CHERITech'26
CONFERENCE

CHERI

12-13 Nov 2026
Munich, DE

Writing applications for CHERIoT in Rust

Edoardo Marangoni
Compiler Engineer
SCI Semiconductor

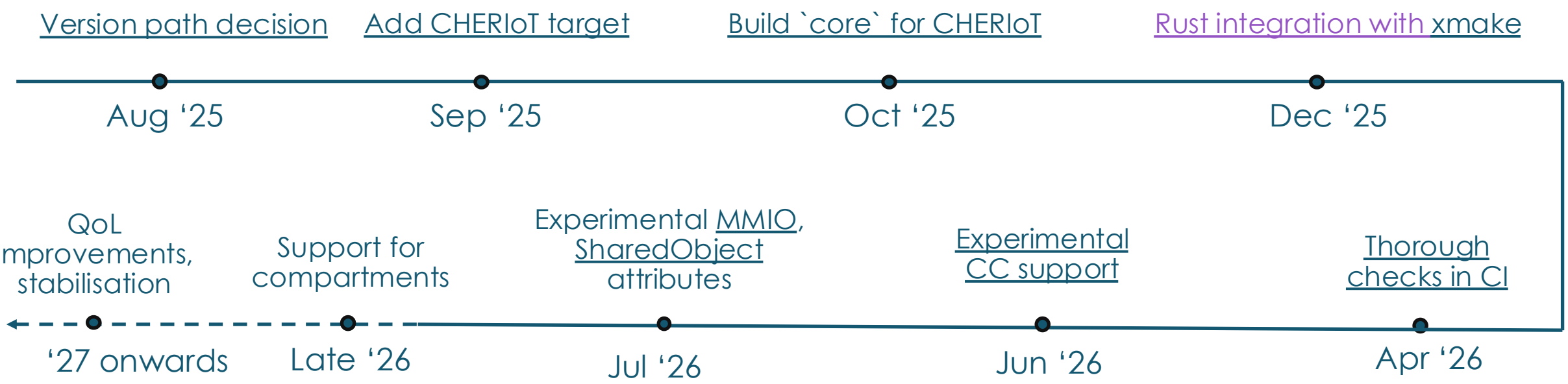


Located at semi | SEMICON EUROPA

Agenda

- Rust on CHERIoT: a short recap
- CHERI-specific features in Rust
  - MMIO-bound objects
  - Sealed objects
  - Compartments
  - New features on the horizon

15 months of efforts: Rust works on CHERIoT!



> Rust provides compile-time guarantees for safe code, whereas CHERI provides runtime guarantees for every fragment of unsafe code

Uni of Kent's Rust on Morello team

> CHERIoT's compartmentalisation model allows fearless code reuse, with the assurance that third-party (C or actively malicious Rust) software cannot compromise the guarantees of your Rust code
>
> *David Chisnall*

Current status

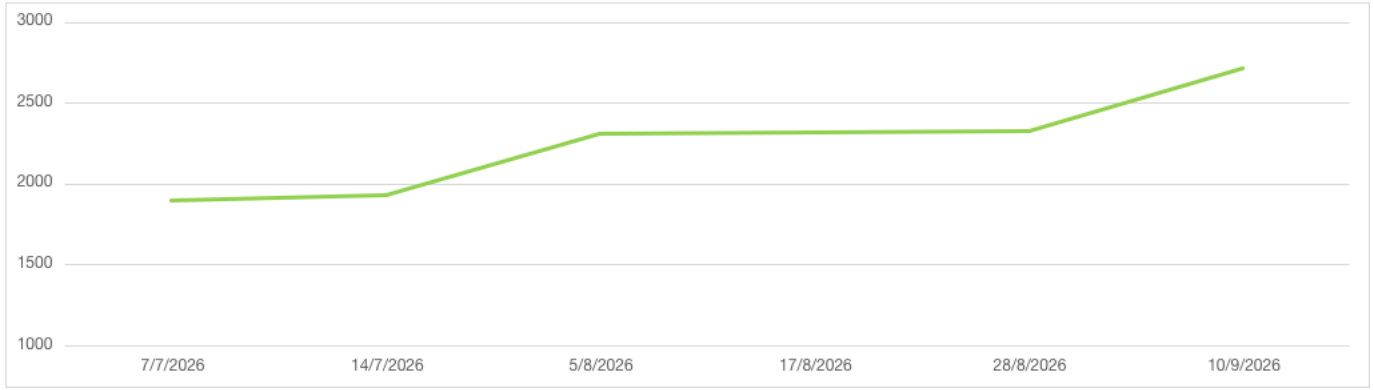
What *should* work

- The `core` library
- The `alloc` library
- Third-party dependencies via `cargo`
- MMIO-bound objects
- Shared objects
- **Manual** definition of shared libraries and compartments
- Static sealed objects

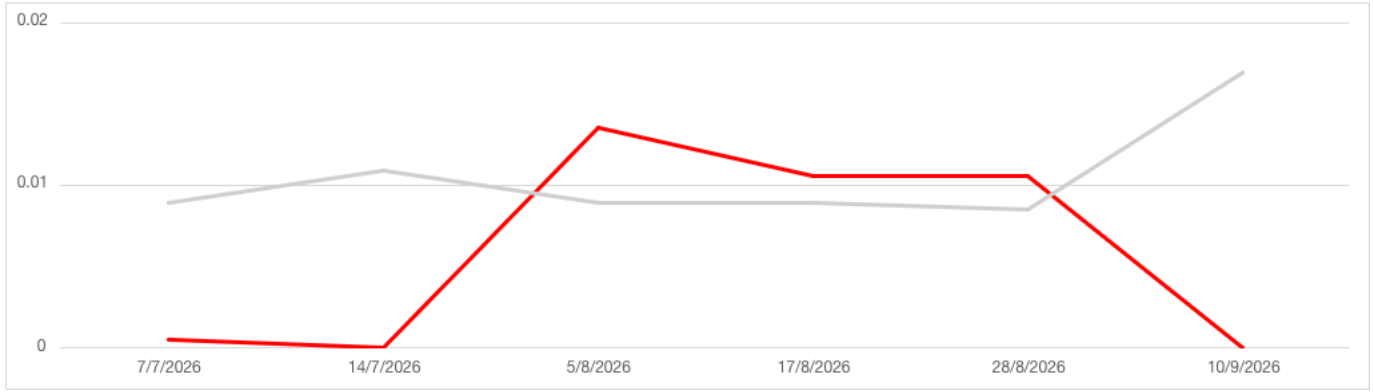
What doesn't work yet

- Declarative definition of compartments via Cargo.toml, general cargo interop with compartments
- Debug information for vtables
- Also see the issue tracker, and please report any error you may find!

Coretests: more than 2.5k tests passing



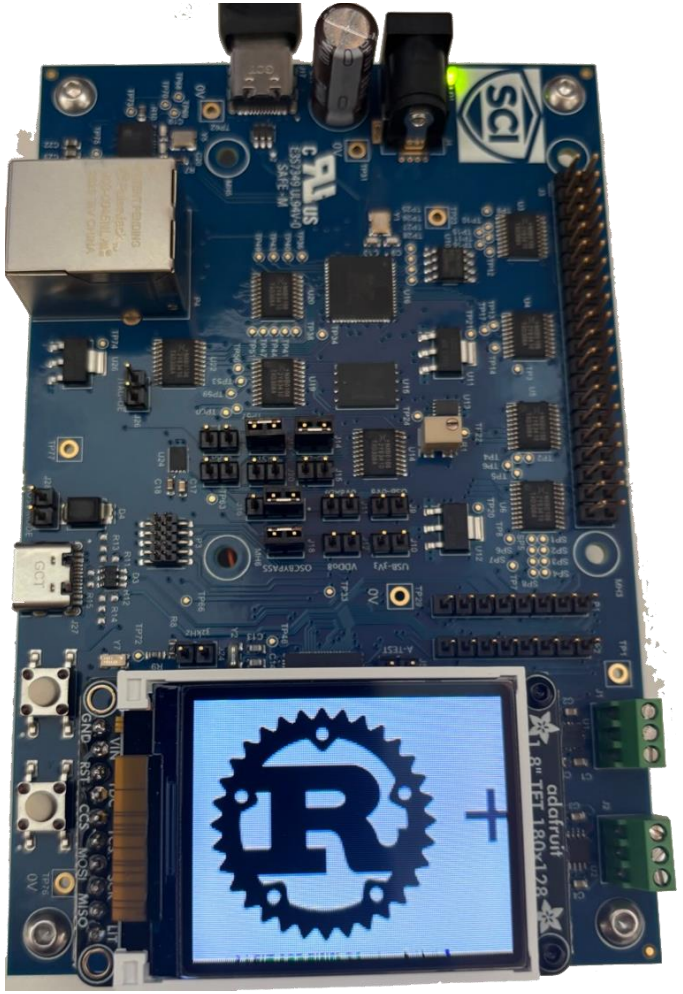
Total passing tests over time



Failing and ignored tests over total, over time

Synthesizing valid MMIO capabilities

- On CHERI systems, we can't cast integers to pointers
- How can we get a pointer to an MMIO-bound area of memory?
- This mechanism is fundamental to use peripherals!



Synthesizing valid MMIO capabilities

```
/// Get the UART #0 from the board (address 0x80100000).
pub unsafe fn uart0() -> Uart<'static> {
        #[allow(static_mut_refs)]
        unsafe {
                Uart {
                        ptr: core::mem::transmute(0x80100000),
                }
        }
}
```

```
E0512 - cannot transmute between
types of different sizes, or
dependently-sized types
```

Synthesizing valid MMIO capabilities

```
/// Get the UART #0 from the board (address 0x80100000).
pub unsafe fn uart0() -> Uart<'static> {
        unsafe extern "Rust" {
                #[cheriot_mmio(name = "uart", permissions = "RW")]
                pub static mut UART0: InnerUart;
        }

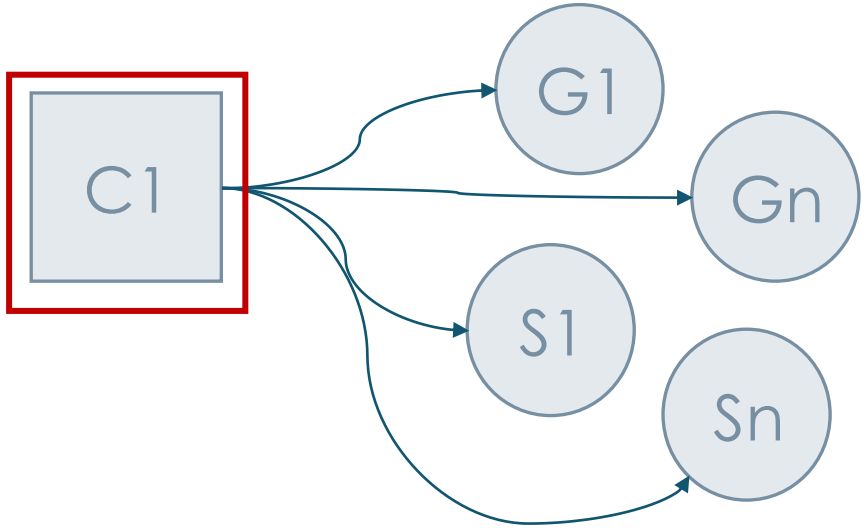
        #[allow(static_mut_refs)]
        unsafe {
                Uart {
                        ptr: &mut UART0,
                }
        }
}
```

A pure-Rust LCD driver for Iceni



Compartments

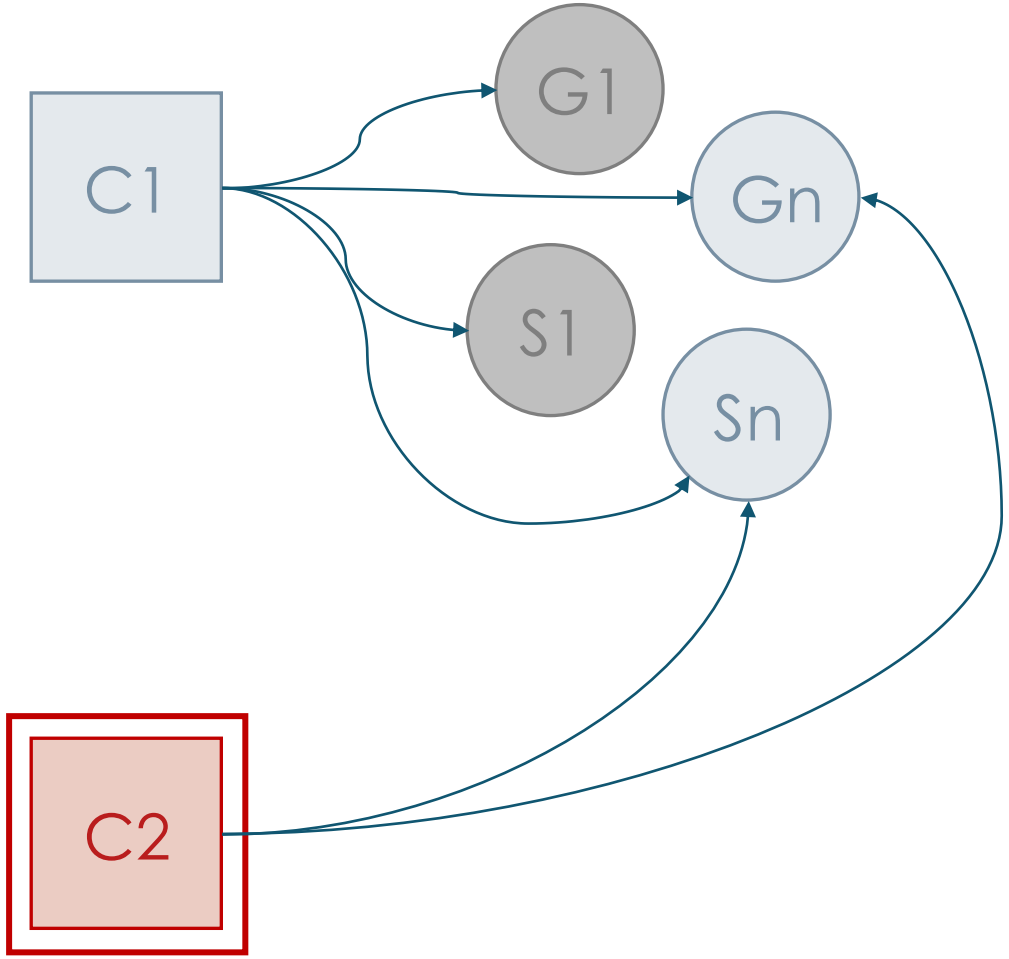
- A way to define security boundaries
- Compartments isolate code and data
- Cross-compartment calls prevent memory leaks
- A fault in a compartment doesn't take down the whole firmware with it



C2

Compartments

- A way to define security boundaries
- Compartments isolate code and data
- Cross-compartment calls prevent memory leaks
- A fault in a compartment doesn't take down the whole firmware with it



Compartments in Rust

```rust
#![no_std]
#![feature(cheriot_attributes)]
#![cheri_compartment = "my_app"]

pub fn call_rust() -> core::ffi::c_int {
        log::record("hello", "from rust");
        core::ffi::c_int::from(0)
}
```

```rust
#![no_std]
#![feature(cheriot_attributes)]
#![cheri_compartment = "log4rust"]

pub fn record(key: &str, value: &str) {
        println!("[log] {key} -> {value}");
}
```

Tells the Rust compiler which compartment this code lives in

The compiler knows that log::record lives in another compartment, and emits the correct cross-compartment call instructions

Compartments in Rust

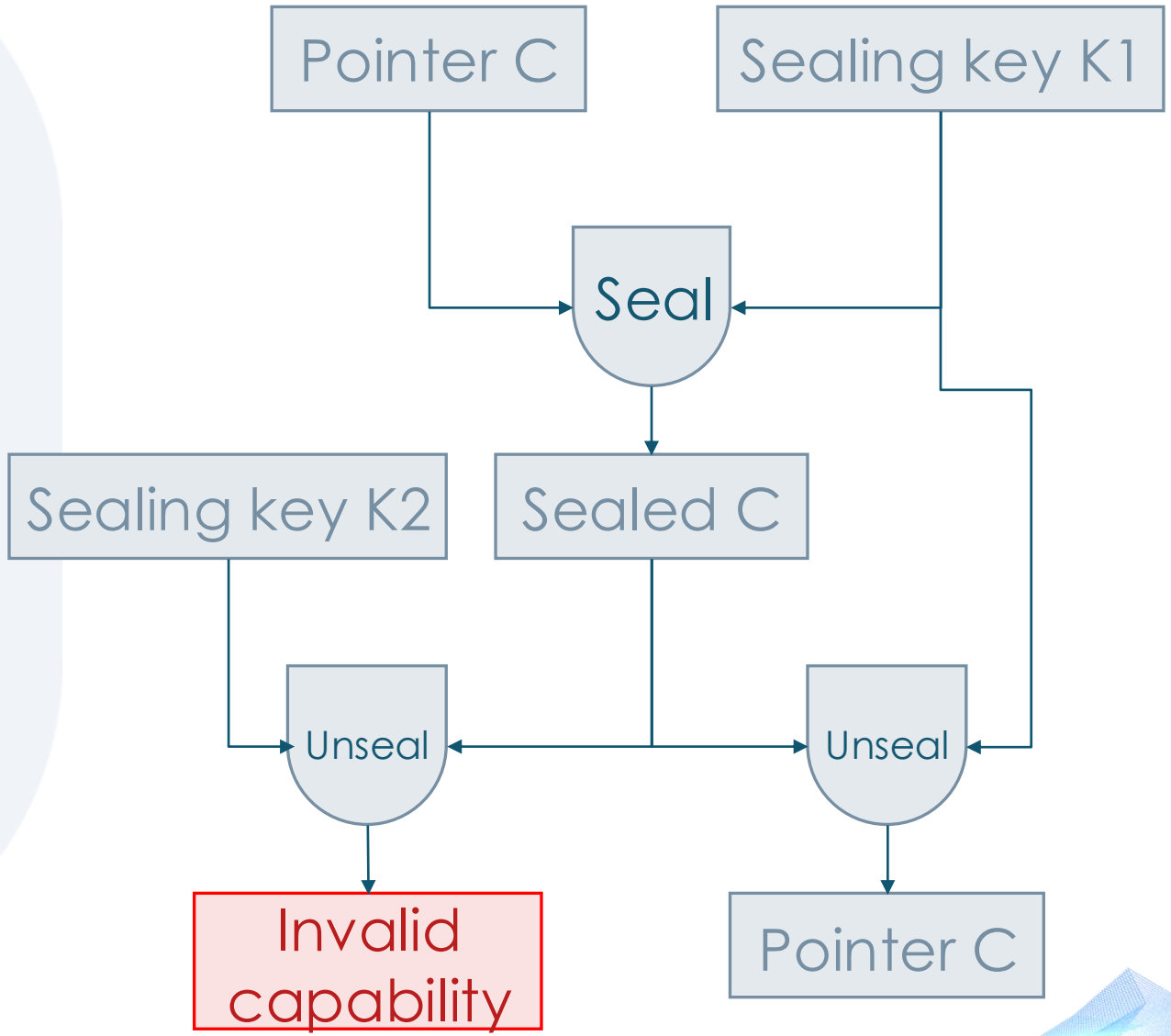
- The current support does not let users ascribe a compartment to a dependency without touching its source code
- It requires more interoperability between cargo and the CHERIoT build system
- It is necessary for ease-of-use of compartments!

```
[package]
name = "my-app"
version = "0.1.0"
edition = "2024"

[dependencies.log]
version = "0.1.0"
path = "../log"
compartment = "logger"
```

Sealed capabilities

- A mechanism for type-safe opaque pointers
- In CHERIoT, it can be hardware or software (virtualized)
- Hardware: requires access to CHERI intrinsics
- Software: requires ad-hoc types, annotations and API



Sealed capabilities in Rust

```rust
#[repr(transparent)]
pub struct Sealed<K, T> {
        sealed: *mut T,
        key: Phantom<K>
}

impl<K, T> Sealed<K, T> {
        pub fn seal(value: T) -> Self {
                …
        }

        pub fn try_unseal_as_ref_mut(&mut self) -> Result<&mut T> {
                …
        }
}
```

Sealed capabilities in Rust

```
#![cheri_compartment = "monotonic"]

struct CounterKey;

pub type MonotonicCounterState = core::sync::atomic::AtomicU16;

pub struct MonotonicCounter(Sealed<CounterKey, MonotonicCounterState>);

impl MonotonicCounter {
        pub fn new(value: u16) -> Self {
                Self(Sealed::<CounterKey, MonotonicCounterState>::seal(value))
        }

        pub fn increment(self) -> Result<u16> {
                match self.0.try_unseal_as_ref_mut() {
                        Ok(x) => Ok(x.fetch_add(1, Ordering::Relaxed)),
                        Err(e) => err!(e)
                }
        }
}
```

Sealed capabilities in Rust

```
#![cheri_compartment = ”runner"]

pub fn run() {
        let valid_counter = MonotonicCounter::new(0);
        let invalid_counter_state = AtomicU16::new(0);
        let invalid_counter = transmute::<_, MonotonicCounter>(&invalid_counter_state);
        assert_eq!(valid_counter.increment(), Ok(0));
        assert_eq!(invalid_counter.increment(), Err(_));
}
```

On the horizon: sub-object bounds

- When you get a pointer to an allo… it has the bounc…
- Pointe… alloc h…
- This co… effects sometimes
- It can be turned off!

CVE-2026-41678 Detail

Description

rust-openssl provides OpenSSL bindings for the Rust programming language. From to before 0.10.78, aes::unwrap_key() contains an incorrect assertion: it checks that out.len() + 8 <= in_.len(), but this condition is reversed. The intended invariant is out.len() >= in_.len() - 8, ensuring the output buffer is large enough. Because of the inverted check, the function only accepts buffers at or below the minimum required size and rejects larger ones. If a smaller buffer is provided the function will write past the end of out by in_.len() - 8 - out.len() bytes, causing an out-of-bounds write from a safe public function. This vulnerability is fixed in 0.10.78.

```
…stuff,

…oreStuff::new(..);
…ey);
```

/

The bounds cover the full alloc!

On the horizon: sub-object bounds

- When you get a pointer to an allocation, it has the bounds of the full allocation
- Pointers to offsets within the alloc have the full bounds
- This can have undesired effects sometimes
- It can be turned off!

```
#[cheriot_subobject_bounds]
struct KeyAndMoreStuff {
          key: [u8;32],
          more_stuff: MoreStuff,
}

let key_and_more = KeyAndMoreStuff::new(..);
unwrap_key(&key_and_more.key);
```

The bounds cover the size of the key field only!

On the horizon: safer unsafe

- CHERI systems can enforce at the hardware level invariants that Rust computes at the type level
- We can leverage the types to make unsafe blocks safer!

```
unsafe extern "C" {
    fn do_it(val: *const u8);
}

fn run() -> u8 {
    let x = 0u8;
    unsafe {
        do_it(&x as *const u8);
    }
    x
}
```

```
void do_it(char* ptr) {
            *ptr = 10;
}
```

On the horizon: safer unsafe

- CHERI systems can enforce at the hardware level invariants that Rust computes at the type level
- We can leverage the types to make unsafe blocks safer!

```
// -Ccheriot-strip-permissions
unsafe extern "C" {
 fn do_it(val: *const u8);
}

fn run() -> u8 {
 let x = 0u8;
 unsafe {
  let arg = &x as *const u8;
  let ro_arg = perms_and(arg, Perms::Write);
  do_it(ro_arg);
 }
 x
}
```

```
void do_it(char* ptr) {
          *ptr = 10;
}
```

CHERIException: pointer is read-only!

Conclusions



CHERIoT-Platform/cheri-rust



- Recently added features let you write non-trivial firmwares for CHERIoT in pure Rust
- We plan to add QoL improvements and new security features
- Try it out, and please file issues if you encounter any problems!

CHERITech'26 CONFERENCE

12-13 Nov 2026
Munich, DE

Located at semi | SEMICON EUROPA

CHERI

THANK YOU

Contact edoardo.marangoni@cheri-alliance.org

Web https://rust.cheriot.org/

CHERI™ and the CHERI logo are protected trademarks. Use of these marks is subject to the CHERI Alliance Trademark Policy – https://cheri-alliance.org/legal-and-policies/trademark/

This work © 2026 by CHERI Alliance is licensed under CC BY-SA 4.0 (Creative Commons Attribution-ShareAlike 4.0 International) – https://creativecommons.org/licenses/by-sa/4.0/