Google Security Blog

The latest news and insights from Google on security and safety on the Internet

Deploying Rust in Existing Firmware Codebases

September 4, 2024

Posted by Ivan Lozano and Dominik Maier, Android Team

Android's use of safe-by-design principles drives our adoption of memory-safe languages like Rust, making exploitation of the OS increasingly difficult with every release. To provide a secure foundation, we're extending hardening and the use of memory-safe languages to low-level firmware (including in Trusty apps).

In this blog post, we'll show you how to gradually introduce Rust into your existing firmware, prioritizing new code and the most security-critical code. You'll see how easy it is to boost security with drop-in Rust replacements, and we'll even demonstrate how the Rust toolchain can handle specialized bare-metal targets.

Drop-in Rust replacements for C code are not a novel idea and have been used in other cases, such as librsvg's adoption of Rust which involved replacing C functions with Rust functions in-place. We seek to demonstrate that this approach is viable for firmware, providing a path to memory-safety in an efficient and effective manner.

Memory Safety for Firmware

Firmware serves as the interface between hardware and higher-level software. Due to the lack of software security mechanisms that are standard in higher-level software, vulnerabilities in firmware code can be dangerously exploited by malicious actors. Modern phones contain many coprocessors responsible for handling various operations, and each of these run their own firmware. Often, firmware consists of large legacy code bases written in memory-unsafe languages such as C or C++. Memory unsafety is the leading cause of vulnerabilities in Android, Chrome, and many other code bases.

Rust provides a memory-safe alternative to C and C++ with comparable performance and code size. Additionally it supports interoperability with C with no overhead. The Android team has discussed Rust for bare-metal firmware previously, and has developed training specifically for this domain.

Incremental Rust Adoption

Our incremental approach focusing on replacing new and highest risk existing code (for example, code which processes external untrusted input) can provide maximum security benefits with the least amount of effort. Simply writing any new code in Rust reduces the number of new vulnerabilities and over time can lead to a reduction in the number of outstanding vulnerabilities.

You can replace existing C functionality by writing a thin Rust shim that translates between an existing Rust API and the C API the codebase expects. The C API is replicated and exported by the shim for the existing codebase to link against. The shim serves as a wrapper around the Rust library API, bridging the existing C API and the Rust API. This is a common approach when rewriting or replacing existing libraries with a Rust alternative.

Challenges and Considerations

There are several challenges you need to consider before introducing Rust to your firmware codebase. In the following section we address the general state of `no_std` Rust (that is, bare-metal Rust code), how to find the right off-the-shelf crate (a rust library), porting an std crate to `no_std`, using Bindgen to produce FFI bindings, how to approach allocators and panics, and how to set up your toolchain.

The Rust Standard Library and Bare-Metal Environments

Rust's standard library consists of three crates: core, alloc, and std. The `core` crate is always available. The `alloc` crate requires an allocator for its functionality. The `std` crate assumes a full-blown operating system and is commonly not supported in bare-metal environments. A third-party crate indicates it doesn't rely on `std` through the crate-level `#![no_std]` attribute. This crate is said to be `no_std` compatible. The rest of the blog will focus on these.

Choosing a Component to Replace

When choosing a component to replace, focus on self-contained components with robust testing. Ideally, the components functionality can be provided by an open-source implementation readily available which supports bare-metal environments.

Parsers which handle standard and commonly used data formats or protocols (such as, XML or DNS) are good initial candidates. This ensures the initial effort focuses on the challenges of integrating Rust with the existing code base and build system rather than the particulars of a

complex component and simplifies testing. This approach eases introducing more Rust later on.

Choosing a Pre-Existing Crate (Rust Library)

Picking the right open-source crate (Rust library) to replace the chosen component is crucial. Things to consider are:

- Is the crate well maintained, for example, are open issues being addressed and does it use recent crate versions?
- How widely used is the crate? This may be used as a quality signal, but also important to consider in the context of using crates later on which may depend on it.
- Does the crate have acceptable documentation?
- Does it have acceptable test coverage?

Additionally, the crate should ideally be no_std compatible, meaning the standard library is either unused or can be disabled. While a wide range of no_std compatible crates exist, others do not yet support this mode of operation – in those cases, see the next section on converting a std library to no_std.

By convention, crates which optionally support no_std will provide an std feature to indicate whether the standard library should be used. Similarly, the alloc feature usually indicates using an allocator is optional.

Note: Even when a library declares #![no_std] in its source, there is no guarantee that its dependencies don't depend on std. We recommend looking through the dependency tree to ensure that all dependencies support no_std, or test whether the library compiles for a no_std target. The only way to know is currently by trying to compile the crate for a bare-metal target.

For example, one approach is to run cargo check with a bare-metal toolchain provided through rustup:

```
$ rustup target add aarch64-unknown-none
$ cargo check --target aarch64-unknown-none --no-default-features
```

Porting a std Library to no_std

If a library does not support no_std, it might still be possible to port it to a bare-metal environment – especially file format parsers and other OS agnostic workloads. Higher-level functionality such as file handling, threading, and async code may present more of a challenge. In those cases, such functionality can be hidden behind feature flags to still provide the core functionality in a no_std build.

To port a std crate to no_std (core+alloc):

- In the cargo.toml file, add a std feature, then add this std feature to the default features
- Add the following lines to the top of the lib.rs:

```
#![no_std]

#[cfg(feature = "std")]
extern crate std;
extern crate alloc;
```

Then, iteratively fix all occurring compiler errors as follows:

a. Move any use directives from std to either core or alloc.
b. Add use directives for all types that would otherwise automatically be imported by the std prelude, such as alloc::vec::Vec and alloc::string::String.
c. Hide anything that doesn't exist in core or alloc and cannot otherwise be supported in the no_std build (such as file system accesses) behind a #[cfg(feature = "std")] guard.
d. Anything that needs to interact with the embedded environment may need to be explicitly handled, such as functions for I/O. These likely need to be behind a #[cfg(not(feature = "std"))] guard.
e. Disable std for all dependencies (that is, change their definitions in Cargo.toml, if using Cargo).

This needs to be repeated for all dependencies within the crate dependency tree that do not support no_std yet.

Custom Target Architectures

There are a number of officially supported targets by the Rust compiler, however, many bare-metal targets are missing from that list. Thankfully, the Rust compiler lowers to LLVM IR and uses an internal copy of LLVM to lower to machine code. Thus, it can support any target architecture that LLVM supports by defining a custom target.

Defining a custom target requires a toolchain built with the channel set to dev or nightly. Rust's Embedonomicon has a wealth of information on this subject and should be referred to as the source of truth.

To give a quick overview, a custom target JSON file can be constructed by finding a similar supported target and dumping the JSON representation:

```
$ rustc --print target-list
[...]
armv7a-none-eabi
[...]

$ rustc -Z unstable-options --print target-spec-json --target armv7a-none-eabi
```

This will print out a target JSON that looks something like:

```
$ rustc --print target-spec-json -Z unstable-options --target=armv7a-none-eabi
{
  "abi": "eabi",
  "arch": "arm",
  "c-enum-min-bits": 8,
  "crt-objects-fallback": "false",
  "data-layout": "e-m:e-p:32:32-Fi8-i64:64-v128:64:128-a:0:32-n32-S64",
  [...]
}
```

This output can provide a starting point for defining your target. Of particular note, the data-layout field is defined in the LLVM documentation.

Once the target is defined, libcore and liballoc (and libstd, if applicable) must be built from source for the newly defined target. If using Cargo, building with -Z build-std accomplishes this, indicating that these libraries should be built from source for your target along with your crate module:

```
# set build-std to the list of libraries needed
cargo build -Z build-std=core,alloc --target my_target.json
```

Building Rust With LLVM Prebuilts

If the bare-metal architecture is not supported by the LLVM bundled internal to the Rust toolchain, a custom Rust toolchain can be produced with any LLVM prebuilts that support the target.

The instructions for building a Rust toolchain can be found in detail in the Rust Compiler Developer Guide. In the config.toml, llvm-config must be set to the path of the LLVM prebuilts.

You can find the latest Rust Toolchain supported by a particular version of LLVM by checking the release notes and looking for releases which bump up the minimum supported LLVM version. For example, Rust 1.76 bumped the minimum LLVM to 16 and 1.73 bumped the minimum LLVM to 15. That means with LLVM15 prebuilts, the latest Rust toolchain that can be built is 1.75.

Creating a Drop-In Rust Shim

To create a drop-in replacement for the C/C++ function or API being replaced, the shim needs two things: it must provide the same API as the replaced library and it must know how to run in the firmware's bare-metal environment.

Exposing the Same API

The first is achieved by defining a Rust FFI interface with the same function signatures.

We try to keep the amount of unsafe Rust as minimal as possible by putting the actual implementation in a safe function and exposing a thin wrapper type around.

For example, the FreeRTOS coreJSON example includes a JSON_Validate C function with the following signature:

```
JSONStatus_t JSON_Validate( const char * buf, size_t max );
```

We can write a shim in Rust between it and the memory safe serde_json crate to expose the C function signature. We try to keep the unsafe code to a minimum and call through to a safe function early:

```
#[no_mangle]
pub unsafe extern "C" fn JSON_Validate(buf: *const c_char, len: usize) ->
JSONStatus_t {
    if buf.is_null() {
        JSONStatus::JSONNullParameter as _
    } else if len == 0 {
        JSONStatus::JSONBadParameter as _
    } else {
        json_validate(slice_from_raw_parts(buf as _, len).as_ref().unwrap()) as _
    }
}

// No more unsafe code in here.
fn json_validate(buf: &[u8]) -> JSONStatus {
    if serde_json::from_slice::<Value>(buf).is_ok() {
        JSONStatus::JSONSuccess
    } else {
        ILLEGAL_DOC
    }
}
```

Note: This is a very simple example. For a highly resource constrained target, you can avoid alloc and use serde_json_core, which has even lower overhead but requires pre-defining the JSON structure so it can be allocated on the stack.

For further details on how to create an FFI interface, the Rustinomicon covers this topic extensively.

Calling Back to C/C++ Code

In order for any Rust component to be functional within a C-based firmware, it will need to call back into the C code for things such as allocations or logging. Thankfully, there are a variety of tools available which automatically generate Rust FFI bindings to C. That way, C functions can easily be invoked from Rust.

The standard means of doing this is with the Bindgen tool. You can use Bindgen to parse all relevant C headers that define the functions Rust needs to call into. It's important to invoke Bindgen with the same CFLAGS as the code in question is built with, to ensure that the bindings are generated correctly.

Experimental support for producing bindings to static inline functions is also available.

Hooking Up The Firmware's Bare-Metal Environment

Next we need to hook up Rust panic handlers, global allocators, and critical section handlers to the existing code base. This requires producing definitions for each of these which call into the existing firmware C functions.

The Rust panic handler must be defined to handle unexpected states or failed assertions. A custom panic handler can be defined via the panic_handler attribute. This is specific to the target and should, in most cases, either point to an abort function for the current task/process, or a panic function provided by the environment.

If an allocator is available in the firmware and the crate relies on the alloc crate, the Rust allocator can be hooked up by defining a global allocator implementing GlobalAlloc.

If the crate in question relies on concurrency, critical sections will need to be handled. Rust's core or alloc crates do not directly provide a means for defining this, however the critical_section crate is commonly used to handle this functionality for a number of architectures, and can be extended to support more.

It can be useful to hook up functions for logging as well. Simple wrappers around the firmware's existing logging functions can expose these to Rust and be used in place of print or eprint and the like. A convenient option is to implement the Log trait.

Fallible Allocations and `alloc`

Rusts alloc crate normally assumes that allocations are infallible (that is, memory allocations won't fail). However due to memory constraints this isn't true in most bare-metal environments. Under normal circumstances Rust panics and/or aborts when an allocation fails; this may be acceptable behavior for some bare-metal environments, in which case there are no further considerations when using alloc.

If there's a clear justification or requirement for fallible allocations however, additional effort is required to ensure that either allocations can't fail or that failures are handled.

One approach is to use a crate that provides statically allocated fallible collections, such as the heapless crate, or dynamic fallible allocations like fallible_vec. Another is to exclusively use try_* methods such as Vec::try_reserve, which check if the allocation is possible.

Rust is in the process of formalizing better support for fallible allocations, with an experimental allocator in nightly allowing failed allocations to be handled by the implementation. There is also the unstable cfg flag for alloc called no_global_oom_handling which removes the infallible methods, ensuring they are not used.

Build Optimizations

Building the Rust library with LTO is necessary to optimize for code size. The existing C/C++ code base does not need to be built with LTO when passing `-C lto=true` to `rustc`. Additionally, setting `-C codegen-unit=1` results in further optimizations in addition to reproducibility.

If using Cargo to build, the following Cargo.toml settings are recommended to reduce the output library size:

```
[profile.release]
panic = "abort"
lto = true
codegen-units = 1
strip = "symbols"

# opt-level "z" may produce better results in some circumstances
opt-level = "s"
```

Passing the `-Z remap-cwd-prefix=.` flag to `rustc` or to Cargo via the `RUSTFLAGS` env var when building with Cargo to strip cwd path strings.

In terms of performance, Rust demonstrates similar performance to C. The most relevant example may be the Rust binder Linux kernel driver, which found "that Rust binder has similar performance to C binder".

When linking LTO'd Rust staticlibs together with C/C++, it's recommended to ensure a single Rust staticlib ends up in the final linkage, otherwise there may be duplicate symbol errors when linking. This may mean combining multiple Rust shims into a single static library by re-exporting them from a wrapper module.

Memory Safety for Firmware, Today

Using the process outlined in this blog post, You can begin to introduce Rust into large legacy firmware code bases immediately. Replacing security critical components with off-the-shelf open-source memory-safe implementations and developing new features in a memory safe language will lead to fewer critical vulnerabilities while also providing an improved developer experience.

Special thanks to our colleagues who have supported and contributed to these efforts: Roger Piqueras Jover, Stephan Chen, Gil Cukierman, Andrew Walbran, and Erik Gilling

Labels: android , android security , rust

No comments :

Post a Comment