Bringing Rust to Safety-Critical Systems in Space

Lukas Seidel (*Binarly, TU Berlin*) & Julian Beier (*TU Berlin*)

IEEE Security for Space Systems (3S) 2024
28/05/2024

Software Challenges in Aerospace

- Legacy firmware is traditionally written in C / C++
  - Inherently easy to make critical mistakes

- Security was neglected for a long time
  - Security-By-Obscurity was the default
  - No dedicated security testing

Software Challenges in Aerospace (cont.)

- Embedded chips / firmware lack common mitigations
  - Bare-metal firmware has no ASLR
  - Common Cortex-M cores have no MMU ⟹ no NX stack

- At the same time: firmware is often safety-critical and every security issue is also a safety issue!
- Memory corruptions leading to remote takeover have been recently demonstrated


RUST IN
SPACE

Rust

- Strong type system and enforced memory safety guarantees
- The only language supported in the Linux Kernel besides C and assembly

- Recently introduced to Android development
  - Less memory-unsafe code ⟹ fewer vulnerabilities
    - Memory safety issues dropped from 76% to 35% of Android’s total vulns

Embedded Rust

- Spacecraft usually integrate multiple embedded systems
- For adaptation, practitioners need platform support
  - [embedded-hal]: unified Hardware Abstraction Layer to access low-level functionality
  - Over 200 drivers for sensors, actuators etc. developed on top of it and openly available
  - Support for ~50 different MCUs
- Cortex-M support especially mature
- Crates improving security: [flip-link] to catch stack overflows

The Safety-Critical Ecosystem

Programming Guidelines

- Software safety standards as check lists, e.g.,
  - DO-178C (aerospace systems)
  - ISO 26262 (automotive)
  - IEC 61508 (industrial environments)

- Safety standards require external coding standard to qualify against but do not provide their own

Programming Guidelines (cont.)

- MISRA-C
  - Guidelines for safe, secure, and reliable embedded systems
  - Initially for automotive, now widely used. e.g, in NASA's Jet Propulsion Laboratory

- High Assurance Rust Initiative
  - Introduced in 2022, based on MISRA-C
  - Covers data structures, static program verification, differential fuzzing, and deductive verification

Standardization and Qualification

- Qualified open-source Rust compiler: ferrocene by Ferrous Systems
  - ISO 26262 (ASIL D) and IEC 61508 (SIL 4)
  - C and C++, although having qualified language specs, also don’t have qualified standard libs
  - Upcoming: Qualification of [core] and adding DO-178C

- AdaCore GNAT Pro for Rust
  - Interoperability with Ada ecosystem
  - Offering long-term support for compilers, debuggers etc.

Case Studies

CS1: libCSP

- Cubesat Space Protocol: network protocol for standardized communication in distributed embedded systems

- Most popular satellite protocol on Github:
  - 40 contributors, 2000 commits and 400 stars
- Used in
  - ESA OPS-SAT (2019)
  - DLR CubeL (2020)
  - …

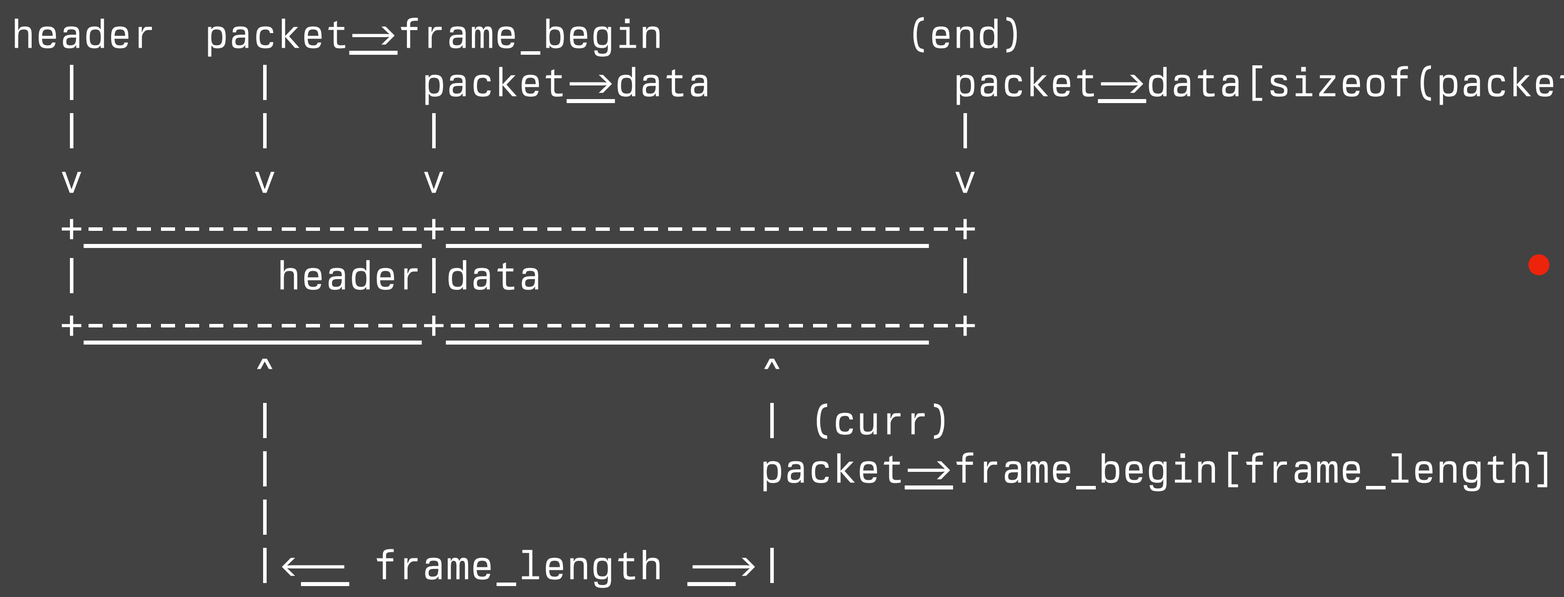
libCSP: Security Analysis

- zero-copy buffer- and queueing system
  - After initial enqueue operation, no more raw memory operations on incoming packet
  - ⟹ reduced attack surface
    - interface backend implementations are the most critical components
- Manual analysis and fuzzing (LibAFL) of CAN interface

libCSP: Security Analysis (cont.)

- Off-by-one in csp_can2_rx leading to buffer overflow

```
/* Check for overflow. The frame input + dlc must not exceed the end of the packet data field */
if (&packet->frame_begin[packet->frame_length] + dlc > &packet->data[sizeof(packet->data)]) {
if (&packet->frame_begin[packet->frame_length] + dlc >= &packet->data[sizeof(packet->data)]) {
    csp_dbg_can_errno = CSP_DBG_CAN_ERR_RX_OVF;
    iface->frame++;
    csp_can_pbuf_free(ifdata, packet, 1, task_woken);
```



- Fixed 7 memory / safety issues in total

csp_if_can.c

```c
// point at Rust implementation for `csp_can2_rx`
#include "libcsp_rs.h"

extern int csp_can2_rx(csp_iface_t * iface, uint32_t id, const uint8_t * data, uint8_t dlc, int *
task_woken);

// remove local C implementation
// int csp_can2_rx(csp_iface_t * iface, uint32_t id, const uint8_t * data, uint8_t dlc, int * task_woken) {
..
```

CS2: Partial Rewriting

rustc-link-lib=libcsp

libcsp_ffi.rs

libcsp.so

compile

libcsp_rs.h

lib.rs
(static lib)

```rust
#[no_mangle]
pub extern "C" fn csp_can2_rx(
    iface: *mut libcsp_ffi::csp_iface_t,
    id: u32,
    data: *const u8,
    data_length: u8,
    task_woken: *mut i32,
) -> CspError
..
```

CS3: A New Target Platform

- Idea: extend Rust’s applicability in space

- Solution: Compiler Target for Bare-Metal PowerPC32!
  - So far, Rust only compiles to PPC Linux
  - PPC is a popular architecture for radiant-resistant CPUs
    - RAD750 used in Mars Curiosity Rover and James Webb Telescope

Recommendations

R01: Gradually incorporate Rust

R02: Use qualified toolchains

R03: Utilize the existing embedded Rust ecosystem

R04: Employ dedicated security testing

Thank you!
Questions?