Software Assurance Tips

A product of the Software Assurance Tips Team[3]

Jon Hood

Monday 12th May, 2025

1 Leaking Through the Cracks: Rust's Soft Memory Shell

Updated Friday 9th May, 2025

Last year, Cybersecurity leadership in the U.S. encouraged organizations to use memory-safe programming languages. I criticized the blanket recommendation[2] as a partial solution for a more serious underlying concern in developer training, citing the problems faced with legacy Ada code[1]. Recently, I have had the privilege of sitting through lengthy meetings extoling the virtues of memory-safe languages and creating policies that enumerate the languages that developers must use for certain contracts. In spite of all the warnings, it looks like certain government agencies are pushing ahead with the recommendation.

But is requiring a memory-safe language the panacea to memory safety? Certainly not! Heap inspection, logical errors caused by algorithm complexity, and memory leaks are not areas of memory safety guaranteed by Rust. After appreciated and constructive criticism from my last article focusing on Ada, I have been politely asked to demonstrate memory concerns that should be considered in Rust.

Rust Background

In "safe" code (and most of the development in Rust is safe by default unless you use the `unsafe` keyword), many of the memory safety concerns have very strong mitigations. At compile time, the Rust borrow checker will prevent most dangling pointers, use-after-free errors, memory-related TOCTOU and data race issues, invalid memory accesses, and more. At runtime, additional checks will panic in ways that can be handled appropriately.

What doesn't get caught, however, are the profundity of memory issues related to data confidentiality (heap inspections), integrity (logical errors due to algorithm complexity), and availability (memory leaks and DoS attacks) that plague safety-critical systems today.

A Demonstration of Memory Leaks

Consider the intentional memory leak in Listing 1.

```
use std::rc::Rc;
use std::cell::RefCell;
use std::thread;
use std::time::Duration;

#[derive(Debug)]
struct Node {
    value: i32,
    next: RefCell<Option<Rc<Node>>>,
}

fn create_and_leak(n: i32) {
    let a = Rc::new(Node { value: n + 1, next: RefCell::new(None) });
    let b = Rc::new(Node { value: n + 2, next: RefCell::new(None) });

    a.next.borrow_mut().replace(Rc::clone(&b));
    b.next.borrow_mut().replace(Rc::clone(&a));

    println!("Leaky cycle {}.", a.value);
    println!("Reference counts: a = {}, b = {}.", Rc::strong_count(&a), Rc::strong_count(&b));
    println!("Values: a = {}, b = {}.", a.value, b.value);

    // 'a' and 'b' go out of scope but are not deallocated.
}
```

```
fn main() {
    println!("Starting the main loop...");
    for i in 0..2 {
        create_and_leak(i);
        thread::sleep(Duration::from_secs(1));
    }
}
```

Listing 1: leaky.rs: Intentional Memory Leak

The code can be compiled, executed, and run through the tool `heaptrack` on the command line as shown in Listing 2.

```
$ rustc leaky.rs
$ heaptrack ./leaky
heaptrack output will be written to "./heaptrack.leaky.10245.zst"
starting application, this might take some time...
Starting the main loop...
Leaky cycle 1.
Reference counts: a = 2, b = 2.
Values: a = 1, b = 2.
NOTE: heaptrack detected DEBUGINFOD_URLS but will disable it to prevent
unintended network delays during recording
If you really want to use DEBUGINFOD, export HEAPTRACK_ENABLE_DEBUGINFOD=1
Leaky cycle 2.
Reference counts: a = 2, b = 2.
Values: a = 2, b = 3.
heaptrack stats:
        allocations:            19
        leaked allocations:     4
        temporary allocations:  1
Heaptrack finished! Now run the following to investigate the data:

  heaptrack --analyze "./heaptrack.leaky.10245.zst"
$ heaptrack_print heaptrack.leaky.10245.zst | grep "^total memory"
total memory leaked: 160B
```

Listing 2: Heaptrack

Conclusion

In the mid-2000s, Gentoo Linux was at the height of its popularity. It had an active community and extremely friendly documentation. The bugfixes, performance improvements, and security enhancements it brought to the Linux community are a shining beacon of what new technologies can do for the overall community. In like manner, Rust and other memory-safe languages are ushering in a new age security-conscious development. Proposals for memory-safe C++, safety-critical Rust technologies like Ferrocene, and even experimental borrow checkers in other languages have exploded onto the scene. The benefits of Rust cannot be overstated, and the second-order effects of the Rust community on other programming languages will have good and meaningful cybersecurity enhancements for developers who may never have a desire to learn Rust. The programming landscape is changing. Just as Chrome OS wouldn't exist without Gentoo Linux, the same can be said about many of the upcoming C++26 proposals inspired by Rust programming paradigms.

I cannot say enough good things about Rust, Ada, and other memory-safe programming languages; however, that praise should be tempered with the realization that these languages are not the silver bullet of memory safety to mandate at an organizational level. Doing so will cause leadership to develop a false sense of security that will be dangerous in the long run.

References

[1] Jon Hood. “Ada Unchecked Conversions”. In: SwATips.com (2023). URL: https://www.swatips.com/articles/20230410.html.

[2] Jon Hood. “Back to the Building Blocks: Codifying Complacency”. In: SwATips.com (2024). URL: https://www.swatips.com/articles/20240902.html.

[3] Jon Hood, ed. SwATips. https://www.SwATips.com/.