

SINGAM TECH · SECURITY ARCHITECTURE

Why Rust's Allocator API Has Been Nightly for a Decade

...and what it means for AI-generated code

THE PROBLEM

One Keyword, Too Many Kinds of Risk

unsafe means one precise thing: the compiler can't prove the absence of undefined behavior. But it's Rust's **only** machine-checked "needs scrutiny" flag — so it ends up covering:

- Memory safety violations
- Custom allocation behavior
- Uncontrolled I/O
- Raw-pointer access

CASE STUDY

allocator_api: RFC 1398

2015	RFC 1398 proposes <code>allocator_api</code> — <code>Vec<T></code> , <code>Box<T></code> generic over a custom allocator.
2015-2026	Tracking issue rust-lang/rust#32838 stays open. Blocking question never resolves.
Today	Still nightly-only. One trait, once stabilized, is permanent — so the whole feature waits.

The blocking question: should `Allocator::allocate` be `unsafe fn` or `safe fn`?

Rust → Rusty

Ten-plus years nightly-only, blocked on one yes/no question about one keyword. Not a knock on Rust's core safety guarantees — a knock on asking a single binary flag to settle debates it was never built for.

WHY IT MATTERS NOW

The AI Programming World

AI assistants draft a growing share of production Rust. When a task needs raw performance, FFI, or systems control, the only lever stable Rust gives them is `unsafe`.

That block conflates a raw pointer, a custom allocator, and uncontrolled I/O into **one yes/no signal** — right as review time per line is going down and code volume is going up.

Review capacity, not generation speed, is the actual bottleneck.

THE ORTHOGONAL FIX

capability-attr

Sidestep the stalled debate instead of waiting for it. Stable Rust, compile-time, today:

```
#[capability(alloc(pool = "arena"), io(none), ptr(none))]  
fn build_response(req: &Request) -> Response {  
    // Compiler-enforced: only "arena" allocation.  
    // Any I/O or raw pointer here is a compile error.  
    ...  
}
```

THE ORTHOGONAL FIX

sensitive-ifc

The same move, one layer up — for data classification instead of side-effect scope.

```
let ssn: Sensitive<String, Restricted> = Sensitive::new(raw);  
  
// error[E0277]: Sensitive<String, Restricted>  
// does not implement Display  
println!("{}", ssn);
```

A compile error, not a code-review hope.

Where This Ends

Compile-time (this project)

Catches: a function exceeding its declared alloc/io/ptr scope — in code built with the attribute applied.

Can't see: a compromised binary, a dependency that skipped the attribute, any other process on the host.

Runtime, OS-enforced (AppArmor / SELinux)

Catches: what any process actually does at the syscall boundary — regardless of source language.

Can't see: which internal function allocated what — no notion of intra-process scope.

SAY IT PLAINLY

Not a Replacement. A Second Layer.

Adopting capability-attr, sensitive-ifc, taint-check, taint-generate or taint-refactor should never mean dropping an AppArmor or SELinux profile. These tools check alloc/io/ptr scope, sensitive-data flow and taint at compile time — they are not a sandbox. Keep a profile for the resulting binary, exactly as you would without them.

THE NEXT STEP

From Verify to Generate

capability-attr and taint-check only ever verify an attribute a human already wrote — they never decide where one belongs, or fix what they find. Two new tools go further.

Same posture, more automation: review before you trust, at every step.

PHASE 2, CONTINUED

taint-check Goes Crate-Wide

The AST walk used to see one module's own direct children. Now it follows `mod foo;` declarations across an entire crate and tracks taint through interprocedural, `let-bound` calls — not just simple assignments.

```
taint-check --crate src/lib.rs
```

AUTO-WRITE THE ANNOTATIONS

taint-generate

Capability attributes: deterministic. Same body inspector, writes what it actually found — not a guess.

Taint attributes: heuristic. No way to know a parameter is a password from its body alone, so this matches names against a keyword list — and `handle_login` gets flagged as a sink candidate, because "login" contains "log".

Review every generated taint attribute before trusting it.

READ THIS FIRST

taint-refactor: The Riskiest Tool Here

Given one violation, re-scans the whole crate for every occurrence of the same (`label`, `sink`) pattern and inserts a placeholder sanitizer at each call site.

- The placeholder is a naive, fixed-string redaction
- Wrapping a value can change its type — confirmed directly
- Nested `mod { ... }` violations are skipped, reported, not guessed at

SHIP THE FIX, SHOW THE WORK

A Bug We Found Along the Way

Verifying taint-refactor's own output against taint-check surfaced a genuine, pre-existing gap: `sink(sanitizer(tainted))`, called directly with no intermediate `let`, was still flagged as a violation.

`is_directly_sanitized`: does a sink's argument, after unwrapping `&/(...)`, directly call a registered sanitizer? Fixed narrowly — the broad conservative scan is untouched elsewhere.

TRY IT

Open Source, Apache-2.0

```
cargo add rusty-capability-attr  
cargo add rusty-sensitive-ifc  
cargo add rusty-taint-check-macros  
cargo install rusty-taint-check  
cargo install rusty-taint-generate  
cargo install rusty-taint-refactor
```

Published on crates.io. RFCs and architecture decision records in the repo alongside the source.

<https://github.com/2rusty-lang/rusty-lang>

SINGAM TECH

Guardrails for the AI-Era Codebase

Compile-time and runtime, together.
singamtech.com/contact