) == a.partial_cmp(b)`.

The compiler do not check any of those requirements except for the type checking itself. However, comparisons are critical because they intervene both in liveness critical systems such as schedulers

and load balancers, and in optimized algorithms that may use `unsafe` blocks. In the first use, a bad ordering may lead to availability issues such as deadlocks. In the second use, it may lead to classical security issues linked to memory safety violations. That is again a factor in the practice of limiting the use of `unsafe` blocks.

R29

RULE - Respect the invariants of standard comparison traits

In a Rust secure development, the implementation of standard comparison traits must respect the invariants described in the documentation.

R30

RECO - Use the default method implementation of standard comparison traits

In a Rust secure development, the implementation of standard comparison traits should only define methods with no default implementation, so as to reduce the risk of violating the invariants associated with the traits.

There is a Clippy lint to check that `PartialEq::ne` is not defined in `PartialEq` implementations.

Rust comes with a standard way to automatically construct implementations of the comparison traits through the `#[derive(...)]` attribute:

- Derivation `PartialEq` implements `PartialEq<Self>` with a **structural equality** providing that each subtype is `PartialEq<Self>`.
- Derivation `Eq` implements the `Eq` marker trait providing that each subtype is `Eq`.
- Derivation `PartialOrd` implements `PartialOrd<Self>` as a **lexicographical order** providing that each subtype is `PartialOrd`.
- Derivation `Ord` implements `Ord` as a **lexicographical order** providing that each subtype is `Ord`.

For instance, the short following code shows how to compare two `T1`s easily. All the assertions hold.

Rust

```
#[derive(PartialEq, Eq, PartialOrd, Ord)]
struct T1 {
    a: u8, b: u8
}

assert!(&T1 { a: 0, b: 0 } == Box::new(T1 { a: 0, b: 0 }).as_ref());
assert!(T1 { a: 1, b: 0 } > T1 { a: 0, b: 0 });
assert!(T1 { a: 1, b: 1 } > T1 { a: 1, b: 0 });
```

Warning

Derivation of comparison traits for compound types depends on the **field order**, and not on their names.
First, it means that changing the order of declaration of two fields change the resulting lexicographical order. For instance, provided this second ordered type:

Rust

```
#[derive(PartialEq, Eq, PartialOrd, Ord)]
struct T2 {
   b: u8, a: u8
};
```

we have `T1 {a: 1, b: 0} > T1 {a: 0, b: 1}` but `T2 {a: 1, b: 0} < T2 {a: 0, b: 1}`.
Second, if one of the underlying comparison panics, the order may change the result due to the use of short-circuit logic in the automatic implementation.
For enums, the derived comparisons depends first on the **variant order** then on the field order.

Despite the ordering caveat, derived comparisons are a lot less error-prone than manual ones and makes code shorter and easier to maintain.

R31

RECO - Derive comparison traits when possible

In a secure Rust development, the implementation of standard comparison traits should be automatically derived with `#[derive(...)]` when structural equality and lexicographical comparison is needed. Any manual implementation of standard comparison traits should be documented and justified.

7
Foreign Function Interface (FFI)

The Rust approach to interfacing with other languages relies on a strong compatibility with C. However, this boundary is by its very nature **unsafe** (see Rust Book: Unsafe Rust).

Functions that are marked `extern` are made compatible with C code during compilation. They may be called from C code with any parameter values. The exact syntax is `extern "<ABI>"` where ABI is a calling convention and depends on the target platform. The default one is `C` which corresponds to a standard C calling convention on the target platform.

Rust

```
// export a C-compatible function
#[no_mangle]
unsafe extern "C" fn mylib_f(param: u32) -> i32 {
    if param == 0xCAFEBABE { 0 } else { -1 }
}
```

For the function `mylib_f` to be accessible with the same name, the function must also be annotated with the `#[no_mangle]` attribute.

Conversely, one can call C functions from Rust if they are declared in an `extern` block:

Rust

```
use std::os::raw::c_int;
// import an external function from libc
extern "C" {
    fn abs(args: c_int) -> c_int;
}

fn main() {
    let x = -1;
    println!("{} {}\n", x, unsafe { abs(x) });
}
```

Note

Any foreign function imported in Rust through an `extern` block is **automatically `unsafe`**. That is why, any call to a foreign function must be done from an `unsafe` context.

`extern` blocks may also contain foreign global variable declarations prefixed with the `static` keyword:

Rust

```
//! A direct way to access environment variables (on Unix).
//! Should not be used! Not thread safe, have a look at `std::env`!

extern {
    // Libc global variable
    #[link_name = "environ"]
    static libc_environ: *const *const std::os::raw::c_char;
}

fn main() {
    let mut next = unsafe { libc_environ };
    while !next.is_null() && !unsafe { *next }.is_null() {
        let env = unsafe { std::ffi::CStr::from_ptr(*next) }
            .to_str()
            .unwrap_or("<invalid>");
        println!("{}", env);
        next = unsafe { next.offset(1) };
    }
}
```

7.1 Typing

Typing is the way Rust ensures memory safety. When interfacing with other languages, which may not offer the same guarantee, the choice of types in the binding is essential to maintain the memory safety.

7.1.1 Data layout

Rust provides no short or long term guarantees with respect to how the data is laid out in the memory. The only way to make data compatible with a foreign language is through explicit use of a C-compatible data layout with the `repr` attribute (see Rust Reference: Type Layout). For instance, the following Rust types:

Rust

```
#[repr(C)]
struct Data {
    a: u32,
    b: u16,
    c: u64,
}
#[repr(C, packed)]
struct PackedData {
    a: u32,
    b: u16,
    c: u64,
}
```

are compatible with the following C types:

C

```
struct Data {
    uint32_t a;
    uint16_t b;
    uint64_t c;
```

```
};
__attribute__((packed))
struct PackedData {
    uint32_t a;
    uint16_t b;
    uint64_t c;
}
```

R32

RULE - Use only C-compatible types in FFI

In a secure Rust development, only C-compatible types must be used as parameter or return type of imported or exported functions and as types of imported or exported global variables.
The lone exception is types that are considered **opaque** on the foreign side.

The following types are considered C-compatible:

- integral or floating point primitive types,
- `repr(C)`-annotated `struct`,
- `repr(C)` or `repr(Int)`-annotated `enum` with at least one variant and only fieldless variants (where `Int` is an integral primitive type),
- pointers.

The following types are not C-compatible:

- Dynamically sized types,
- Trait objects,
- Enums with fields,
- Tuples (but `repr(C)` tuple structures are OK).

Some types are compatibles with some caveats:

- Zero-sized types, which is really zero sized (which is let unspecified in C and contradicts the C++ specification),
- `repr(C)`, `repr(C, Int)`, or `repr(Int)`-annotated enum with fields (see RFC 2195).

7.1.2 Type consistency

R33

RULE - Use consistent types at FFI boundaries

Types must be consistent on each side of the FFI boundary.
Although some details may be hidden on one side with respect to the other (typically to make a type opaque), types on both sides must have the same size and the same alignment requirement.

Concerning enums with fields in particular, the corresponding types in C (or C++) are not obvious, cf. RFC 2195.

Automated tools to generate bindings, such as rust-bindgen or cbindgen, may be of help in making types consistent between C and Rust.

R34

RECO - Use automatic binding generator tools

In a secure Rust development, automated generation tools should be used to generate bindings when possible and to maintain them continually.

Warning

For binding C/C++ to Rust, rust-bindgen is able to automatically generate the low-level binding. A high-level safe binding is highly recommended (see recommendation "Provide safe wrapping to foreign library"). Also some options of rust-bindgen may result in dangerous translations, in particular `rustified_enum`.

7.1.3 Platform-dependent types

When interfacing with a foreign language, like C or C++, it is often required to use platform-dependent types such as C's `int`, `long`, etc.

In addition to `c_void` in `std::ffi` (or `core::ffi`) for `void`, the standard library offers portable type aliases in `std:os::raw` (or `core::os::raw`):

- `c_char` for `char` (either `i8` or `u8`),
- `c_schar` for `signed char` (always `i8`),
- `c_uchar` for `unsigned char` (always `u8`),
- `c_short` for `short`,
- `c_ushort` for `unsigned short`,
- `c_int` for `int`,
- `c_uint` for `unsigned int`,
- `c_long` for `long`,
- `c_ulong` for `unsigned long`,
- `c_longlong` for `long long`,
- `c_ulonglong` for `unsigned long long`,
- `c_float` for `float` (always `f32`),
- `c_double` for `double` (always `f64`).

The libc crate offers more C compatible types that cover almost exhaustively the C standard library.

R35

RULE - Use portable aliases `c_*` when binding to platform-dependent types

In a secure Rust development, when interfacing with foreign code that uses platform-dependent types, such as C's `int` and `long`, Rust code must use portable type aliases,

such as provided by the standard library or the libc crate, rather than platform-specific types, except if the binding is automatically generated for each platform (see Note below).

Note

Automatic binding generation tools (e.g. cbindgen, rust-bindgen) are able to ensure type consistency on a specific platform. They should be used during the build process for each target to ensure that the generation is sound for the specific target platform.

7.1.4 Non-robust types: references, function pointers, enums

A *trap representation* of a particular type is a representation (pattern of bits) that respects the type's representation constraints (such as size and alignment) but does not represent a valid value of this type and leads to undefined behavior.

In simple terms, if a Rust variable is set to such an invalid value, anything can happen from a simple program crash to arbitrary code execution. When writing safe Rust, this cannot happen (except through a bug in the Rust compiler). However, when writing unsafe Rust and in particular in FFI, it is really easy.

In the following, **non-robust types** are types that have such trap representations (at least one). A lot of Rust types are non-robust, even among the C-compatible types:

- `bool` (1 byte, 256 representations, only 2 valid ones),
- references,
- function pointers,
- enums,
- floats (even if almost every language have the same understanding of what is a valid float),
- compound types that contain a field of a non-robust type.

On the other hand, integer types (`u*`/`i*`), packed compound types that contain no non-robust fields, for instance are *robust types*.

Non-robust types are a difficulty when interfacing two languages. It revolves into deciding **which language of the two is responsible in asserting the validity of boundary-crossing values** and how to do it.

R36

RULE - Do not use unchecked non-robust foreign values

In a secure Rust development, there must not be any use of *unchecked* foreign values of non-robust types.
In other words, either Rust translates robust types to non-robust types through explicit checking or the foreign side offers strong guarantees of the validity of the value.

R37

RECO - Check foreign values in Rust

In a secure Rust development, the validity checks of foreign values should be done in Rust when possible.

Those generic rules are to be adapted to a specific foreign language or for the associated risks. Concerning languages, C is particularly unfit to offer guarantees about validity. However, Rust is not the only language to offer strong guarantees. For instance, some C++ subset (without reinterpretation) allows developers to do lot of type checking. Because Rust natively separates the safe and unsafe segments, the recommendation is to always use Rust to check when possible. Concerning risks, the most dangerous types are references, function references, and enums, and are discussed below.

Warning

Rust's `bool` has been made equivalent to C99's `_Bool` (aliased as `bool` in `<stdbool.h>`) and C++'s `bool`. However, loading a value other than 0 and 1 as a `_Bool`/`bool` is an undefined behavior *on both sides*. Safe Rust ensures that. Standard-compliant C and C++ compilers ensure that no value but 0 and 1 can be *stored* in a `_Bool`/`bool` value but cannot guarantee the absence of an *incorrect reinterpretation* (e.g., union types, pointer cast). To detect such a bad reinterpretation, sanitizers such as LLVM's `-fsanitize=bool` may be used.

7.1.4.1 References and pointers

Although they are allowed by the Rust compiler, the use of Rust references in FFI may break Rust's memory safety. Because their "unsafety" is more explicit, pointers are preferred over Rust references when binding to another language.

On the one hand, reference types are very non-robust: they allow only pointers to valid memory objects. Any deviation leads to undefined behavior.

When binding to and from C, the problem is particularly severe because C has no references (in the sense of valid pointers) and the compiler does not offer any safety guarantee.

When binding with C++, Rust references may be bound to C++ references in practice even though the actual ABI of an `extern "C"` function in C++ with references is "implementation-defined". Also, the C++ code should be checked against pointer/reference confusion.

Rust references may be used reasonably with other C-compatible languages including C variants allowing for non-null type checking, e.g. Microsoft SAL annotated code.

On the other hand, Rust's *pointer types* may also lead to undefined behaviors but are more verifiable, mostly against `std/core::ptr::null()` (C's `(void*)0`) but also in some context against a known valid memory range (particularly in embedded systems or kernel-level programming). Another advantage of using Rust pointers in FFI is that any load of the pointed value is clearly marked inside an `unsafe` block or function.

R38

RECO - Do not use reference types but pointer types

In a secure Rust development, the Rust code should not use references types but pointer types.
Exceptions include:

- Rust references that are opaque in the foreign language and only manipulated from the Rust side,
- `Option`-wrapped references (see Note below),
- references bound to foreign safe references, e.g. from some augmented C variants or from C++ compiled in an environment where `extern "C"` references are encoded as pointers.

R39

RULE - Do not use unchecked foreign references

In a secure Rust development, every foreign references that is transmitted to Rust through FFI must be **checked on the foreign side** either automatically (for instance, by a compiler) or manually.
Exceptions include Rust references in an opaque wrapping that is created and manipulated only from the Rust side and `Option`-wrapped references (see Note below).

R40

RULE - Check foreign pointers

In a secure Rust development, any Rust code that dereferences a foreign pointer must check their validity beforehand. In particular, pointers must be checked to be non-null before any use.
Stronger approaches are advisable when possible. They includes checking pointers against known valid memory range or tagging (or signing) pointers (particularly applicable if the pointed value is only manipulated from Rust).

The following code a simple example of foreign pointer use in an exported Rust function:

Rust

```
/// Add in place
#[no_mangle]
pub unsafe extern fn add_in_place(a: *mut u32, b: u32) {
    // checks for nullity of `a`
    // and takes a mutable reference on it if it's non-null
    if let Some(a) = a.as_mut() {
        *a += b
    }
}
```

Note that the methods `as_ref` and `as_mut` (for mutable pointers) allows easy access to a reference while ensuring a null check in a very *Rusty* way. On the other side in C, it can be used as follows:

C

```c
#include <stdint.h>
#include <inttypes.h>

//! Add in place
void add_in_place(uint32_t *a, uint32_t b);

int main() {
    uint32_t x = 25;
    add_in_place(&x, 17);
    printf("%" PRIu32 " == 42", x);
    return 0;
}
```

Note

`Option<&T>` and `Option<&mut T>` for any `T: Sized` are allowable in FFI instead of pointers with explicit nullity checks. Due to the Rust guaranteed "nullable pointer optimization", a nullable pointer is acceptable on the C side. The C `NULL` is understood as `None` in Rust while a non-null pointer is encapsulated in `Some`. While quite ergonomic, this feature does not allow stronger validations such as memory range checking.

7.1.4.2 Function pointers

Function pointers that cross FFI boundaries may ultimately lead to arbitrary code execution and represents a real security risks.

R41

RULE - Mark function pointer types in FFI as `extern` and `unsafe`

In a secure Rust development, any function pointer types at the FFI boundary must be marked `extern` (possibly with the specific ABI) and `unsafe`.

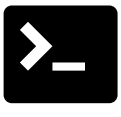
Function pointers in Rust are a lot more similar to references than they are to normal pointers. In particular, the validity of function pointers cannot be checked directly on the Rust side. However, Rust offers two alternative possibilities:

- use `Option`-wrapped function pointer and check against `null`:

Rust

```rust
#[no_mangle]
pub unsafe extern "C" fn repeat(
    start: u32, n: u32,
    f: Option<unsafe extern "C" fn(u32) -> u32>
) -> u32 {
    if let Some(f) = f {
        let mut value = start;
        for _ in 0..n {
            value = f(value);
        }
        value
    } else {
        start
    }
}
```

On the C side:



C

```
uint32_t repeat(uint32_t start, uint32_t n, uint32_t (*f)(uint32_t));
```

- use raw pointers with an `unsafe` transmutation to the function pointer type, allowing more powerful checks at the cost of ergonomics.

R42

RULE - Check foreign function pointers

In a secure Rust development, any foreign function pointer must be checked at the FFI boundary.

When binding with C or even C++, one cannot guarantee easily the validity of the function pointer. C++ functors are not C-compatible.

7.1.4.3 Enums

Usually the possible bit patterns of valid `enum` values are really small with respect to the number of possible bit patterns of the same size. Mishandling an `enum` value provided by a foreign code may lead to type confusion and have severe consequences on software security. Unfortunately, checking an `enum` value at the FFI boundary is not simple on both sides.

On the Rust side, it consists to actually use an integer type in the `extern` block declaration, a *robust* type, and then to perform a checked conversion to the enum type.

On the foreign side, it is possible only if the other language allows for stricter checks than plain C. `enum class` in C++ are for instance allowable. Note however that as for reference the actual `extern "C"` ABI of `enum class` is implementation defined and should be verified for each environment.

R43

RECO - Do not use incoming Rust `enum` at FFI boundary

In a secure Rust development, when interfacing with a foreign language, the Rust code should not accept incoming values of any Rust `enum` type.
Exceptions include Rust `enum` types that are:

- opaque in the foreign language and only manipulated from the Rust side,
- bound to safe enums in the foreign language, e.g. `enum class` types in C++.

Concerning fieldless enums, crates like [`num_derive`] or [`num_enum`] allows developer to easily provide safe conversion from integer to enumeration and may be use to safely convert an integer (provided from a C `enum`) into a Rust enum.

7.1.5 Opaque types

Opacifying types is a good way to increase modularity in software development. When doing multilingual development, it is something very common.

R44

RECO - Use dedicated Rust types for foreign opaque types

In a secure Rust development, when binding foreign opaque types, one should use pointers to dedicated opaque types rather than `c_void` pointers.

Currently the recommended way to make a foreign opaque type is like so:

Rust

```
#[repr(C)]
pub struct Foo {_private: [u8; 0]}
extern "C" {
    fn foo(arg: *mut Foo);
}
```

The not yet implemented RFC 1861 proposes to facilitate the coding by allowing to declare opaque types in `extern` blocks.

R45

RECO - Use incomplete C/C++ `struct` pointers to make type opaque

In a secure Rust development, when interfacing with C or C++, Rust types that are to be considered opaque in C/C++ should be translated as incomplete `struct` type (i,e., declared without definition) and be provided with a dedicated constructor and destructor.

Example of opaque Rust type:

Rust

```
struct Opaque {
    // (...) details to be hidden
}

#[no_mangle]
pub unsafe extern "C" fn new_opaque() -> *mut Opaque {
    catch_unwind(|| // Catch panics, see below
        Box::into_raw(Box::new(Opaque {
            // (...) actual construction
        }))
    ).unwrap_or(std::ptr::null_mut())
}

#[no_mangle]
pub unsafe extern "C" fn destroy_opaque(o: *mut Opaque) {
    catch_unwind(||
        if !o.is_null() {
            drop(Box::from_raw(o))
        }
    ); // Only needed if Opaque or one of its subfield is Drop
}
```

7.2 Memory and resource management

Programming languages handle memory in various ways. As a result, it is important to known when transmitting data between Rust and another language which language is responsible for

reclaiming the memory space for this data. The same is true for other kind of resources such as sockets or files.

Rust tracks variable ownership and lifetime to determine at compilation time if and when memory should be deallocated. Thanks to the `Drop` trait, one can exploit this system to reclaim other kind of resources such as file or network access. *Moving* some piece of data from Rust to a foreign language means also abandoning the possible reclamations associated with it.

R46

RULE - Do not use types that implement `Drop` at FFI boundary

In a secure Rust development, Rust code must not implement `Drop` for any types that are directly transmitted to foreign code (i.e. not through a pointer or reference).

In fact, it is advisable to only use `Copy` types. Note that `*const T` is `Copy` even if T is not.

However if not reclaiming memory and resources is bad, using reclaimed memory or reclaiming twice some resources is worst from a security point of view. In order to correctly release a resource only once, one must known which language is responsible for allocating and deallocating memory.

R47

RULE - Ensure clear data ownership in FFI

In a secure Rust development, when data of some type passes without copy through a FFI boundary, one must ensure that:

- A single language is responsible for both allocation and deallocation of data.
- The other language must not allocate or free the data directly but use dedicated foreign functions provided by the chosen language.

Ownership is not enough. It remains to ensure the correct lifetime, mostly that no use occurs after reclamation. It is a lot more challenging. When the other language is responsible for the memory, the best way is to provide a safe wrapper around the foreign type:

R48

RECO - Wrap foreign data in memory releasing wrapper

In a secure Rust development, any non-sensitive foreign piece of data that are allocated and deallocated in the foreign language should be encapsulated in a `Drop` type in such a way as to provide automatic deallocation in Rust, through an automatic call to the foreing language deallocation routine.

A simple example of Rust wrapping over an external opaque type:

Rust

```
/// Private ""raw opaque foreign type Foo
#[repr(C)]
struct RawFoo {
    _private: [u8; 0],
}

/// Private ""raw C API
extern "C" {
```

```
    fn foo_create() -> *mut RawFoo;
    fn foo_do_something(this: *const RawFoo);
    fn foo_destroy(this: *mut RawFoo);
}

/// Foo
pub struct Foo(*mut RawFoo);

impl Foo {
    /// Create a Foo
    pub fn new() -> Option<Foo> {
        let raw_ptr = unsafe { foo_create() };
        if raw_ptr.is_null() {
            None
        } else {
            Some(Foo(raw_ptr))
        }
    }

    /// Do something on a Foo
    pub fn do_something(&self) {
        unsafe { foo_do_something(self.0) }
    }
}

impl Drop for Foo {
    fn drop(&mut self) {
        if !self.0.is_null() {
            unsafe { foo_destroy(self.0) }
        }
    }
}
```

Warning

Because panics may lead to not running the `Drop::drop` method this solution is not sufficient for sensitive deallocation (such as wiping sensitive data) except if the code is guaranteed to never panic.
For wiping sensitive data case, one could address the issue with a dedicated panic handler.

When the foreign language is the one exploiting Rust allocated resources, it is a lot more difficult to offer any guarantee.

In C for instance there is no easy way to check that the appropriate destructor is checked. A possible approach is to exploit callbacks to ensure that the reclamation is done.

The following Rust code is a **thread-unsafe** example of a C-compatible API that provide callback to ensure safe resource reclamation:

Rust

```
pub struct XtraResource {/*fields */}

impl XtraResource {
    pub fn new() -> Self {
        XtraResource { /* ... */}
    }
    pub fn dosthg(&mut self) {
        /*...*/
    }
}
```

```
impl Drop for XtraResource {
    fn drop(&mut self) {
        println!("xtra drop");
    }
}

pub mod c_api {
    use super::XtraResource;
    use std::panic::catch_unwind;

    const INVALID_TAG: u32 = 0;
    const VALID_TAG: u32 = 0xDEAD_BEEF;
    const ERR_TAG: u32 = 0xDEAF_CAFE;

    static mut COUNTER: u32 = 0;

    pub struct CXtraResource {
        tag: u32, // to detect accidental reuse
        id: u32,
        inner: XtraResource,
    }

    #[no_mangle]
    pub unsafe extern "C" fn xtra_with(cb: extern "C" fn(*mut CXtraResource) -> ())
         {
        let inner = if let Ok(res) = catch_unwind(XtraResource::new) {
            res
        } else {
            return;
        };
        let id = COUNTER;
        let tag = VALID_TAG;

        COUNTER = COUNTER.wrapping_add(1);
        // Use heap memory and do not provide pointer to stack to C code!
        let mut boxed = Box::new(CXtraResource { tag, id, inner });

        cb(boxed.as_mut() as *mut CXtraResource);

        if boxed.id == id && (boxed.tag == VALID_TAG || boxed.tag == ERR_TAG) {
            boxed.tag = INVALID_TAG; // prevent accidental reuse
                                     // implicit boxed drop
        } else {
            // (...) error handling (should be fatal)
            boxed.tag = INVALID_TAG; // prevent reuse
            std::mem::forget(boxed); // boxed is corrupted it should not be
        }
    }

    #[no_mangle]
    pub unsafe extern "C" fn xtra_dosthg(cxtra: *mut CXtraResource) {
        let do_it = || {
            if let Some(cxtra) = cxtra.as_mut() {
                if cxtra.tag == VALID_TAG {
                    cxtra.inner.dosthg();
                    return;
                }
            }
            println!("doing nothing with {:p}", cxtra);
        };
        if catch_unwind(do_it).is_err() {
            if let Some(cxtra) = cxtra.as_mut() {
                cxtra.tag = ERR_TAG;
            }
        };
    }
}
```

A compatible C call:

C

```
struct XtraResource;
void xtra_with(void (*cb)(XtraResource* xtra));
void xtra_sthg(XtraResource* xtra);

void cb(XtraResource* xtra) {
    // (...) do anything with the proposed C API for XtraResource
    xtra_sthg(xtra);
}

int main() {
    xtra_with(cb);
}
```

7.3 Panics with foreign code

When calling Rust code from another language (e.g. C), the Rust code must be careful to never panic. Stack unwinding from Rust code into foreign code results in **undefined behavior**.

R49

RULE - Handle `panic!` correctly in FFI

Rust code called from FFI must either ensure the function cannot panic, or use a panic handling mechanism (such as `std::panic::catch_unwind`, `std::panic::set_hook`, `#[panic_handler]`) to ensure the rust code will not abort or return in an unstable state.

Note that `catch_unwind` will only catch unwinding panics, not those that abort the process.

Rust

```
use std::panic::catch_unwind;

fn may_panic() {
    if rand::random() {
        panic!("panic happens");
    }
}

#[no_mangle]
pub unsafe extern "C" fn no_panic() -> i32 {
    let result = catch_unwind(may_panic);
    match result {
        Ok(_) => 0,
        Err(_) => -1,
    }
}
```

7.3.1 `no_std`

In the case of `#![no_std]` program, a panic handler (`#[panic_handler]`) must be defined to ensure security. The panic handler should be written with great care in order to ensure both the safety and security of the program.

Another approach is to simply ensure that there is no use of `panic!` with the `panic-never` crate. Like `no-panic`, `panic-never` relies on a linking trick: the linker fails if a non-trivially-dead branch leads to `panic!`.

7.4 Binding a foreign library in Rust

R50

RECO - Provide safe wrapping to foreign library

Interfacing a library written in another language in Rust should be done in two parts:

- a low-level, possibly *hidden*, module that closely translates the original C API into `extern` blocks,
- a safe wrapping module that ensures memory safety and security invariants at the Rust level.

If the low-level API is exposed to the world, it should be done in a dedicated crate with a name of the form `*-sys`.

The crate rust-bindgen may be used to automatically generate the low-level part of the binding from C header files.

7.5 Binding a Rust library in another language

R51

RECO - Expose dedicated C-compatible API only

In a secure Rust development, exposing a Rust library to a foreign language should only be done through a **dedicated C-compatible API**.

The crate cbindgen may be used to automatically generate C or C++ bindings to the Rust C-compatible API of a Rust library.

7.5.1 Minimal example of a C-exported Rust library

`src/lib.rs`:

Rust

```
/// Opaque counter
pub struct Counter(u32);

impl Counter {
    /// Create a counter (initially at 0)
    fn new() -> Self {
        Self(0)
    }
    /// Get the current value of the counter
    fn get(&self) -> u32 {
        self.0
    }
    /// Increment the value of the counter if there's no overflow
    fn incr(&mut self) -> bool {
```

```
        if let Some(n) = self.0.checked_add(1) {
            self.0 = n;
            true
        } else {
            false
        }
    }
}

// C-compatible API

#[no_mangle]
pub unsafe extern "C" fn counter_create() -> *mut Counter {
    Box::into_raw(Box::new(Counter::new()))
}

#[no_mangle]
pub unsafe extern "C" fn counter_incr(counter: *mut Counter) -> std::os::raw::c_int
     {
    if let Some(counter) = counter.as_mut() {
        if counter.incr() {
            0
        } else {
            -1
        }
    } else {
        -2
    }
}

#[no_mangle]
pub unsafe extern "C" fn counter_get(counter: *const Counter) -> u32 {
    if let Some(counter) = counter.as_ref() {
        return counter.get();
    }
    return 0;
}

#[no_mangle]
pub unsafe extern fn counter_destroy(counter: *mut Counter) -> std::os::raw::c_int
    {
    if !counter.is_null() {
        let _ = Box::from_raw(counter); // get box and drop
        return 0;
    }
    return -1;
}
```

Using cbindgen (`[cbindgen] -l c > counter.h`), one can generate a consistent C header, `counter.h`:

C

```
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

typedef struct Counter Counter;

Counter *counter_create(void);

int counter_destroy(Counter *counter);

uint32_t counter_get(const Counter *counter);

int counter_incr(Counter *counter);
```

counter_main.c:

C

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

#include "counter.h"

int main(int argc, const char** argv) {
    if (argc < 2) {
        return -1;
    }
    size_t n = (size_t)strtoull(argv[1], NULL, 10);

    Counter* c = counter_create();
    for (size_t i=0; i < n; i++) {
        if (counter_incr(c) != 0) {
            printf("overflow\n");
            counter_destroy(c);
            return -1;
        }
    }

    printf("%" PRIu32 "\n", counter_get(c));
    counter_destroy(c);

    return 0;
}
```

Recommendation List

ANSSI-PA-074-EN
Version 1.0 - 06/09/2020

AGENCE NATIONALE DE LA SÉCURITÉ DES SYSTÈMES D'INFORMATION

ANSSI - 51, boulevard de La Tour-Maubourg, 75700 PARIS 07 SP
www.ssi.gouv.fr / conseil.technique@ssi.gouv.fr

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
Premier ministre
SGDSN