RÈGLES DE PROGRAMMATION POUR LE DÉVELOPPEMENT D'APPLICATIONS SÉCURISÉES EN RUST

GUIDE ANSSI

ANSSI-PA-074
09/06/2020

PUBLIC VISÉ :

Développeur Administrateur RSSI DSI Utilisateur

Informations

Attention

Ce document rédigé par l'ANSSI présente les « **Règles de programmation pour le développement d'applications sécurisées en Rust** ». Il est téléchargeable sur le site `www.ssi.gouv.fr`. Il constitue une production originale de l'ANSSI. Il est à ce titre placé sous le régime de la « Licence ouverte » publiée par la mission Etalab (`www.etalab.gouv.fr`). Il est par conséquent diffusable sans restriction.

Ces recommandations n'ont pas de caractère normatif, elles sont livrées en l'état et adaptées aux menaces au jour de leur publication. Au regard de la diversité des systèmes d'information, l'ANSSI ne peut garantir que ces informations puissent être reprises sans adaptation sur les systèmes d'information cibles. Dans tous les cas, la pertinence de l'implémentation des éléments proposés par l'ANSSI doit être soumise, au préalable, à la validation de l'administrateur du système et/ou des personnes en charge de la sécurité des systèmes d'information.

Évolutions du document :

VERSION	DATE	NATURE DES MODIFICATIONS
1.0	09/06/2020	Version initiale

Table des matières

1
Introduction

Rust est un langage multiparadigmes orienté vers la sûreté mémoire.

Il est entre autres orienté programmation système, en permettant une gestion fine de la mémoire sans ramasse-miettes, mais également sans nécessiter d'allocations et de désallocations manuelles, souvent sources d'erreurs et de confusions. Le langage atteint ce but en introduisant un système d'*ownership* (fortement lié à l'*aliasing* des variables). À tout point d'un programme Rust, le compilateur recense les variables qui se réfèrent à une même donnée, et applique un ensemble de règles qui permettent la récupération automatique de la mémoire, la sûreté mémoire et l'absence de problèmes d'accès concurrents.

Le langage est également axé sur la performance, avec des constructions permettant des abstractions à coût nul et un compilateur proposant de puissantes optimisations.

Enfin, le langage Rust fournit des fonctionnalités de programmation de haut niveau. Grâce aux fonctions d'ordre supérieur, aux fermetures, aux itérateurs, etc., il permet d'écrire tout ou parties des programmes dans un style proche des langages de programmation fonctionnelle. En outre, le typage statique, l'inférence de types et le polymorphisme *ad hoc* (sous la forme de *traits*) sont d'autres moyens que Rust propose pour construire des bibliothèques et des programmes de façon sûre.

Néanmoins, du fait de sa polyvalence, le langage offre des constructions et fonctionnalités qui, si elles ne sont pas utilisées correctement, peuvent potentiellement introduire des problèmes de sécurité, soit par construction, soit en permettant d'écrire du code qui serait mal interprété par un développeur ou un relecteur. De plus, comme pour la plupart des outils dans le domaine de la compilation et de la vérification logicielles, les outils utilisés pour développer, mettre au point, compiler et exécuter des programmes proposent des options et des possibilités de configuration qui, si mal utilisées, peuvent introduire des vulnérabilités.

L'objet de ce document est ainsi de rassembler une collection de conseils et de recommandations pour rester autant que possible dans une zone sûre pour le développement d'applications sécurisées, tout en profitant de la variété de possibilités que le langage Rust peut offrir.

1.1 Public visé

Ce guide compile une liste de recommandations qui doivent être observées pour le développement d'applications aux besoins de sécurité élevés. Il peut toutefois être suivi par quiconque souhaite s'assurer que les garanties offertes par la plateforme Rust ne sont pas invalidées par l'utilisation d'une fonctionnalité non sûre, trompeuse ou peu claire.

Il ne vise pas à constituer un cours sur la manière d'écrire des programmes en Rust, il existe déjà une grande quantité de ressources de qualité sur le sujet (par exemple, la page principale de documentation de Rust). L'intention est plutôt de guider le développeur et de l'informer à propos de certains pièges. Ces recommandations forment un complément au bon niveau de confiance que le langage Rust apporte déjà. Ceci étant dit, des rappels peuvent parfois être nécessaires pour la clarté du discours, et le développeur Rust expérimenté peut s'appuyer principalement sur le contenu des encarts (*Règle*, *Recommandation*, *Avertissement*, etc.).

1.2 Contributions

Ce guide est rédigé de manière collaborative et ouverte, via la plateforme GitHub (`https://github.com/ANSSI-FR/rust-guide`). Toutes les contributions pour de futures versions sont les bienvenues, que ce soit directement sous la forme de propositions (*pull requests*) ou bien de suggestions et discussions (*issues*).

1.3 Structure du document

La structure de ce document vise à discuter successivement des différentes phases typiques (et simplifiées) d'un processus de développement. Tout d'abord, nous proposons des recommandations concernant l'utilisation des outils de l'écosystème Rust dans un cadre sécurisé. Ensuite, nous détaillons les précautions à prendre durant le choix et l'utilisation de bibliothèques externes. Enfin, les recommandations à propos des constructions du langage sont présentées. Un résumé des règles et recommandations est disponible à la fin de ce guide.

2
Environnement de développement

2.1 Rustup

Rustup est l'installateur des outils de la chaîne de compilation pour Rust. Entre autres choses, il permet de basculer entre différentes variantes de la chaîne d'outils (*stable*, *beta*, *nightly*), de gérer l'installation des composants additionnels et de maintenir le tout à jour.

Attention

Du point de vue de la sécurité, `rustup` effectue tous les téléchargements en HTTPS, mais ne valide pas les signatures des fichiers téléchargés. Les protections contre les attaques par déclassement, le *pinning* de certificats et la validation des signatures sont des travaux actuellement en cours. Pour les cas les plus sensibles, il peut être préférable d'opter pour une méthode d'installation alternative comme celles listées dans la section *Install* du site officiel du langage Rust.

2.1.1 Éditions Rust

Il existe plusieurs variantes du langage Rust, appelées *éditions*. Le concept d'éditions a été introduit afin de distinguer la mise en place de nouvelles fonctionnalités dans le langage, et de rendre ce processus incrémental. Toutefois, comme mentionné dans le *Edition Guide*, cela ne signifie pas que de nouvelles fonctionnalités et améliorations ne seront incluses que dans la dernière édition.

Certaines éditions peuvent introduire de nouvelles constructions dans le langage et de nouveaux mots-clés. Les recommandations concernant ces fonctionnalités deviennent alors fortement liées à une édition en particulier. Dans le reste de ce guide, un effort sera réalisé pour mettre en évidence les règles qui ne s'appliqueraient qu'à certaines éditions de Rust en particulier.

Note

Aucune édition spécifique n'est recommandée, tant que le développement se conforme aux recommandations exprimées à propos des fonctionnalités que l'édition utilisée propose.

2.1.2 Chaînes d'outils stable, nightly et beta

De manière orthogonale aux éditions qui permettent d'opter pour une variante du langage en termes de fonctionnalités, la chaîne d'outils du langage Rust est déclinée en trois variantes appelées *release channels*.

- La version *nightly* est produite une fois par jour.
- La version *nightly* est promue en version *beta* toutes les six semaines.
- La version *beta* est promue en version *stable* toutes les six semaines.

Lors du développement d'un projet, il est important de vérifier non seulement la version de la chaîne d'outils couramment sélectionnée par défaut, mais aussi les potentielles surcharges qui peuvent être définies en fonction des répertoires :

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

RÈGLE - Utilisation de la chaîne d'outils stable

Le développement d'applications sécurisées doit être mené en utilisant la chaîne d'outils dans sa version *stable*, afin de limiter les potentiels *bugs* à la compilation, à l'exécution et lors de l'utilisation d'outils complémentaires.

Enfin, lorsque l'utilisation d'un outil (par exemple, une sous-commande `cargo`) requiert l'installation d'un composant dans une chaîne d'outils non *stable*, le basculement de variante doit être effectué de la façon la plus locale possible (idéalement, uniquement pendant la commande concernée) au lieu d'explicitement basculer la version courante. Par exemple, pour lancer la version *nightly* de la commande `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

Une fois que la chaîne d'outils appropriée a été sélectionnée avec Rustup, l'outil Cargo est disponible pour exécuter ces différents outils en fournissant la commande `cargo`. Cargo est le gestionnaire de paquetages de Rust. Il joue plusieurs rôles fondamentaux tout au long d'un développement en Rust. Il permet notamment de :

- structurer le projet en fournissant un squelette de projet (`cargo new`) ;

- lancer la compilation du projet (`cargo build`) ;
- lancer la génération de la documentation (`cargo doc`) ;
- lancer les tests (`cargo test`) et les *benchmarks* (`cargo bench`) ;
- gérer le téléchargement des dépendances ;
- rendre le projet distribuable et le publier sur crates.io (`cargo publish`) ;
- lancer des outils complémentaires tels que ceux décrits ci-après, sous la forme de sous-commandes.

Attention

Tout comme `rustup`, `cargo` effectue tous les téléchargements en HTTPS, mais ne valide pas l'index du registre. Des discussions sont en cours pour déterminer le meilleur moyen de protéger et de valider les *crates*. Pour le moment, la sécurité de `cargo` repose sur la bonne sécurité du site web crates.io ainsi que celle du dépôt, hébergé sur GitHub, contenant l'index du registre de *crates*. Pour les cas les plus sensibles, il peut être préférable d'opter pour une méthode d'installation alternative pour les dépendances.

Cargo propose différentes commandes et options pour adapter le processus de compilation aux besoins de chaque projet, principalement au travers du fichier `Cargo.toml`. Pour une présentation complète, voir le *Cargo Book*.

Certaines de ces options requièrent une attention particulière.

La section `[profile.*]` permet de configurer la façon dont le compilateur est invoqué. Par exemple :

- La variable `debug-assertions` contrôle l'activation des assertions de *debug*.
- La variable `overflow-checks` contrôle l'activation de la vérification des dépassements d'entiers lors d'opérations arithmétiques.

Changer les options par défaut pour ces variables peut entraîner l'apparition de *bugs* non détectés, même si le profil de *debug* qui active normalement les vérifications (par exemple, les vérifications de dépassements d'entiers, voir 4.3) est utilisé.

R2

RÈGLE - Conservation des valeurs par défaut des variables critiques dans les profils cargo

Les variables `debug-assertions` et `overflow-checks` ne doivent pas être modifiées dans les sections de profils de développement (`[profile.dev]` and `[profile.test]`).

Cargo propose d'autres moyens de configuration afin de modifier son comportement sur un système donné. Cela peut être très pratique, mais il peut alors aussi être difficile de connaître et de se souvenir de toutes les options qui sont effectivement passées à `cargo`, et en particulier passées ensuite au compilateur Rust. Finalement, cela peut affecter la robustesse du processus de compilation et la confiance qu'on lui accorde. Il est préférable de centraliser les options de compilation dans le fichier de configuration `Cargo.toml`. Pour le cas spécifique de la variable d'environnement `RUSTC_WRAPPER`, utilisée par exemple pour générer une partie du code ou pour invoquer un outil

externe avant la compilation, il est préférable d'utiliser la fonctionnalité de *scripts de compilation* de Cargo.

R3

RÈGLE - Conservation des valeurs par défaut des variables d'environnement à l'exécution de cargo

Les variables d'environnement `RUSTC`, `RUSTC_WRAPPER` et `RUSTFLAGS` ne doivent pas être modifiées lorsque Cargo est appelé pour compiler un projet.

2.2.1 Clippy

Clippy est un outil permettant la vérification de nombreux *lints* (*bugs*, style et lisibilité du code, problèmes de performances, etc.). Depuis la chaîne d'outils stable en version 1.29, `clippy` peut être installé dans l'environnement `rustup` stable. Il est aussi recommandé d'installer `clippy` en tant que composant (`rustup component add clippy`) dans la chaîne d'outils stable plutôt que de l'installer comme une dépendance de chaque projet.

L'outil fournit plusieurs catégories de *lints*, selon le type de problème qu'il vise à détecter dans le code. Les avertissements doivent être revérifiés par le développeur avant d'appliquer la réparation suggérée par `clippy`, en particulier dans le cas des *lints* de la catégorie `clippy::nursery` puisque ceux-ci sont encore en cours de développement et de mise au point.

R4

RÈGLE - Utilisation régulière d'un linter

Un *linter* comme `clippy` doit être utilisé régulièrement tout au long du développement d'une application sécurisée.

2.2.2 Rustfmt

Rustfmt est un outil offrant la possibilité de formater du code en fonction de consignes de style (*style guidelines*). La documentation de l'outil mentionne certaines limitations parmi lesquelles un support partiel des macros (déclaration et utilisation). L'option `--check`, qui affiche les différences de formatage entre le code actuel et le code proposé, doit être utilisé. À la suite de cette première utilisation, l'utilisateur doit vérifier les changements, puis éventuellement les valider en réinvoquant l'outil sans option.

En résumé :

Console

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

Les règles de convention de style peuvent être personnalisées au besoin dans le fichier `rustfmt.toml` ou `.rustfmt.toml` à la racine du projet. Il sera utilisé par l'outil en écrasant les préférences par défaut. Par exemple :

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
```

Pour plus d'informations à propos des règles de convention de style que `rustfmt` propose, voir le *Rust Style Guide*.

R5

RÈGLE - Utilisation d'un outil de formatage (rustfmt)

L'outil de formatage `rustfmt` peut être utilisé pour assurer le respect de règles de convention de style (comme décrites dans le fichier `rustfmt.toml`) sur une base de code, avec l'option `--check` ainsi qu'une revue de code manuelle.

2.2.3 Rustfix

Inclus dans la distribution Rust depuis la fin de l'année 2018, Rustfix est un outil dédié à la réparation des avertissements de compilation et facilitant la transition entre éditions.

Console

```
$ cargo fix
```

Pour préparer la transition d'un projet de l'édition Rust 2015 à Rust 2018, il est possible de lancer la commande suivante :

Console

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

Rustfix va soit réparer le code afin de le rendre compatible avec Rust 2018, ou bien afficher un avertissement décrivant le problème. Le problème devra alors être réparé manuellement. En exécutant la commande (et en réparant potentiellement les problèmes manuellement) jusqu'à ce qu'elle n'affiche plus aucun avertissement, il est possible de s'assurer que le code est compatible tant avec Rust 2015 qu'avec Rust 2018.

Pour basculer définitivement le projet sous Rust 2018 :

Console

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

Il est important de noter que l'outil ne fournit que peu de garanties quant à la correction (*soundness*) des réparations proposées. Dans une certaine configuration, certaines réparations (comme

celles proposées avec l'option `--edition-idioms`) sont connues pour casser la compilation ou pour modifier la sémantique d'un programme dans certains cas.

R6

RÈGLE - Vérification manuelle des réparations automatiques

Dans le cadre du développement d'une application sécurisée, toute réparation automatique (comme celles appliquées par `rustfix` par exemple) doit être vérifiée par le développeur.

2.2.4 Autres

D'autres outils ou sous-commandes `cargo` utiles pour renforcer la sécurité d'un programme existent, par exemple, en recherchant des motifs de code particuliers. Nous en discutons dans les chapitres suivants en fonction de leurs portées et de leurs objectifs.

3
Bibliothèques

En complément de la bibliothèque standard du langage, l'outil `cargo` fournit un moyen pratique d'intégrer des bibliothèques tierces dans un projet en Rust. Ces bibliothèques, appelées *crates* dans l'écosystème Rust, sont importées depuis le dépôt central de composants en sources ouvertes crates.io.

Il doit être noté que la qualité (en termes de sécurité, de performances, de lisibilité, etc.) des *crates* publiées est très variable. De plus, leur maintenance peut être irrégulière ou interrompue. L'usage de chaque composant de ce dépôt doit donc être justifié, et le développeur doit également valider le bon respect des règles du présent guide sur le code correspondant. Plusieurs outils peuvent l'aider dans cette tâche.

3.1 Cargo-outdated

L'outil Cargo-outdated permet de faciliter la gestion des versions des dépendances.

Pour une *crate* donnée, l'outil liste les versions utilisées des dépendances (dépendances listées dans le fichier `Cargo.toml`), et vérifie s'il s'agit de la dernière version compatible disponible ainsi que la dernière version en général.

R7

RÈGLE - Vérification des dépendances obsolètes (cargo-outdated)

L'outil `cargo-outdated` doit être utilisé pour vérifier le statut des dépendances. Ensuite, chaque dépendance importée en version obsolète doit être mise à jour ou bien, le cas échéant, le choix de la version doit être justifié.

3.2 Cargo-audit

Cargo-audit est un outil permettant de vérifier s'il existe des vulnérabilités connues dans la *RustSec Advisory Database* pour les dépendances utilisées dans un projet.

R8

RÈGLE - Vérification des vulnérabilités connues pour les dépendances (cargo-audit)

L'outil `cargo-audit` doit être utilisé pour rechercher des vulnérabilités connues dans les dépendances d'un projet.

4
Généralités sur le langage

4.1 Nommage

La convention de nommage employée par la bibliothèque standard est *de facto* le standard pour le nommage des éléments des programmes écrits en Rust. Un effort a été fait pour formaliser ces conventions de nommage, d'abord dans la RFC 430, puis dans le document des *Rust API Guidelines*.

La règle de base consiste à utiliser :

- *`UpperCamelCase`* pour les types, traits et valeurs d'énumérations ;
- *`snake_case`* pour les fonctions, méthodes, macros, variables et modules ;
- *`SCREAMING_SNAKE_CASE`* pour les variables statiques et les constantes ;
- *`'lowercase`* pour les durées de vie (*lifetimes*).

Les Rust API Guidelines recommandent également des conventions de nommage plus précises pour certaines constructions particulières :

- (C-CONV) pour les méthodes de conversion (`as_`, `to_`, `into_`) ;
- (C-GETTER) pour les accesseurs ;
- (C-ITER) pour les méthodes produisant des itérateurs ;
- (C-ITER-TY) pour les types itérateur ;
- (C-FEATURE) pour les noms de *features* ;
- (C-WORD-ORDER) pour la cohérence sur l'ordre des mots.

R9

RÈGLE - Respect des conventions de nommage

Le développement d'une application sécurisée doit suivre les conventions de nommage décrites dans les Rust API Guidelines.

4.2 Code unsafe

L'utilisation conjointe du système de types et du système d'*ownership* vise à apporter un haut niveau de sûreté quant à la gestion de la mémoire dans les programmes écrits en Rust. Le langage permet alors d'éviter les débordements mémoire, la construction de pointeurs nuls ou invalides, et les problèmes d'accès concurrents à la mémoire. Pour effectuer des actions considérées risquées comme des appels système, des conversions de types ou la manipulation directe de pointeurs mémoire, le langage fournit le mot-clé `unsafe`.

R10

RÈGLE - Non-utilisation des blocs unsafe

Pour un développement sécurisé, les blocs `unsafe` doivent être évités. Ci-dessous, nous listons les seuls cas pour lesquels des blocs `unsafe` peuvent être utilisés, à la condition que leur usage soit justifié :

- L'interfaçage entre Rust et d'autres langages (FFI) permet la déclaration de fonctions dont l'implantation est faite en C, en utilisant le préfixe `extern "C"`. Pour une utiliser une telle fonction, le mot-clé `unsafe` est requis. Un *wrapper* "sûr" doit être défini pour que le code C soit finalement appelé de façon souple et sûre.
- Pour la programmation des systèmes embarqués, on accède souvent aux registres et à d'autres ressources au travers d'adresses mémoire fixées Dans ce cas, des blocs `unsafe` sont nécessaires afin de pouvoir initialiser et déréférencer des pointeurs en Rust pour ces adresses. Afin de minimiser le nombre de déclarations `unsafe` pour permettre au développeur de facilement identifier les accès critiques, une abstraction adaptée (structure de données ou module) doit être mise en place.
- Une fonction peut être marquée globalement comme non sûre (en préfixant sa déclaration par le mot-clé `unsafe`) lorsqu'elle exhibe inévitablement des comportements non sûrs en fonction de ses arguments. Par exemple, cela arrive lorsqu'une fonction doit déréférencer un pointeur passé en argument.

À l'exception de l'un ou plusieurs de ces cas `#[forbid(unsafe_code)]` doit apparaître dans le fichier `main.rs` afin de générer des erreurs de compilation dans le cas ou le mot-clé `unsafe` est utilisé dans le projet.

4.3 Dépassement d'entiers

Bien que des vérifications soient effectuées par Rust en ce qui concerne les potentiels dépassements d'entiers, des précautions doivent être prises lors de l'exécution d'opérations arithmétiques sur les entiers.

En particulier, il doit être noté que les profils de compilation *debug* et *release* produisent des variations de comportements quant à la gestion des dépassements d'entiers. Dans la configuration *debug*, un dépassement provoque la terminaison du programme (`panic`), tandis que dans la configuration *release* la valeur calculée est silencieusement tronquée en fonction de la valeur maximum qui peut être stockée pour le type considéré.

Ce comportement peut être rendu explicite en utilisant le type générique `Wrapping`, ou les opérations sur les entiers `overflowing_<op>` et `wrapping_<op>` (la partie `<op>` étant remplacée par le type de calcul : `add`, `mul`, `sub`, `shr`, etc.).

Rust

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

let x: u8 = 242;

println!("{}", x + 50);   // panique en mode debug, affiche 36 en mode release.
println!("{}", x.overflowing_add(50).0);     // affiche toujours 36.
```

```
println!("{}", x.wrapping_add(50));          // affiche toujours 36.
println!("{}", Wrapping(x) + Wrapping(50)); // affiche toujours 36.

// panique toujours :
let (res, c) = x.overflowing_add(50);
if c { panic!("custom error"); }
else { println!("{}", res); }
```

R11

RÈGLE - Utilisation des opérations arithmétiques appropriées au regard des potentiels dépassements

Lorsqu'une opération arithmétique peut produire un dépassement d'entier, les fonctions spécialisées `overflowing_<op>`, `wrapping_<op>` ou le type `Wrapping` doivent être utilisés.

4.4 Gestion des erreurs

Le type `Result` est la façon privilégiée en Rust pour décrire le type de retour des fonctions dont le traitement peut échouer. Un objet `Result` doit être testé et jamais ignoré.

R12

RECO - Mise en place d'un type `Error` personnalisé pouvant contenir toutes les erreurs possibles

Une *crate* peut implanter son propre type `Error` qui peut contenir toutes les erreurs possibles. Des précautions supplémentaires doivent être prises : ce type doit être *exception-safe* (RFC 1236) et implémenter les traits `Error + Send + Sync + 'static` ainsi que `Display`.

R13

RECO - Utilisation de l'opérateur `?` et non-utilisation de la macro `try!`

L'opérateur `?` doit être utilisé pour améliorer la lisibilité du code. La macro `try!` ne doit pas être utilisée.

Des *crates* tierces peuvent être utilisées pour faciliter la gestion d'erreurs. La plupart (failure, snafu, thiserror) proposent la création de types d'erreurs personnalisées qui implémentent les traits nécessaires et permettent l'encapsulation d'autres erreurs.

Une autre approche (notamment proposée dans anyhow) consiste à envelopper automatiquement les erreurs dans un seul type d'erreur universel. Une telle approche ne devrait pas être utilisée dans des bibliothèques ou des systèmes complexes parce qu'elle ne permet pas de fournir de contexte sur les erreurs ainsi initialement enveloppées, contrairement à la première approche.

4.4.1 Panics

La gestion explicite des erreurs (`Result`) doit être préférée à la place de l'utilisation de la macro `panic`. La cause de l'erreur doit être rendue disponible, et les erreurs trop génériques doivent être évitées.

Les *crates* fournissant des bibliothèques ne doivent pas utiliser de fonctions ou d'instructions qui peuvent échouer en engendrant un `panic`.

Des motifs courants de code qui provoquent des `panic` sont :

- une utilisation de `unwrap` ou de `expect` ;
- une utilisation de `assert` ;
- un accès non vérifié à un tableau ;
- un dépassement d'entier (en mode *debug*) ;
- une division par zéro ;
- l'utilisation de `format!` pour le formatage d'une chaîne de caractères.

R14

RÈGLE - Non-utilisation de fonctions qui peuvent causer des `panic`

Les fonctions et instructions qui peuvent causer des `panic` à l'exécution ne doivent pas être utilisées.

R15

RÈGLE - Test des indices d'accès aux tableaux ou utilisation de la méthode `get`

L'indice d'accès à un tableau doit être testé, ou la méthode `get` doit être utilisée pour récupérer une `Option`.

4.4.2 FFI et `panics`

Lorsque du code Rust est appelé depuis du code écrit dans un autre langage (par exemple, du code C), le code Rust doit être écrit de sorte à ne jamais pouvoir paniquer. Dérouler (*unwinding*) depuis le code Rust vers le code étranger résulte en un comportement indéfini.

R16

RÈGLE - Gestion correcte des `panic!` dans les FFI

Le code Rust appelé depuis une FFI doit soit être assuré de ne pas paniquer, soit utiliser `catch_unwind` ou le module `std::panic` pour s'assurer qu'il ne va pas abandonner un traitement puis que l'exécution retourne dans le langage appelant dans un état instable.

Il est porté à l'attention du développeur que `catch_unwind` ne va traiter que les cas de `panic`, et va préserver les abandons de processus causés par d'autres raisons.

5
Gestion de la mémoire

5.1 `Forget` et fuites de mémoire

En général, la mémoire est automatiquement récupérée en Rust lorsqu'une variable sort de la portée lexicale courante. En complément de ce mécanisme, Rust fournit des fonctions spéciales pour réclamer manuellement la mémoire : les fonctions `forget` et `drop` du module `std::mem` (ou `core::mem`). `drop` déclenche simplement une récupération prématurée de la mémoire tout en appelant les destructeurs associés lorsque nécessaire, `forget` quant à elle n'appelle pas ces destructeurs.

Rust

```
let pair = ('*', 0xBADD_CAFEu32);
drop(pair); // ici, `forget` serait équivalent (pas de destructeur à appeler)
```

Les deux fonctions sont considérées comme **sûres du point de vue mémoire** par Rust. Toutefois, `forget` rendra toute ressource gérée par la valeur libérée inaccessible, mais non libérée.

Rust

```
let s = String::from("Hello");
forget(s); // fuite mémoire
```

En particulier, l'utilisation de `forget` peut causer la rétention en mémoire de ressources critiques, menant à des interblocages et à la persistance de données sensibles en mémoire. C'est pourquoi `forget` doit être considérée comme **non sécurisée**.

R17

RÈGLE - Non-utilisation de `forget`

Dans un développement sécurisé en Rust, la fonction `forget` de `std::mem` (`core::mem`) ne doit pas être utilisée.

R18

RECO - Utilisation du lint clippy pour détecter l'utilisation de `forget`

Le *lint* `mem_forget` de Clippy peut être utilisé pour automatiquement détecter toute utilisation de la fonction `forget`. Pour s'assurer de l'absence d'appel à `forget`, ajouter la directive suivante en début de fichier racine (en général `src/lib.rs` ou `src/main.rs`) :

Rust

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

La bibliothèque standard inclut d'autres moyens d'*oublier* une valeur :

- `Box::leak` pour libérer une ressource ;
- `Box::into_raw` pour exploiter une valeur dans un bloc *unsafe*, notamment dans une FFI ;
- `ManuallyDrop` (dans `std::mem` ou `core::mem`) pour assurer la libération manuelle d'une valeur.

Ces alternatives peuvent mener au même type de problème de sécurité, mais ont l'avantage de faire apparaître explicitement leur but.

R19

RÈGLE - Absence de fuite mémoire

Dans un développement sécurisé en Rust, le code ne doit pas faire fuiter de la mémoire ou des ressources *via* `Box::leak`.

`ManuallyDrop` et `Box::into_raw` passent la responsabilité de la libération de la ressource concernée du compilateur au développeur.

R20

RÈGLE - Libération des valeurs wrappées dans `ManuallyDrop`

Dans un développement sécurisé en Rust, toute valeur *wrappée* dans le type `ManuallyDrop` doit être *unwrapped* pour permettre sa libération automatique (`ManuallyDrop::into_inner`) ou bien doit être manuellement libérée (*unsafe* `ManuallyDrop::drop`).

R21

RÈGLE - Appel systématique à `from_raw` pour les valeurs créées avec `into_raw`

Dans un développement sécurisé en Rust, tout pointeur créé par un appel à `into_raw` (ou `into_raw_nonnull`) depuis un des types suivants doit finalement être transformé en valeur avec l'appel à la fonction `from_raw` correspondant, pour permettre sa libération :

- `std::boxed::Box` (ou `alloc::boxed::Box`) ;
- `std::rc::Rc` (ou `alloc::rc::Rc`) ;
- `std::rc::Weak` (ou `alloc::rc::Weak`) ;
- `std::sync::Arc` (ou `alloc::sync::Arc`) ;
- `std::sync::Weak` (ou `alloc::sync::Weak`) ;
- `std::ffi::CString` ;
- `std::ffi::OsString`.

Rust

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

Note

Dans le cas de `Box::into_raw`, le nettoyage automatique est possible, mais est bien plus compliqué que de *re-boxer* le pointeur brut et doit être évité :

Rust

```
// extrait de la documentation de la bibliothèque standard
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>());
}
```

Puisque les autres types (`Rc` et `Arc`) sont opaques et plus complexes, la libération manuelle n'est pas possible.

5.2 Mémoire non initialisée

Par défaut, le langage Rust impose que toutes les valeurs soient initialisées, pour prévenir l'utilisation de mémoire non initialisée (à l'exception de l'utilisation de `std::mem::uninitialized` ou de `std::mem::MaybeUninit`).

R22

RÈGLE - Pas de mémoire non initialisée

La fonction `std::mem::uninitialized` (dépréciée depuis la version 1.38) ou le type `std::mem::MaybeUninit` (stabilisé dans la version 1.36) ne doivent pas être utilisés, ou bien explicitement justifiés si nécessaire.

L'utilisation de mémoire non initialisée peut induire deux problèmes de sécurité distincts :

- la libération de mémoire non initialisée (étant également un problème de sûreté mémoire) ;
- la non-libération de mémoire initialisée.

Note

Le type `std::mem::MaybeUninit` est une amélioration de la fonction `std::mem::uninitialized`. En effet, il rend la libération des valeurs non initialisées bien plus difficile. Toutefois, cela ne change pas le second problème : la non-libération de la mémoire initialisée est bien possible. C'est problématique en particulier si l'on considère l'utilisation de `Drop` pour effacer des valeurs sensibles.

5.3 Effacement sécurisé des informations sensibles

L'effacement sécurisé (mise à zéro) est nécessaire pour les variables sensibles, en particulier dans lorsque le code Rust est utilisé *via* des FFI.

R23

RÈGLE - Mise à zéro des données sensibles après utilisation

Les variables contenant des données sensibles doivent être mises à zéro après utilisation, en utilisant des fonctions dont les appels ne seront pas supprimés par les optimisations du compilateur, comme `std::ptr::write_volatile` ou bien la *crate* `zeroize`.

Le code suivant montre comment définir un type entier qui sera remis à zéro à sa libération, en utilisant le trait `Drop` :

Rust

```
/// Exemple : newtype pour u32, réécrit à 0 quand libéré
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 est libéré ici
```

6
Système de types

6.1 Traits de la bibliothèque standard

6.1.1 Trait `Drop` : le destructeur

Les types implémentent le trait `std::ops::Drop` dans le but d'effectuer certaines opérations lorsque la mémoire associée à une valeur est réclamée. `Drop` est l'équivalent Rust d'un destructeur en C++ ou un finaliseur en Java.

`Drop` agit récursivement, depuis la valeur externe vers les valeurs imbriquées. Lorsqu'une valeur sort du scope (ou est explicitement relâchée avec `std::mem::drop`), elle est relâchée en deux étapes. La première étape a lieu uniquement si le type de la valeur en question implémente le trait `Drop` et consiste en l'appel de la méthode `drop`. La seconde étape consiste en la répétition de processus de *drop* récursivement sur tous les champs que contient la valeur. Il est à noter que l'implémentation de `Drop` est *responsable uniquement de la valeur extérieure*.

Tout d'abord, l'implémentation de `Drop` ne doit pas être systématique. Elle est nécessaire uniquement lorsque le type requiert un traitement logique à la destruction. `Drop` est typiquement utilisé dans le cas du relâchement des ressources externes (connexions réseau, fichier, etc.) ou de ressources mémoire complexes (*smart pointers* comme les `Box` ou les `Rc` par exemple). Au final, il est probable que l'implémentation du trait `Drop` contienne des blocs `unsafe` ainsi que d'autres opérations critiques du point de vue de la sécurité.

R24

RECO - Justification de l'implémentation du trait `Drop`

Dans un développement sécurisé en Rust, l'implémentation du trait `std::ops::Drop` doit être justifiée, documentée et examinée par des pairs.

Ensuite, le système de types de Rust assure seulement la sûreté mémoire et, du point de vue du typage, des `drops` peuvent tout à fait être manqués. Plusieurs situations peuvent mener à manquer des `drops`, comme :

- un cycle dans la référence (par exemple avec `Rc` ou `Arc`) ;
- un appel explicite à `std::mem::forget` (ou `core::mem::forget`) (voir section 5.1) ;
- un `panic` dans un `drop` ;
- un arrêt du programme (et un `panic` lorsque `abort-on-panic` est activé).

Les `drops` manqués peuvent mener à l'exposition de données sensibles ou bien encore à l'épuisement de ressources limitées et par là même à des problèmes d'indisponibilité.

R25

RÈGLE - Absence de `panic` dans l'implémentation de `Drop`

Dans un développement sécurisé en Rust, l'implémentation du trait `std::ops::Drop` ne doit pas causer de `panic`.

En plus des `panics`, les `drops` contenant du code critique doivent être protégés.

R26

RÈGLE - Absence de cycles de références avec valeurs `Dropables`

Les valeurs dont le type implémente `Drop` ne doivent pas être incluses, directement ou indirectement, dans un cycle de références à compteurs.

R27

RECO - Sécurité assurée par d'autres mécanismes en plus du trait `Drop`

Certaines opérations liées à la sécurité d'une application à la fin d'un traitement (comme l'effacement de secrets cryptographiques par exemple) ne doivent pas reposer uniquement sur l'implémentation du trait `Drop`.

6.1.2 Les traits `Send` et `Sync`

Les traits `Send` et `Sync` (définis dans `std::marker` ou `core::marker`) sont des marqueurs utilisés pour assurer la sûreté des accès concurrents en Rust. Lorsqu'ils sont correctement implémentés, ils permettent au compilateur Rust de garantir l'absence de problèmes d'accès concurrents. Leurs sémantiques sont définies comme suit :

- Un type est `Send` s'il est sûr d'envoyer (*move*) des valeurs de ce type vers un autre fil d'exécution.
- Un type est `Sync` s'il est sûr de partager des valeurs de ce type par une référence immutable avec un autre fil d'exécution.

Ces deux traits sont des traits *unsafe*, c'est-à-dire que le compilateur Rust ne vérifie d'aucune manière que leur implémentation est correcte. Le danger est réel : une implémentation incorrecte peut mener à un **comportement indéfini**.

Heureusement, dans la plupart des cas, il n'est pas nécessaire de fournir une implémentation. En Rust, la quasi-totalité des types primitifs implémente `Send` et `Sync`, et dans la majorité des cas, Rust fournit de manière automatique une implémentation pour les types composés. Quelques exceptions notables sont :

- les pointeurs `raw`, qui n'implémentent ni `Send`, ni `Sync`, puisqu'ils n'offrent aucune garantie quant à la sûreté ;
- les références `UnsafeCell`, qui n'implémentent pas `Sync` (et par extensions, les références `Cell` et `RefCell` non plus), puisqu'elles autorisent la mutabilité des valeurs contenues (*interior mutability*) ;
- les références `Rc`, qui n'implémentent ni `Send`, ni `Sync`, puisque les compteurs de références seraient partagés de manière désynchronisée.

L'implémentation automatique de `Send` (respectivement `Sync`) a lieu pour les types composés (structures ou énumérations) lorsque tous les champs contenus implémentent `Send` (respectivement `Sync`). Une fonctionnalité notable, mais **instable**, de Rust (depuis 1.37.0) permet d'empêcher cette implémentation automatique en annotant explicitement le type considéré avec une *négation d'implementation* :

Rust

```
#![feature(option_builtin_traits)]

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

L'implémentation négative de `Send` ou `Sync` est également utilisée dans la bibliothèque standard pour les exceptions, et est automatiquement implémentée lorsque cela est approprié. En résultat, la documentation générée est toujours explicite : un type implémente soit `Send` (respectivement `Sync`), soit `!Send` (respectivement `!Sync`).

En guise d'alternative *stable* à l'implémentation négative, il est possible d'utiliser un champ typé par un type fantôme (`PhantomData`) :

Rust

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

R28

RECO - Justification de l'implémentation des traits `Send` et `Sync`

Dans un développement sécurisé en Rust, l'implémentation manuelle des traits `Send` et `Sync` doit être évitée, et, si nécessaire, doit être justifiée, documentée et révisée par des pairs.

6.1.3 Les traits de comparaison : `PartialEq`, `Eq`, `PartialOrd`, `Ord`

Les comparaisons (==, !=, <, <=, >, >=) en Rust reposent sur quatre traits de la bibliothèque standard disponibles dans `std::cmp` (ou `core::cmp` pour une compilation avec `no_std`) :

- `PartialEq<Rhs>` qui définit la relation d'équivalence partielle entre objets de types `Self` et `Rhs` ;
- `PartialOrd<Rhs>` qui définit la relation d'ordre partiel entre les objets de types `Self` et `Rhs` ;
- `Eq` qui définit la relation d'équivalence totale entre les objets du même type. Il s'agit d'un trait de marquage qui requiert le trait `PartialEq<Self>` ;
- `Ord` qui définit la relation d'ordre total entre les objets du même type. Le trait `PartialOrd<Self>` est alors requis.

Comme stipulé dans la documentation de la bibliothèque standard, Rust présuppose **de nombreux invariants** lors de l'implémentation de ces traits :

- Pour `PartialEq` :
  - *Cohérence interne* : `a.ne(b)` est équivalent à `!a.eq(b)`, c.-à-d., `ne` est le strict inverse de `eq`. Cela correspond précisément à l'implémentation par défaut de `ne`.
  - *Symétrie* : `a.eq(b)` et `b.eq(a)` sont équivalents. Du point de vue du développeur, cela signifie que :
    - `PartialEq<B>` est implémenté pour le type `A` (noté `A: PartialEq<B>`).
    - `PartialEq<A>` est implémenté pour le type `B` (noté `B: PartialEq<A>`).
    - Les deux implémentations sont cohérentes l'une avec l'autre.
  - *Transitivité* : `a.eq(b)` et `b.eq(c)` impliquent `a.eq(c)`. Cela signifie que :
    - `A: PartialEq<B>`.
    - `B: PartialEq<C>`.
    - `A: PartialEq<C>`.
    - Les trois implémentations sont cohérentes les unes avec les autres (ainsi qu'avec leurs implémentations symétriques).
- Pour `Eq` :
  - `PartialEq<Self>` est implémenté.
  - *Réflexivité* : `a.eq(a)`. Cela signifie que `PartialEq<Self>` est implémenté (`Eq` ne fournit aucune méthode).
- Pour `PartialOrd` :
  - *Consistance de la relation d'égalité* : `a.eq(b)` est équivalent à `a.partial_cmp(b) == Some(std::ordering::Eq)`.
  - *Consistence interne* :
    - `a.lt(b)` ssi `a.partial_cmp(b) == Some(std::ordering::Less)`.
    - `a.gt(b)` ssi `a.partial_cmp(b) == Some(std::ordering::Greater)`.
    - `a.le(b)` ssi `a.lt(b) || a.eq(b)`.
    - `a.ge(b)` ssi `a.gt(b) || a.eq(b)`.

    Il faut noter qu'en définissant seulement `partial_cmp`, la consistance interne est garantie par les implémentations par défaut de `lt`, `le`, `gt`, and `ge`.
  - *Antisymétrie* : `a.lt(b)` (respectivement `a.gt(b)`) implique `b.gt(a)` (respectivement `b.lt(b)`). Du point de vue du développeur, cela signifie que :
    - `A: PartialOrd<B>`.
    - `B: PartialOrd<A>`.
    - Les deux implémentations sont cohérentes l'une avec l'autre.
  - *Transitivité* : `a.lt(b)` et `b.lt(c)` impliquent `a.lt(c)` (également avec `gt`, `le` et `ge`). Cela signifie que :
    - `A: PartialOrd<B>`.
    - `B: PartialOrd<C>`.
    - `A: PartialOrd<C>`.
    - Les trois implémentations sont cohérentes les unes avec les autres (et avec leurs implémentations symétriques).

- Pour `Ord` :
  - `PartialOrd<Self>`
  - *Totalité* : `a.partial_cmp(b) != None` est toujours vrai. En d'autres mots, exactement une assertion parmi `a.eq(b)`, `a.lt(b)` et `a.gt(b)` est vraie.
  - *Cohérence avec `PartialOrd<Self>`* : `Some(a.cmp(b)) == a.partial_cmp(b)`.

Le compilateur ne vérifie aucun de ces prérequis, à l'exception des vérifications sur les types. Toutefois, les comparaisons sont des éléments importants puisqu'elles jouent un rôle tant dans les propriétés de vivacité des systèmes critiques comme des ordonnanceurs ou des répartiteurs de charge que dans les algorithmes optimisés qui peuvent éventuellement utiliser des blocs `unsafe`. Dans le premier cas d'usage, une mauvaise relation d'ordre peut causer des problèmes de disponibilité comme des interblocages. Dans le second cas, cela peut mener à des problèmes classiques de sécurité liés à des violations de propriétés de sûreté mémoire. C'est là encore un atout que de limiter au possible l'utilisation des blocs `unsafe`.

R29

RÈGLE - Respect des invariants des traits de comparaison standards

Dans un développement sécurisé en Rust, l'implémentation des traits de comparaison standards doit respecter les invariants décrits dans la documentation.

R30

RECO - Utilisation des implémentations par défaut des traits de comparaison standards

Dans un développement sécurisé en Rust, l'implémentation des traits de comparaison standard ne doit être effectuée que par l'implémentation des méthodes ne fournissant pas d'implémentation par défaut, dans le but de réduire le risque de violation des invariants associés auxdits traits.

Il existe un *lint* Clippy qui permet de vérifier que `PartialEq::ne` n'est pas défini lors d'une implémentation du trait `PartialEq`.

Rust propose une façon de fournir automatiquement des implémentations par défaut pour les traits de comparaison, au travers de l'attribut `#[derive(...)]` :

- La dérivation de `PartialEq` implémente `PartialEq<Self>` avec une **égalité structurelle** à condition que chacun des types des données membres implémente `PartialEq<Self>`.
- La dérivation de `Eq` implémente le trait de marquage `Eq` à condition que chacun des types des données membres implémente `Eq`.
- La dérivation de `PartialOrd` implémente `PartialOrd<Self>` comme un **ordre lexicographique** à condition que chacun des types des données membres implémente `PartialOrd`.
- La dérivation de `Ord` implémente `Ord` comme un **ordre lexicographique** à condition que chacun des types des données membres implémente `Ord`.

Par exemple, le court extrait de code suivant montre comment comparer deux valeurs de type `T1` facilement. Toutes les assertions sont vraies.

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 });
```

Attention

La dérivation des traits de comparaison pour les types composites dépend de **l'ordre de déclaration des champs** et non de leur nom.
D'abord, cela implique que changer l'ordre des champs modifie l'ordre des valeurs. Par exemple, en considérant le type suivant :

Rust

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

on a `T1 {a: 1, b: 0} > T1 {a: 0, b: 1}` mais `T2 {a: 1, b: 0} < T2 {a: 0, b: 1}`.
Ensuite, si une comparaison sous-jacente provoque un `panic`, l'ordre peut changer le résultat à cause de l'utilisation d'un opérateur logique court- circuitant dans l'implémentation automatique.
Pour les énumérations, les comparaisons dérivées dépendent d'abord de **l'ordre des variants**, puis de l'ordre des champs.

En dépit de ces avertissements sur les ordres dérivés, les comparaisons dérivées automatiquement sont bien moins sujettes à erreurs que des implémentations manuelles, et rendent le code plus court et plus simple à maintenir.

R31

RECO - Dérivation des traits de comparaison lorsque c'est possible

Dans un développement sécurisé en Rust, l'implémentation des traits de comparaison standard doit être automatiquement dérivée à l'aide de `#[derive(...)]` lorsque l'égalité structurelle et l'ordre lexicographique sont nécessaires. Toute implémentation manuelle d'un trait de comparaison standard doit être justifiée et documentée.

7
Interfaçage avec des fonctions externes (FFI)

L'approche de Rust en ce qui concerne l'interfaçage avec des fonctions d'autres langages repose sur une compatibilité forte avec le C. Toutefois, cette frontière est par nature **non sûre** (voir Rust Book : Unsafe Rust).

Les fonctions marquées comme externes (mot-clé `extern`) sont rendues compatibles avec du code C à la compilation. Elles peuvent être appelées depuis un code C avec n'importe quelle valeur en argument. La syntaxe complète est `extern "<ABI>"` où `"<ABI>"` décrit la convention d'appel et dépend de la plateforme d'exécution visée. Par défaut, elle vaut `"C"`, ce qui correspond à la manière standard en C d'appeler des fonctions.

Rust

```
// exportation d'une fonction compatible avec le C
unsafe extern "C" fn mylib_f(param: u32) -> i32 {
    if param == 0xCAFEBABE { 0 } else { -1 }
}
```

Pour que la fonction `mylib_f` soit accessible avec le même nom, la fonction doit être annotée avec l'attribut `#[no_mangle]`).

À l'inverse, il est possible d'appeler des fonctions écrites en C depuis du code Rust si celles-ci sont déclarées dans un bloc `extern` :

Rust

```
use std::os::raw::c_int;
// importation d'une fonction externe de la libc
extern "C" {
    fn abs(args: c_int) -> c_int;
}

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

Note

Toute fonction écrite dans un autre langage et importée dans Rust par l'usage d'un bloc `extern` est **automatiquement *unsafe***. C'est pourquoi tout appel à une telle fonction doit être fait dans un contexte `unsafe`.

Les blocs `extern` peuvent également contenir des déclarations de variables globales externes, préfixées alors par le mot-clé `static` :

Rust

```
//! Un accès direct aux variables d'environnement (sur Unix).
//! Ne doit pas être utilisé ! Non *thread-safe*, voir `std::env` !

extern {
    // Variable globale de la libc
    #[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 Typage

Le typage est le moyen qu'utilise Rust pour assurer la sûreté mémoire. Lors de l'interfaçage avec d'autres langages, qui n'offrent peut-être pas les mêmes garanties, le choix des types lors du *binding* est essentiel pour maintenir au mieux cette sûreté mémoire.

7.1.1 Agencement des données

Rust ne fournit aucune garantie, que ce soit sur un court ou un long terme, vis-à-vis de la façon dont sont agencées les données en mémoire. La seule manière de rendre les données compatibles avec d'autres langages est la déclaration explicite de la compatibilité avec le C, avec l'attribut `repr` (voir Rust Reference : Type Layout). Par exemple, les types Rust suivants :

Rust

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

sont compatibles avec les types C suivants :

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

RÈGLE - Utilisation exclusive de types compatibles avec le C dans les FFI

Dans un développement sécurisé, seuls les types compatibles avec le C peuvent être utilisés comme argument ou type de retour des fonctions importées ou exportées et comme type de variables globales importées ou exportées.
La seule exception à cette règle est l'utilisation de types considérés comme **opaques** du côté du langage externe.

Les types suivants sont considérés comme compatibles avec le C :

- les types primitifs entiers et à virgule flottante ;
- les `structs` annotées avec `repr(C)` ;
- les `enums` annotées avec `repr(C)` ou `repr(Int)` (où `Int` est un type primitif entier), contenant au moins un variant et dont tous les variants ne comportent pas de champ ;
- les pointeurs.

Les types suivants ne sont pas compatibles avec le C :

- les types à taille variable ;
- les `trait objects` ;
- les `enums` dont les variants comportent des champs ;
- les n-uplets (sauf les `structs` à n-uplet annotées avec `repr(C)`).

Certains types sont compatibles, mais avec certaines limitations :

- les types à taille nulle, qui ne sont pas spécifiés pour le C et mènent à des contradictions dans les spécifications du C++ ;
- les `enums` avec champs annotés avec `repr(C)`, `repr(C, Int)` ou `repr(Int)` (voir RFC 2195).

7.1.2 Cohérence du typage

R33

RÈGLE - Utilisation de types cohérents pour les FFI

Les types doivent être cohérents entre les deux côtés des frontières des FFI.
Bien que certains détails peuvent être masqués de la part d'un côté envers l'autre (typiquement, pour rendre un type opaque), les types des deux parties doivent avoir la même taille et respecter le même alignement.

En ce qui concerne les `enums` avec des champs en particulier, les types correspondant en C (ou en C++) ne sont pas évidents (RFC 2195).

Les outils permettant de générer automatiquement des *bindings*, comme rust-bindgen ou cbindgen, peuvent aider à assurer la cohérence entre les types du côté C et ceux du côté Rust.

R34

RECO - Utilisation des outils de génération automatique de bindings

Dans un développement sécurisé en Rust, les outils de génération automatique de *bindings* doivent être utilisés lorsque cela est possible, et ce en continu.

Attention

Pour les *bindings* C/C++ vers Rust, rust-bindgen est capable de générer automatiquement des *bindings* de bas niveau. L'écriture d'un *binding* de plus haut niveau est fortement recommandée (voir recommandation « Mise en place d'une encapsulation sûre pour les bibliothèques externes »). Attention également à certaines options dangereuses de `rust-bindgen`, en particulier `rustified_enum`.

7.1.3 Types dépendants de la plateforme d'exécution

Lors de l'interfaçage avec un langage externe, comme C ou C++, il est souvent nécessaire d'utiliser des types dépendants de la plateforme d'exécution, comme les `int`s C, les `long`s, etc.

En plus du type `c_void` de `std::ffi` (ou `core::ffi`) pour le type C `void`, la bibliothèque standard offre des alias de types portables dans `std::os::raw` (or `core::os::raw`) :

- `c_char` pour `char` (soit `i8` ou bien `u8`) ;
- `c_schar` pour `signed char` (toujours `i8`) ;
- `c_uchar` pour `unsigned char` (toujours `u8`) ;
- `c_short` pour `short` ;
- `c_ushort` pour `unsigned short` ;
- `c_int` pour `int` ;
- `c_uint` pour `unsigned int` ;
- `c_long` pour `long` ;
- `c_ulong` pour `unsigned long` ;

- `c_longlong` pour `long long` ;
- `c_ulonglong` pour `unsigned long long` ;
- `c_float` pour `float` (toujours `f32`) ;
- `c_double` pour `double` (toujours `f64`).

La crate libc offre des types supplémentaires compatibles avec le C qui couvrent la quasi-entièreté de la bibliothèque standard du C.

R35

RÈGLE - Utilisation des alias portables `c_*` pour faire correspondre les types dépendants de la plateforme d'exécution

Dans un développement sécurisé en Rust, lors de l'interfaçage avec du code faisant usage de types dépendants de la plateforme d'exécution, comme les `int`s et les `long`s du C, le code Rust doit utiliser les alias portables de types, comme ceux fournis dans la bibliothèque standard ou dans la crate libc, au lieu des types spécifiques à la plateforme, à l'exception du cas où les *bindings* sont générés automatiquement pour chaque plateforme (voir la note ci-dessous).

Note

Les outils de génération automatiques de *bindings* (par exemple cbindgen ou rust-bindgen) sont capables d'assurer la cohérence des types dépendants de la plateforme. Ils doivent être utilisés durant le processus de compilation pour chaque cible afin d'assurer que la génération est cohérente pour la plateforme visée.

7.1.4 Types non-robustes : références, pointeurs de fonction, énumérations

Une *représentation piégeuse* d'un type particulier est une représentation (motif d'octets) qui respecte les contraintes de représentation du type (telles que sa taille et son alignement), mais qui ne représente pas une valeur valide de ce type et mène à des comportements indéfinis.

En d'autres termes, si une telle valeur invalide est affectée à une variable Rust, tout peut arriver ensuite, d'un simple *crash* à une exécution de code arbitraire. Quand on écrit du code Rust sûr, ce genre de comportement ne peut arriver (à moins d'un *bug* dans le compilateur Rust). Toutefois, en écrivant du code Rust non sûr, et en particulier dans des FFI, cela peut facilement avoir lieu.

Dans la suite, on appelle des **types non-robustes** les types dont les valeurs peuvent avoir ces représentations piégeuses (au moins une). Beaucoup de types Rust sont non-robustes, même parmi les types compatibles avec le C :

- `bool` (1 octet, 256 représentations, seules deux d'entre elles valides) ;
- les références ;
- les pointeurs de fonction ;
- les énumérations ;
- les flottants (même si de nombreux langages ont la même compréhension de ce qu'est un flottant valide) ;

- les types composés qui contiennent au moins un champ ayant pour type un type non-robuste.

De l'autre côté, les types entiers (`u*`/`i*`), les types composés *packés* qui ne contiennent pas de champs de type non-robuste, sont par exemple des *types robustes*.

Les types non-robustes engendrent des difficultés lors de l'interfaçage entre deux langages. Cela revient à décider **quel langage des deux est le plus responsable pour assurer la validité des valeurs hors bornes** et comment mettre cela en place.

R36

RÈGLE - Non-vérification des valeurs de types non-robustes

Dans un développement sécurisé en Rust, toute valeur externe de type non- robuste doit être vérifiée.
Plus précisément, soit une conversion (en Rust) est effectuée depuis des types robustes vers des types non-robustes à l'aide de vérifications explicites, soit le langage externe offre des garanties fortes quant à la validité des valeurs en question.

R37

RECO - Vérification des valeurs externes en Rust

Dans un développement Rust sécurisé, la vérification des valeurs provenant d'un langage externe doit être effectuée du côté Rust lorsque cela est possible.

Ces règles génériques peuvent être adaptées à un langage externe spécifique ou selon les risques associés. En ce qui concerne les langages, le C est particulièrement inapte à offrir des garanties de validité. Toutefois, Rust n'est pas le seul langage à offrir de telles possibilités. Par exemple, un certain sous-ensemble de C++ (sans la réinterprétation) permet au développeur de faire beaucoup dans ce domaine à l'aide du typage. Parce que Rust sépare nativement les segments sûrs des segments non-sûrs, la recommandation est de toujours utiliser Rust pour les vérifications lorsque c'est possible. En ce qui concerne les risques, les types présentant le plus de dangers sont les références, les références de fonction et les énumérations, qui sont discutées ci-dessous.

Attention

Le type `bool` de Rust a été rendu équivalent au type `_Bool` (renommé `bool` dans `<stdbool.h>`) de C99 et au type `bool` de C++. Toutefois, charger une valeur différente de 0 ou 1 en tant que `_Bool`/`bool` est un comportement indéfini *des deux côtés*. La partie sûre de Rust assure ce fait. Les compilateurs C et C++ assurent qu'aucune autre valeur que 0 et 1 ne peut être *stockée* dans un `_Bool`/`bool` mais ne peuvent garantir l'absence d'une *réinterprétation incorrecte* (par exemple dans un type union ou *via* un *cast* de pointeur). Pour détecter une telle réinterprétation, un *sanitizer* tel que l'option `-fsanitize=bool` de LLVM peut être utilisé.

7.1.4.1 Références et pointeurs

Bien qu'autorisée par le compilateur Rust, l'utilisation des références Rust dans une FFI peut casser la sûreté mémoire. Parce que leur côté non sûr est plus explicite, les pointeurs sont préférés aux références Rust pour un interfaçage avec un autre langage.

D'un autre côté, les types des références ne sont pas robustes : ils permettent seulement de pointer vers des objets valides en mémoire. Toute déviation mène à des comportements indéfinis.

R38

RÈGLE - Vérification des références provenant d'un langage externe

Dans un développement sécurisé en Rust, les références externes transmises au côté Rust par le biais d'une FFI doivent être **vérifiées du côté du langage externe**, que ce soit de manière automatique (par exemple, par un compilateur) ou de manière manuelle.
Les exceptions comprennent les références Rust *wrappées* de façon opaque et manipulées uniquement du côté Rust, et les références *wrappées* dans un type `Option` (voir note ci-dessous).

Lors d'un *binding* depuis et vers le C, le problème peut être particulièrement sévère, parce que le langage C n'offre pas de références (dans le sens de pointeurs valides) et le compilateur n'offre pas de garantie de sûreté.

Lors d'un *binding* avec le C++, les références Rust peuvent en pratique être liées aux références C++ bien que l'ABI d'une fonction `extern "C"` en C++ avec des références soit défini par l'implémentation. Enfin, le code C++ doit être vérifié pour éviter toute confusion de pointeurs et de références.

Les références Rust peuvent être raisonnablement utilisées avec d'autres langages compatibles avec le C, incluants les variantes de C qui mettent en oeuvre la vérification que les pointeurs sont non nuls, comme du code annoté à l'aide Microsoft SAL par exemple.

R39

RECO - Non-utilisation des types références et utilisation des types pointeurs

Dans un développement sécurisé en Rust, le code Rust ne doit pas utiliser de types références, mais des types pointeurs.
Les exceptions sont :

- les références qui sont opaques dans le langage externe et qui sont seulement manipulées du côté Rust ;
- les références *wrappées* dans un type `Option` (voir note ci-dessous) ;
- les références liées à des références sûres dans le langage externe, par exemple dans des variantes du C ou dans du code compilé en C++ dans un environnement où les références de fonctions `extern "C"` sont encodées comme des pointeurs.

D'un autre côté, les *types pointeur* Rust peuvent aussi mener à des comportements indéfinis, mais sont plus aisément vérifiables, principalement par la comparaison avec `std/code::ptr::null()` (`((void*)0` en C), mais aussi dans certains contextes par la vérification de l'appartenance à une plage d'adresses mémoire (en particulier dans des systèmes embarqués ou pour un développement au niveau noyau). Un autre avantage à utiliser les pointeurs Rust dans des FFI est que tout chargement de valeur pointée est clairement marqué comme appartenant à un bloc ou à une fonction `unsafe`.

R40

RÈGLE - Vérification des pointeurs externes

Dans un développement sécurisé en Rust, tout code Rust qui déréférence un pointeur externe doit vérifier sa validité au préalable. En particulier, les pointeurs doivent être vérifiés comme étant non nuls avant toute utilisation.
Des approches plus strictes sont recommandées lorsque cela est possible. Elles comprennent la vérification des pointeurs comme appartenant à une plage d'adresses mémoire valides ou comme étant des pointeurs avérés (étiquetés ou signés). Cette approche est particulièrement applicable si la valeur pointée est seulement manipulée depuis le code Rust.

Le code suivant est un simple exemple d'utilisation de pointeur externe dans une fonction Rust exportée :

Rust

```
/// Ajout en place
#[no_mangle]
pub unsafe extern fn add_in_place(a: *mut u32, b: u32) {
    // Vérification du caractère non nul de `a`
    // et manipulation comme une référence mutable
    if let Some(a) = a.as_mut() {
        *a += b
    }
}
```

Il faut noter que les méthodes `as_ref` et `as_mut` (pour les pointeurs mutables) permettent d'accéder facilement à la référence tout en assurant une vérification du caractère non nul de manière très idiomatique en Rust. Du côté du C, la fonction peut alors être utilisée comme suit :

C

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

//! Ajout en 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

Les valeurs de type `Option<&T>` ou `Option<&mut T>`, pour tout T tel que `T: Sized`, sont admissibles dans un FFI à la place de pointeurs avec comparaison explicite avec la valeur nulle. En raison de la garantie de Rust vis-à-vis des optimisations de pointeurs pouvant être nuls, un pointeur nul est acceptable du côté C. La valeur C `NULL` est comprise par Rust comme la valeur `None`, tandis qu'un pointeur non nul est encapsulé dans le constructeur `Some`. Bien qu'ergonomique, cette fonctionnalité ne permet par contre pas des validations fortes des valeurs de pointeurs comme l'appartenance

à une plage d'adresses mémoire valides.

7.1.4.2 Pointeurs de fonction

Les pointeurs de fonction qui traversent les frontières d'une FFI peuvent mener à de l'exécution de code arbitraire et impliquent donc des risques réels de sécurité.

R41

RÈGLE - Marquage des types de pointeurs de fonction dans les FFI comme `extern` et `unsafe`

Dans un développement sécurisé en Rust, tout type de pointeur de fonction dont les valeurs sont amenées à traverser les frontières d'une FFI doit être marqué comme `extern` (si possible avec l'ABI spécifiée) et comme `unsafe`.

Les pointeurs de fonction en Rust ressemblent bien plus aux références qu'aux pointeurs simples. En particulier, la validité des pointeurs de fonction ne peut pas être vérifiée directement du côté Rust. Toutefois, Rust offre deux alternatives possibles :

- l'utilisation de pointeurs de fonctions *wrappé* dans une valeur de type `Option`, accompagnée d'un test contre la valeur nulle :

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
    }
}
```

Du côté C :

C

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

- l'utilisation de pointeurs *bruts* avec une transformation `unsafe` vers un type pointeur de fonction, permettant des tests plus poussés au prix de l'ergonomie.

R42

RÈGLE - Vérification des pointeurs de fonction provenant d'une FFI

Dans un développement sécurisé en Rust, tout pointeur de fonction provenant de l'extérieur de l'écosystème Rust doit être vérifié à la frontière des FFI.

Lors d'un *binding* avec le C ou encore le C++, il n'est pas simple de garantir la validité d'un pointeur de fonction. Les foncteurs C++ ne sont pas compatibles avec le C.

7.1.4.3 Enumérations

Les valeurs (motifs de bits) valides pour une énumération donnée sont en général assez peu nombreuses par rapport à l'ensemble des valeurs qu'il est possible d'exprimer avec le même nombre de bits. Ne pas traiter correctement une valeur d'`enum` fournie par un code externe peut mener à une confusion de types et avoir de sérieuses conséquences sur la sécurité d'un programme. Malheureusement, vérifier la valeur d'une énumération aux bornes d'une FFI n'est pas une tâche triviale des deux côtés.

Du côté Rust, cette vérification consiste à utiliser un type entier dans la déclaration du bloc `extern`, un type *robuste* donc, et d'effectuer une conversion contrôlée vers le type `enum`.

Du côté externe, cela est possible uniquement si l'autre langage permet la mise en place de tests plus stricts que ceux proposés en C. C'est par exemple possible en C++ avec les `enum class`. Notons toutefois pour référence que l'ABI `extern "C"` d'une `enum class` est définie par l'implémentation et doit être vérifiée pour chaque environnement d'exécution.

R43

RECO - Non-utilisation d'`enums` Rust provenant de l'extérieur par une FFI

Dans un développement sécurisé en Rust, lors de l'interfaçage avec un langage externe, le code Rust ne doit pas accepter de valeurs provenant de l'extérieur en tant que valeur d'un type `enum`.
Les exceptions incluant des types `enum` Rust sont :

- les types opaques du langage externe dont les valeurs sont uniquement manipulées du côté Rust ;
- les types liés à des types d'énumération sûrs du côté du langage externe, comme les `enum class` de C++ par exemple.

Concernant les énumérations ne contenant aucun champ, des *crates* comme [`num_derive`] ou [`num_enum`] permettent au développeur de fournir facilement des opérations de conversions sûres depuis une valeur entière vers une énumération et peuvent être utilisées pour convertir de manière contrôlée un entier (fourni par une énumération C) vers une énumération C.

7.1.5 Types opaques

Rendre opaques des types est une bonne méthode pour augmenter la modularité dans un développement logiciel. C'est notamment une pratique assez courante dans un développement impliquant plusieurs langages de programmation.

R44

RECO - Utilisation de types Rust dédiés pour les types opaques externes

Dans un développement sécurisé en Rust, lors d'un *binding* avec des types opaques externes, des pointeurs vers des types opaques dédiés doivent être utilisés au lieu de pointeurs `c_void`.

La pratique recommandée pour récupérer des valeurs externes de type opaque est illustrée comme suit :

Rust

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

La proposition RFC 1861, non implémentée à la rédaction de ce guide, propose de faciliter cette situation en permettant de déclarer des types opaques dans des blocs `extern`.

R45

RECO - Utilisation de pointeurs vers des `structs` C/C++ pour rendre des types opaques

Dans un développement sécurisé en Rust, lors de l'interfaçage avec du C ou du C++, les valeurs de types Rust considérés comme opaques dans la partie C/C++ doivent être transformées en valeurs de type `struct` incomplet (c'est-à-dire déclaré sans définition) et être fournies avec un constructeur et un destructeur dédiés.

Un exemple d'utilisation de type opaque Rust :

Rust

```
struct Opaque {
    // (...) détails à cacher
}

#[no_mangle]
pub unsafe extern "C" fn new_opaque() -> *mut Opaque {
    catch_unwind(|| // Catch panics, see below
        Box::into_raw(Box::new(Opaque {
            // (...) 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))
        }
    ); // nécessaire seulement si `Opaque` ou un de ses champs est `Drop`
}
```

7.2 Mémoire et gestion des ressources

Les langages de programmation ont de nombreuses façons de gérer la mémoire. En résultat, il est important de savoir quel langage est responsable de la réclamation de l'espace mémoire d'une donnée lorsqu'elle est échangée entre Rust et un autre langage. Il en va de même pour d'autres types de ressources comme les descripteurs de fichiers ou les *sockets*.

Rust piste le responsable ainsi que la durée de vie des variables pour déterminer à la compilation si et quand la mémoire associée doit être libérée. Grâce au trait `Drop`, il est possible d'exploiter ce système pour récupérer toutes sortes de ressources comme des fichiers ou des accès au réseau.

Déplacer une donnée depuis Rust vers un langage signifie également abandonner de possibles réclamations de la mémoire qui lui est associée.

R46

RÈGLE - Non-utilisation de types qui implémentent `Drop` dans des FFI

Dans un développement sécurisé en Rust, le code Rust ne doit pas implémenter `Drop` pour les valeurs de types qui sont directement transmis à du code externe (c'est-à-dire ni par pointeur, ni par référence).

En fait, il est même recommandé de n'utiliser que des types qui implémentent `Copy`. Il faut noter que `*const T` est `Copy` même si `T` ne l'est pas.

Si ne pas récupérer la mémoire et les ressources est une mauvaise pratique, en termes de sécurité, utiliser de la mémoire récupérée plus d'une fois ou libérer deux fois certaines ressources peut être pire. Afin de libérer correctement une ressource une seule et unique fois, il faut savoir quel langage est responsable de la gestion de son allocation et de sa libération.

R47

RÈGLE - Identification du langage responsable de la libération des données dans les FFI

Dans un développement sécurisé en Rust, lorsqu'une donnée, quel que soit son type, est échangée par une FFI, il est nécessaire de s'assurer que :

- un seul langage est responsable de l'allocation et de la libération d'une donnée ;
- l'autre langage ne doit ni allouer, ni libérer la donnée directement, mais peut utiliser une fonction externe dédiée fournie par le langage responsable choisie.

L'identification d'un langage responsable de la gestion des données en mémoire ne suffit pas. Il reste à s'assurer de la durée de vie correcte de ces données, principalement qu'elles ne sont plus utilisées après leur libération. C'est une étape bien plus difficile. Lorsque le langage externe est responsable de la mémoire, la même approche est de fournir un *wrapper* sûr autour du type externe.

R48

RECO - Encapsulation des données externes dans un type `Drop`

Dans un développement sécurisé en Rust, toute donnée à caractère non sensible allouée et libérée du côté du langage externe doit être encapsulée dans un type implémentant `Drop`, de telle sorte que Rust fournisse l'appel automatique au destructeur Rust.

Voici un simple exemple d'encapsulation autour d'un type opaque externe :

Rust

```
/// Type Foo privé, "raw", opaque, externe
#[repr(C)]
struct RawFoo {
    _private: [u8; 0],
}

/// API C privée "raw"
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 {
    /// Création d'une valeur Foo
    pub fn new() -> Option<Foo> {
        let raw_ptr = unsafe { foo_create() };
        if raw_ptr.is_null() {
            None
        } else {
            Some(Foo(raw_ptr))
        }
    }

    /// Utilisation de 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) }
        }
    }
}
```

Attention

Parce que des `panic`s peuvent mener à ne pas exécuter la méthode `Drop::drop`, cette solution n'est pas satisfaisante pour le cas de la libération de ressources sensibles (pour effacer les données sensibles par exemple), à moins que le code soit garanti exempt de `panic` potentiel.
Pour le cas de l'effacement des données sensibles, le problème peut être géré par l'utilisation d'un *handler* de `panic`.

Lorsque le langage externe exploite des ressources allouées depuis le côté Rust, il est encore plus difficile d'offrir quelque garantie qui soit.

En C par exemple, il n'y a pas de moyen simple qui permette de vérifier que le destructeur correspondant est appelé. Il est possible d'utiliser des *callbacks* pour assurer que la libération est effectivement faite.

Le code Rust suivant est un exemple ***unsafe* du point de vue des threads** d'une API compatible avec le C qui fournit une *callback* pour assurer la libération d'une ressource :

Rust

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

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, // pour prévenir d'une réutilisation accidentelle
        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);
        // Utilisation de la mémoire du tas pour ne pas fournir de pointeur de
        // pile au code C!
        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; // prévention d'une réutilisation accidentelle
                                     // drop implicite de la `box`
        } else {
            // (...) gestion des erreurs (partie critique)
            boxed.tag = INVALID_TAG; // prévention d'une réutilisation
            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!("ne fait rien avec {:p}", cxtra);
        };
        if catch_unwind(do_it).is_err() {
            if let Some(cxtra) = cxtra.as_mut() {
                cxtra.tag = ERR_TAG;
            }
        };
    }
```

```
}
```

Un appel C compatible :

C

```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 `Panic`s et code externe

Lors de l'appel à du code Rust depuis un autre langage (par exemple, du C), le code Rust ne doit pas provoquer de `panic`. Dérouler (*unwinding*) depuis le code Rust dans du code externe résulte en un **comportement indéfini**.

R49

RÈGLE - Gestion correcte des `panic`s dans les FFI

Le code Rust appelé depuis un langage externe doit soit s'assurer que la fonction ne peut pas provoquer de `panic`, soit utiliser un mécanisme de récupération de `panic` (comme `std::panic::catch_unwind`, `std::panic::set_hook`, `#[panic_handler]`), afin d'assurer que la fonction Rust ne peut pas quitter ou retourner dans un état instable.

Il faut noter que `catch_unwind` rattrapera seulement les *unwinding* *panics* mais pas ceux provoquant un arrêt du processus.

Rust

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

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

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`

Dans le cas des programmes n'utilisant pas la bibliothèque standard Rust (`#[no_std]`), un gestionnaire de `panic` (`#[panic_handler]`) doit être défini pour la sécurité du programme. Le gestionnaire de `panic` doit être écrit avec la plus grande précaution pour garantir non seulement la sécurité, mais aussi la sûreté du programme.

Un approche alternative est de simplement s'assurer qu'il n'y a aucune utilisation de `panic!` avec la *crate* `panic-never`. Comme `no-panic`, `panic-never` repose sur une astuce au moment de l'édition de liens : le programme d'édition de liens échoue si une branche non trivialement inaccessible mène à un appel à `panic!`.

7.4 Liaison d'une bibliothèque externe à du code Rust

R50

RECO - Mise en place d'une encapsulation sûre pour les bibliothèques externes

L'interfaçage entre une bibliothèque écrite dans un autre langage et du code Rust doit être réalisé en deux parties :

- un module bas-niveau, potentiellement *caché*, qui traduit de façon très proche l'API originale en des blocs `extern` ;
- un module qui assure la sûreté mémoire et les invariants de sécurité au niveau de Rust.

Si l'API bas-niveau est exposée, cela doit être fait dans un *crate* dédiée ayant un nom de la forme `*-sys`.

La *crate* rust-bindgen peut être utilisée pour générer automatiquement la partie bas-niveau du *binding* depuis les fichiers *header* C.

7.5 Liaison entre une bibliothèque Rust et du code d'un autre langage

R51

RECO - Exposition exclusive d'API dédiée et compatible avec le C

Dans un développement sécurisé en Rust, exposer un bibliothèque Rust à un langage externe doit être uniquement fait par le biais d'une **API dédiée et compatible avec le C**.

La *crate* cbindgen peut être utilisée pour générer automatiquement les *bindings* C ou C++ pour l'API Rust compatible avec le C d'une bibliothèque Rust.

7.5.1 Exemple minimal d'une bibliothèque Rust exportée vers du C

`src/lib.rs` :

Rust

```
/// Compteur opaque
pub struct Counter(u32);

impl Counter {
    /// Crée un compteur (initialisé à 0)
    fn new() -> Self {
        Self(0)
    }
    /// Récupère la valeur courante du compteur
    fn get(&self) -> u32 {
        self.0
    }
    /// Incrémente la valeur du compteur s'il n'y a pas de dépassement
    fn incr(&mut self) -> bool {
        if let Some(n) = self.0.checked_add(1) {
            self.0 = n;
            true
        } else {
            false
        }
    }
}

// API compatible avec le C

#[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;
}
```

En utilisant cbindgen (`[cbindgen] -l c > counter.h`), il est possible de générer un *header* C cohérent, `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;
}
```

Liste des recommandations

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

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