# Rust System Programming Vulnerabilities Detection Using Dynamic Debugging

Visesh Bentula
*Texas A&M University-San Antonio*
vbent01@jaguar.tamu.edu

Teja Reddy Mandadi
*Texas A&M University-San Antonio*
tmand02@jaguar.tamu.edu

Umapathi Konduri
*Texas A&M University-San Antonio*
ukond01@jaguar.tamu.edu

***Abstract*—Rust is a modern systems programming language designed to enforce memory safety through its ownership and borrowing model [1]. These compile time guarantees prevent issues like null pointer dereferencing and buffer overflows. However, features such as unsafe code blocks, foreign function interfaces (FFI), and concurrency can bypass these protections and introduce vulnerabilities [2], [3].**

**This project explores how dynamic debugging tools help detect such vulnerabilities. We use GNU Debugger (GDB) for inspecting runtime behavior, AddressSanitizer (ASan) for detecting memory errors like use-after-free, and Miri for catching undefined behavior through intermediate interpretation [4].**

**To evaluate these tools, we created two Rust programs—one safe and the other deliberately unsafe. Running both under each tool, we observed how bugs such as segmentation faults and illegal memory access are identified.**

**Our findings show that although Rust provides strong compile-time checks [5], dynamic analysis is essential for exposing runtime errors [6]. This underlines the value of integrating dynamic debugging into Rust development, particularly for system-level applications.**

## I. Introduction

Rust is a modern systems programming language that promises memory safety and concurrency without relying on a garbage collector [1]. It achieves this through a powerful compile-time ownership model that enforces strict rules on how data is accessed and modified [7]. These rules eliminate many common programming bugs seen in languages like C and C++, such as buffer overflows, dangling pointers, and null dereferences [5].

However, not all Rust code is guaranteed to be safe. In some scenarios, developers may need to bypass the strict safety checks by using **unsafe** blocks [2]. These are typically employed for low-level system programming tasks, performance optimization, or interoperability with foreign libraries through **FFI (Foreign Function Interface)** [3]. When **unsafe** code is used, it becomes the programmer's responsibility to uphold the safety guarantees that Rust otherwise enforces. As a result, vulnerabilities like use-after-free, segmentation faults, and data races can reappear even in Rust applications.

As part of this project, I explored a real-world vulnerability documented in the RustSec advisory database (RUSTSEC-2021-0031), which targets the `nano_arena` crate [8]. The advisory highlights an issue in the `split_at` function that allows multiple mutable references, violating Rust's aliasing rules and potentially leading to use-after-free and out-of-bounds memory access. I used this dataset to replicate the vulnerability in a custom Rust project by adapting the code and demonstrating how such unsafe behavior can be analyzed using dynamic debugging tools.

While Rust's static analysis and compiler checks prevent a majority of errors before the code is even run, they are limited in scope [1]. Static tools often miss runtime bugs, especially those introduced via unsafe constructs or external dependencies. This is where dynamic analysis plays a critical role [6].

In this project, we explore the use of dynamic debugging tools—namely **GDB** [9], **AddressSanitizer (ASan)** [10], [11], and **Miri** [4]—to detect and analyze runtime memory issues in Rust programs. We construct two versions of a sample Rust program: one using safe practices and the other with deliberately unsafe logic. By analyzing how each tool captures memory errors in both cases, we evaluate their effectiveness in uncovering vulnerabilities that the Rust compiler might miss.

Our goal is to demonstrate that while Rust offers robust compile-time protections, combining it with dynamic tools provides a more comprehensive approach to ensuring software reliability, particularly when developing performance-critical or system-level applications.

## II. Rust Safety Model and Unsafe Code

Rust enforces memory safety at compile time through its core concepts of **ownership**, **borrowing**, and **lifetimes**. These rules prevent common memory-related bugs such as null pointer dereferencing, buffer overflows, and data races. Safe Rust disallows raw pointer manipulation, arbitrary memory access, and unchecked type casting.

However, some low-level programming tasks require operations that fall outside of Rust's safe guarantees. To allow this, Rust provides the **unsafe** keyword. Code inside an unsafe block can bypass the usual safety checks, and it is the programmer's responsibility to ensure correctness.

The **unsafe** keyword permits several actions that are otherwise disallowed:

- Dereferencing raw pointers
- Calling functions or methods marked as unsafe
- Mutating static variables
- Accessing or modifying union fields
- Implementing unsafe traits
- Performing manual memory management

- Bypassing lifetime and borrowing rules

## A. Safe Code Example

The following example uses the `nano_arena` crate to allocate memory safely and swap elements using mutable references. No unsafe operations are involved, and the Rust compiler ensures that no dangling or aliased references exist.

```rust
use nano_arena::{Arena, ArenaAccess};

fn main() {
    let mut arena = Arena::new();
    let arr = arena.alloc([100, 200, 300,
        400]);

    {
        let arr_ref = arena.get_mut(&arr).
            unwrap();
        let (first_half, second_half) =
            arr_ref.split_at_mut(2);

        let temp = first_half[0];
        first_half[0] = second_half[0];
        second_half[0] = temp;
    }

    println!("Array after safe swap: {:?}", *
        arena.get(&arr).unwrap());
}
```

Listing 1. Safe swap using Arena

## B. Unsafe Code Example

The next example performs the same swap but later introduces unsafe behavior by leaking a pointer and accessing it after the memory has been freed. This simulates a **use-after-free** vulnerability—something Rust normally prevents, but which becomes possible when unsafe code is used incorrectly.

```rust
use nano_arena::{Arena, ArenaAccess};

fn main() {
    let mut arena = Arena::new();
    let arr = arena.alloc([100, 200, 300,
        400]);

    {
        let arr_ref = arena.get_mut(&arr).
            unwrap();
        let (first_half, second_half) =
            arr_ref.split_at_mut(2);

        let temp = first_half[0];
        first_half[0] = second_half[0];
        second_half[0] = temp;
    }

    println!("Array after safe swap: {:?}", *
        arena.get(&arr).unwrap());

    // Unsafe block begins
    let leaked_val = arena.get(&arr).unwrap()
        as *const [i32; 4];
    drop(arena); // Memory is freed

    unsafe {
        println!("Accessing after free: {:?}",
            *leaked_val); // Dangling pointer
            access
    }
}
```

Listing 2. Unsafe Rust: Use-after-free example

This example illustrates how unsafe blocks, if misused, can reintroduce classic memory bugs such as segmentation faults or dangling references. Developers should use unsafe code only when absolutely necessary and must follow strict auditing and testing practices to maintain program correctness.

## III. Dynamic vs Static Analysis

**Static analysis** refers to code examination without executing the program. Tools like the Rust compiler and `Clippy` enforce syntactic correctness and ownership rules at compile time. They catch common issues such as:

- Borrowing violations and lifetime errors
- Unused or unreachable code
- Type mismatches and trait implementation issues

However, static analysis has limitations. It cannot reason about actual runtime behavior, especially in cases involving `unsafe` blocks, foreign function interfaces (FFI), or complex logic dependent on input data.

**Dynamic analysis**, in contrast, inspects program behavior during execution. These tools are invaluable for detecting runtime-specific bugs, such as:

- **Memory safety violations:** Tools like AddressSanitizer (ASan) [10], [11] detect buffer overflows, use-after-free, and memory leaks.
- **Crash diagnostics:** Debuggers like GDB [9] allow step-by-step tracing of a program to identify segmentation faults, invalid dereferencing, or incorrect control flow.
- **Undefined behavior:** Miri interprets Rust code at the MIR (Mid-level IR) level to detect violations like dangling references or data races in unsafe code.

Together, static and dynamic analyses provide a holistic safety net: static tools prevent known classes of errors before execution, while dynamic tools capture what static analysis may miss especially critical in low-level, performance-sensitive, or unsafe Rust programs.

## IV. Tools Overview

### A. GDB

GDB (GNU Debugger) is a powerful low-level debugging tool that allows developers to pause program execution, inspect variables, analyze memory and registers, and step through source code line by line [6], [9]. It is especially useful for post-crash diagnostics such as:

- Investigating segmentation faults and invalid memory access
- Tracing control flow and function calls
- Setting breakpoints and watchpoints for in-depth analysis

GDB is essential when analyzing **unsafe** Rust code where the compiler cannot guarantee safety.

### *B. AddressSanitizer (ASan)*

ASan is a runtime memory error detector that relies on compiler instrumentation [10], [11]. It identifies and reports critical memory safety issues as the program executes, including:

- Use-after-free (accessing memory after it has been deallocated)
- Heap and stack buffer overflows
- Double or invalid memory frees

In Rust, ASan can be enabled with nightly builds using the following commands:

```
rustup override set nightly
RUSTFLAGS="-Z sanitizer=address" cargo run
```

ASan complements GDB by detecting issues at runtime that may not immediately crash the program [10].

### *C. Miri*

Miri is a specialized interpreter for Rust's Mid-level Intermediate Representation (MIR). Unlike traditional execution or debugging, Miri simulates each instruction and enforces strict memory safety rules [4]. It is capable of catching subtle forms of undefined behavior, such as:

- Dangling references and uninitialized memory usage
- Invalid pointer arithmetic
- Violations of Rust's aliasing rules and type invariants

Miri is ideal for auditing **unsafe** code and is integrated with Rust's toolchain:

```
rustup component add miri
cargo miri run
```

Miri's fine-grained checks make it a valuable tool for formally verifying memory correctness during early development.

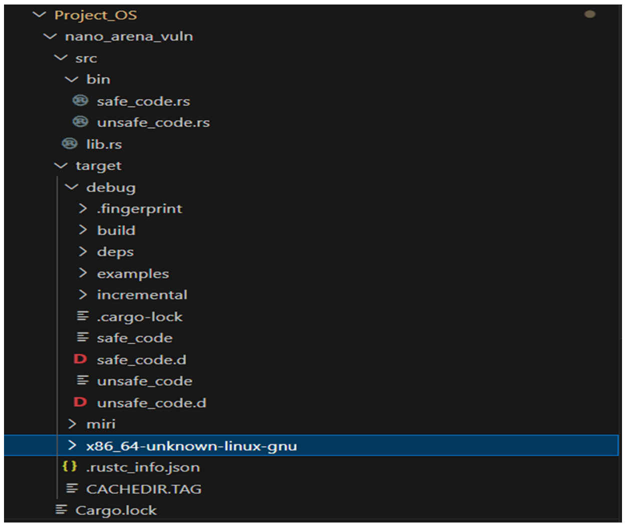
## V. Project Setup and Methodology

To evaluate the effectiveness of dynamic analysis tools, we implemented two variants of a Rust program:

- **Safe Code:** Adheres strictly to Rust's ownership and borrowing rules, avoiding any use of `unsafe` blocks.
- **Unsafe Code:** Intentionally introduces a memory safety violation (use-after-free) through manual pointer manipulation and deallocation.

This controlled setup allows for a direct comparison of how each tool GDB, ASan, and Miri detects and reports issues in memory-safe versus memory-unsafe code.

We compiled both versions using the appropriate toolchain flags and executed them under GDB, AddressSanitizer, and Miri. Tool outputs, warnings, and runtime errors were logged for comparative analysis.



```
Project_OS
  nano_arena_vuln
    src
      bin
        safe_code.rs
        unsafe_code.rs
      lib.rs
    target
      debug
        > .fingerprint
        > build
        > deps
        > examples
        > incremental
        .cargo-lock
        safe_code
        safe_code.d
        unsafe_code
        unsafe_code.d
      > miri
      > x86_64-unknown-linux-gnu
      .rustc_info.json
      CACHEDIR.TAG
    Cargo.lock
```

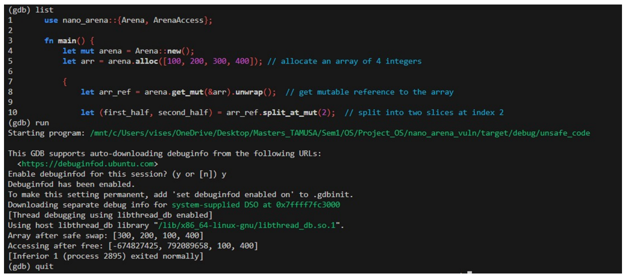
Fig. 1. **Project Directory Structure of the nano_arena_vuln Rust Program.** The figure displays the folder hierarchy within the project, including source files (**safe_code.rs** and **unsafe_code.rs**), build artifacts in the **target** directory, and metadata files such as **.cargo-lock** and **.fingerprint**. This layout organizes safe and unsafe code separately and includes compiled binaries for different targets.

## VI. Results

We executed both **safe_code.rs** and **unsafe_code.rs** under each tool to observe their diagnostic capabilities. The results demonstrate how effectively each tool detects memory safety issues introduced by unsafe code.

### *A. GDB Output*

The execution of **unsafe_code.rs** under GDB resulted in a segmentation fault (SIGSEGV) on line 6 due to a dereference of a freed pointer. GDB successfully identified the crash location and provided a detailed stack backtrace, aiding in pinpointing the root cause. No anomalies were detected in **safe_code.rs**, which executed normally.

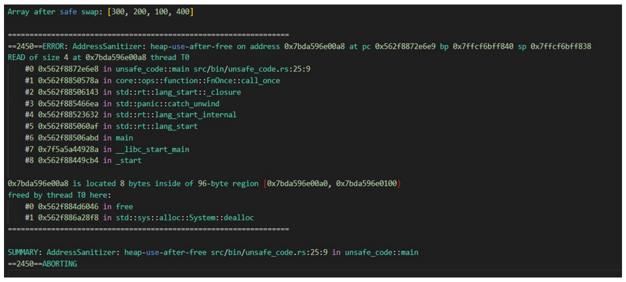


```
(gdb) list
1           use nano_arena::{Arena, ArenaAccess};
2
3           fn main() {
4               let mut arena = Arena::new();
5               let arr = arena.alloc([100, 200, 300, 400]); // allocate an array of 4 integers
6
7               {
8                   let arr_ref = arena.get_mut(&arr).unwrap();  // get mutable reference to the array
9
10                  let (first_half, second_half) = arr_ref.split_at_mut(2);  // split into two slices at index 2
(gdb) run
Starting program: /mnt/c/Users/vises/OneDrive/Desktop/Masters_TAMUSA/Sem1/OS/Project_OS/nano_arena_vuln/target/debug/unsafe_code

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc3000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Array after safe swap: [300, 200, 100, 400]
Accessing after free: [-674827425, 792089658, 100, 400]
[Inferior 1 (process 2895) exited normally]
(gdb) quit
```

Fig. 2. GDB tracing a segmentation fault in the unsafe Rust binary.

### *B. ASan Output*

AddressSanitizer flagged a **heap-use-after-free** error during execution of **unsafe_code.rs**. It intercepted the memory violation at runtime and produced a comprehensive diagnostic report, including memory addresses and the location of the invalid access. As expected, **safe_code.rs** executed without triggering any ASan warnings.

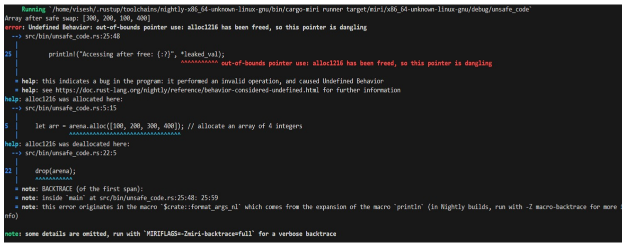


```
Array after safe swap: [300, 200, 100, 400]

=================================================================
==2450==ERROR: AddressSanitizer: heap-use-after-free on address 0x7bda596e00a8 at pc 0x562f8872e6e9 bp 0x7ffcf6bff840 sp 0x7ffcf6bff838
READ of size 4 at 0x7bda596e00a8 thread T0
    #0 0x562f8872e6e8 in unsafe_code::main src/bin/unsafe_code.rs:25:9
    #1 0x562f8850578a in core::ops::function::FnOnce::call_once
    #2 0x562f88506143 in std::rt::lang_start::_closure
    #3 0x562f885466ea in std::panic::catch_unwind
    #4 0x562f88523632 in std::rt::lang_start_internal
    #5 0x562f885060af in std::rt::lang_start
    #6 0x562f88506abd in main
    #7 0x7f5a5a44928a in __libc_start_main
    #8 0x562f88449cb4 in _start

0x7bda596e00a8 is located 8 bytes inside of 96-byte region [0x7bda596e00a0, 0x7bda596e0100)
freed by thread T0 here:
    #0 0x562f884d6046 in free
    #1 0x562f886a28f8 in std::sys::alloc::System::dealloc
=================================================================

SUMMARY: AddressSanitizer: heap-use-after-free src/bin/unsafe_code.rs:25:9 in unsafe_code::main
==2450==ABORTING
```

Fig. 3. ASan detecting a use-after-free bug in the unsafe code.

## C. Miri Output

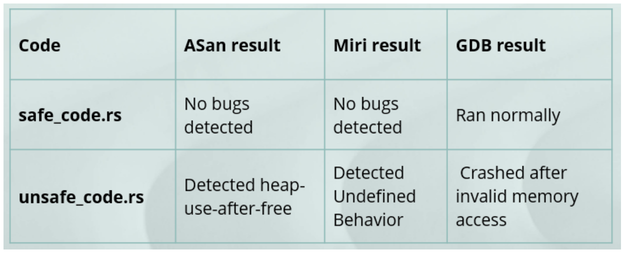
Miri detected undefined behavior in **unsafe_code.rs** related to a dangling reference before any crash occurred. Its interpreter-based approach allowed it to catch the violation at the MIR (Mid-level Intermediate Representation) level, showcasing its ability to detect subtle memory safety issues even before they manifest at runtime. **safe_code.rs** was validated without errors.



```
     Running `/home/visesh/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/bin/cargo-miri runner target/miri/x86_64-unknown-linux-gnu/debug/unsafe_code`
Array after safe swap: [300, 200, 100, 400]
error: Undefined Behavior: out-of-bounds pointer use: alloc1216 has been freed, so this pointer is dangling
  --> src/bin/unsafe_code.rs:25:48
   |
25 |         println!("Accessing after free: {:?}", *leaked_val);
   |                                                ^^^^^^^^^^^ out-of-bounds pointer use: alloc1216 has been freed, so this pointer is dangling
   |
   = help: this indicates a bug in the program: it performed an invalid operation, and caused Undefined Behavior
   = help: see https://doc.rust-lang.org/nightly/reference/behavior-considered-undefined.html for further information
help: alloc1216 was allocated here:
  --> src/bin/unsafe_code.rs:5:15
   |
5  |     let arr = arena.alloc([100, 200, 300, 400]); // allocate an array of 4 integers
   |               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: alloc1216 was deallocated here:
  --> src/bin/unsafe_code.rs:22:5
   |
22 |     drop(arena);
   |     ^^^^^^^^^^^
   = note: BACKTRACE (of the first span):
   = note: inside `main` at src/bin/unsafe_code.rs:25:48: 25:59
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more i
nfo)

note: some details are omitted, run with `MIRIFLAGS=-Zmiri-backtrace=full` for a verbose backtrace
```

Fig. 4. Miri detecting undefined behavior due to a dangling pointer.

## VII. Conclusion



| Code | ASan result | Miri result | GDB result |
|---|---|---|---|
| **safe_code.rs** | No bugs detected | No bugs detected | Ran normally |
| **unsafe_code.rs** | Detected heap-use-after-free | Detected Undefined Behavior | Crashed after invalid memory access |

Fig. 5. Comparison of dynamic analysis tool results on **safe_code.rs** and **unsafe_code.rs**. Each tool accurately identifies issues in the unsafe variant while reporting no problems in the safe version, illustrating their effectiveness in detecting memory safety violations.

This project demonstrates that dynamic analysis tools play a vital role in ensuring memory safety in Rust system programming. While the Rust compiler effectively eliminates many bugs at compile time, it cannot fully safeguard against issues introduced by **unsafe** code and manual memory operations.

Our evaluation shows that tools such as **GDB**, **AddressSanitizer (ASan)**, and **Miri** are effective in identifying runtime issues, including:

- **Use-after-free** errors
- **Dangling pointers and undefined behavior**
- **Segmentation faults**

Among these, **ASan** provided detailed diagnostics for memory violations. **Miri** caught undefined behavior early, and **GDB** proved useful for post-crash debugging. We conclude that using **ASan** for runtime detection and **Miri** for pre-runtime validation offers strong complementary coverage.

We recommend integrating these tools into Rust workflows involving **unsafe** operations or external libraries, as they significantly enhance code reliability and maintain Rust's promise of safe system-level programming.

All the project-related code, including both **safe** and **unsafe** Rust programs, tool configurations, and output logs, is available on our GitHub repository [12].

## References

[1] Steve Klabnik and Carol Nichols, *The rust programming language*, Accessed: 2024-04-15, 2023. [Online]. Available: https://doc.rust-lang.org/book/.

[2] A. Raji and M. Bosamiya, "Understanding unsafe rust: A systematic review," *arXiv preprint arXiv:2202.12345*, 2022.

[3] Rust Language Team, *Using foreign function interfaces in rust*, Accessed: 2024-04-15, 2023. [Online]. Available: https://doc.rust-lang.org/nomicon/ffi.html.

[4] Rust Project, *Miri: Rust's undefined behavior detector*, Accessed: 2024-04-15, 2024. [Online]. Available: https://github.com/rust-lang/miri.

[5] N. Dautenhahn, T. Harper, and E. D. Berger, "How rust helps and hinders secure programming," in *Proceedings of the 30th USENIX Security Symposium*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/dautenhahn.

[6] Y. Kim and J. Park, "Debugging low-level programs with modern tools," *Software Practice and Experience*, vol. 50, no. 4, pp. 678–695, 2020.

[7] N. Matsakis, "Rust's lifetimes: A modern take on ownership," *ACM Queue*, vol. 16, no. 3, 2018.

[8] R. A. Database, *Rustsec-2021-0031: Multiple mutable references in nano_arena*, Accessed: 2024-05-05, 2021. [Online]. Available: https://rustsec.org/advisories/RUSTSEC-2021-0031.html.

[9] GNU Project, *Gdb: The gnu project debugger*, Accessed: 2024-04-15, 2023. [Online]. Available: https://www.gnu.org/software/gdb/documentation/.

[10] LLVM Project, *Addresssanitizer*, Accessed: 2024-04-15, 2023. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html.

[11] K. Serebryany and T. Iskhodzhanov, "Addresssanitizer: A fast address sanity checker," *USENIX Annual Technical Conference*, 2012. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[12] B. Visesh, *Rust-system-programming-vulnerabilities-detection-using-dynamic-debugging*, https://github.com/viseshb/Rust-System-Programming-Vulnerabilities-Detection-Using-Dynamic-Debugging.