다중 언어 어플리케이션에서의 러스트 언어 보호에 대한 연구

유준승[1], 카욘도마틴[1], 백윤흥[2]
[1]서울대학교 전기정보공학부 석박통합과정, 반도체공동연구소
[2]서울대학교 전기정보공학부 교수, 반도체공동연구소
jsyou@sor.snu.ac.kr, kayondo@snu.ac.kr, ypaek@snu.ac.kr

A Study on Securing Rust in Mixed-Language Applications

Junseung You[1], Martin Kayondo[1], Yunheung Paek[1]
[1]Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC), Seoul National University

Abstract

For many decades, memory corruption attacks have posed a significant threat to computer systems, particularly those written in unsafe programming languages such as C/C++. In response, a 'safe' programming language, Rust, was recently developed to prevent memory bugs by using compile-time and runtime checks. Rust's security and efficiency has lead its adoption from multiple popular applications such as Firefox and Tor. Due to the large code base and complexity of legacy software, the adoption generally takes a form of a gradual deployment, where security-critical portion of the program is replaced with Rust, resulting in a mixed-language application. Unfortunately, such adoption strategy introduced a new attack vector that propagates the vulnerabilities residing in the unsafe languages to Rust, undermining the security guarantees provided by Rust. In this paper, we shed light on strategies designed to defend against attacks that target multi-lingual applications to compromise the security of Rust. We study underlying rationale of various defense mechanisms and design decisions taken to improve their performance and effectiveness. Furthermore, we explore the limitations of existing defenses and argue that additional methods are necessary for Rust to fully benefit from its security promises in multi-language environments.

1. Introduction

Rust has been recently developed as a modern, safe programming language to defend against memory vulnerabilities such as buffer overflows and use-after-frees (UAFs) that for decades have plagued the legacy software written in C/C++. Rust achieves memory safety by performing compile-time and runtime checks along with having a strong type system that prevents arbitrary casting. For example, Rust performs compile-time ownership checks to prevent temporal memory safety bugs, and enforces runtime bounds check on dynamic data on top of compile-time bounds checks on static data to prevent spatial memory corruption bugs.

Rust was considered as the best way to develop safe system, resultiing in a wide adoption from multiple popular applications and systems such as Firefox, Tor, and Microsoft Windows operating system. However, due to large size and complexity of legacy software, the developers opted in for an adoption strategy that gradually deploys Rust within the existing code base by replacing partial components of the application with Rust, rather than rewriting the whole software with Rust in a bottom-up fashion. Naturally, strategy lead to an emergence of mixed-language applications, where two (or more) languages are used in development.

Sadly, while such incremental development of parts of the software with Rust is performed in order to *enhance* its security, multiple researches have pointed out that such strategy instead

weakeens the security by exposing a new window for exploitation. Being part of the same program naturally places the components written in different languages into a same memory address space, thereby allowing the vulnerabilities residing in unsafe components to affect the entire program, even the parts written in Rust, thereby undermining the security guarantees that are expected from its adoption.

In this work, we shed light on strategies that aim to protect Rust and maintain its security benefits in presence of components developed in unsafe programming languages within the single application. We conduct an in-depth analysis of existing defense mechanisms and identify that common, underlying rationale that they advocate for is *isolation* of Rust from unsafe components which quarantines the memory accesses from the code written in unsafe languages, thus preventing any vulnerabilities from being propagated to Rust. More importantly, our analysis reveals that existing defenses are not bullet-proof, still leaving out a loophole for exploitation caused by their design decision to realize isolation at a page level. We argue that this loophole, while it may seem small, is critical in safekeeping Rust, and inevitably necessitates a new mechanism to fully tighten the security of Rust in mixed-language environments.

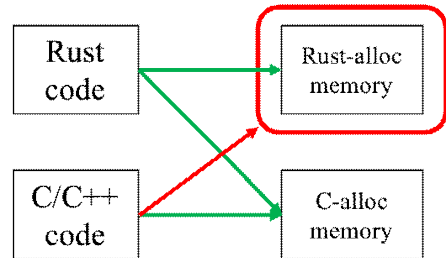
2. Vulnerabilities in Mixed-Language Applications

The root cause of memory vulnerabilities in mixed-language applications is single memory address space, shared by both parts written in different programming languages. While parts written in Rust is safe from memory bugs, the counterparts written in C/C++ are still plagued by them. While the security guarantees of Rust can be preserved if the bugs in unsafe languages are contained within the components written in them, being part of the same application inherently allows both Rust and C/C++ code to access the same memory address space, thereby allowing the bugs in unsafe code to propagate to Rust and affect its secuirty guarantees. Figure 1 shows an example of such vulnerability.

```
pub fn printw(s: &str) -> i32 {
  unsafe {ll:printw(s.to_c_str().as_ptr()) }
}
```

(Figure 1) Format String Vulnerability.

The example shows an call to the external library writen in C/C++ (i.e., *ll:printw*) where the memory safety of stack objects used by Rust can be undermined. Specifically, similar to the format string bug, undefined string conversion parameters (e.g., printw("%s%s%s")) in unsafe language allows an attacker to read as much information as they want from the stack.



(Figure 2) Memory Access Rules with Isolation.

3. Defense Mechanism to Protect Rust in Mixed - Language Applications

Various studies [1, 2, 3, 4] sought to protect Rust for unsafe languages in multi-lingual application. While they differ in implementation details, they all opt for the same design rationale of isolating Rust from unsafe language by quarantining (i.e., restricting) unsafe language from having access to memory used by Rust. Through isolation, memory bugs in C/C++ are contained within the isolation boundary, naturally prohibiting them from being propagated to the Rust side. Figure 2 depicts the common memory access behavior from different languages after enforcment of isolation. Rust code, free of bugs, is allowed to access any application memory that consists of both the memory allocated from the Rust code and the memory allocated from the unsafe language. On the contrary, unsafe code (e.g., C/C++ code) is quarantined such that it is only allowed to access the memory allocated from

themselves.

Existing works on isolating Rust differ in their implementations regarding the analysis of the isolation boundary and isolation enforcement. Isolation boundary analysis consists of identifying and classifying the memory objects to either Rust-used or C/C++-used. The analysis strategy can be roughly divided into two directions. The first direction utilizes static analysis from the compiler to identify the memory objects. Specifically, this approach conducts a points-to analysis to pinpoint the allocation sites of the objects by traversing the points-to relations starting from the object access sites. If the object is accessed from the unsafe language, it is classified as unsafe, and vice versa for the ones accessed solely from Rust. The second approach opts to leverage dynamic profiling for analysis. Rather than relying on static analysis that may result false positives or negatives or overappr - oximation of unsafe objects, they conduct a preliminary exeuction of the application and mark the objects that flow into unsafe languages and classify them as unsafe objects. Isolation strategy adopted by existing mechanisms are generally in line with those designed for intra-process isolation. While a long line of research in that field offers multiple options, such as software fault isolation (SFI) and virtualization, that each have their unique strengths and weaknesses, recent works on Rust isolation sought assistance from hardware, namely Memory Protection Keys (MPK), provided on Intel CPUs. The rationale behind their choice of MPK for isolation is straightforward, as MPK demonstrates dominant performance as memory access restriction is enforced through hardware and offers lightweight access permission update through modifying register accessible in userspace, thus avoiding costly context switches into the kernel.

4. Security Loophole in Existing Defenses

While isolating Rust from unsafe language narrows down the attack surface of exploitation, the existing solutions are not bulletproof. The first loophole stems from their common design decision to classify the unsafe objects into a large, single domain. That is, all objects that are classified as unsafe are placed into a single isolated region, where memory bugs such as overflows within the region (i.e., intra-region overflow) are considered out-of-scope. Such single dimensional classification results in categorizing objects that are accessed both the Rust and unsafe languages as unsafe objects. As a result, overflow inside the unsafe region that modifies the objects that are shared between both safe and unsafe languages can affect Rust's behavior and undermine its security. Figure 3 shows a simplified example of such an attack. In the example, both x1 and x2 is placed in the unsafe region, allowing any one of them to overflow onto the other. As a result, this allows a vulnerable function that takes pointer to x2 as an argument to overflow to x1, which can change the control flow of the Rust code.

```
fn rust_fn(cb_fptr: fn(&mut i64)) {
  let mut x1 = Box::new();
  let mut x2 = Box::new();
  unsafe { fn( /* pointer to x1 */ ); }
  unsafe { vuln_fn( /* pointer to x2 */ ); }
  if (x1 == ?) {
    …
  }
}
```

(Figure 3) Example of Intra-Unsafe Region Overflow.

The second loophole comes from existing defenses' common decision to utilize MPK for isolation. As MPK offeres memory access control at a page (4KB) granularity, whole object must be placed in a unsafe region even if only the subset (or subfield) of the object is actually accessed by the unsafe language. Consequently, this allows intra-object overflow to change Rust code's behavior and undermine its security. Figure 4 shows an example that can lead to such attack.

In this example, the object "Data" is classified as unsafe and is placed in the unsafe region. Data has two subfields where the first field holds an array of values and the second field holds a function pointer that is later used in Rust code. Although the first subfield (i.e., array) is needed by unsafe code, memory bug can overflow the second subfield and modify the function pointer to divert the Rust execution flow to non-intended destination.

```
fn rust_fn(cb_fptr: fn(&mut i64)) {
  let mut x = Data {
    vals: [1,2,3],
    cb: cb_fptr,
  };
  unsafe{ vuln_fn( /* ptr to x.vals */ ) }
  // cb_fptr is used later on
  (x.cb)(&mut x.vals[0]);
```

(Figure 4) Example of Intra-Object Overflow.

5. Conclusion

In light of multi-language application that is developed in both Rust and unsafe languages such as C/C++, isolation is adopted as a common rationale to safekeep Rust by preventing memory bugs in unsafe languages from propagating into Rust code and undermine its security guarantees. Existing works narrows down the attack vector by classifying memory objects to identify the isolation boundary and enforcing the isolation at a page granularity efficiently with hardware support from Intel MPK. Unfortunately, the solutions are not a panacea as their design decisions inherently allows intra-region overflows as well as intra object overflows. This work sheds light on such loopholes by analyzing the existing mechanisms and presenting the possible attack scenarios with concrete examples. We argue that such loopholes are not to be simply considered as out-of-scope and urge attention to devise new means to fully tighten the security to safekeep Rust in multi-lan guage environments.

References

[1] Bang et. al., TRUST: A Compilation Framework for In-process Isolation to Protect Safe Rust aginst Untrusted Code, USENIX Security Symposium, Anaheim, CA, USA, 2023, pg.6947-6964

[2] Mergendahl et. al., Cross Language Attacks, Network and Distributed Systems Security Symposium, San Diego, CA, USA, 2022

[3] Rivera et. al., Keeping Safe Rust Safe with Galeed, Annual Computer Security Applications Conference, Virtual Event, USA, 2021

[4] Kirth et. al., PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Langauges, European Conference on Computer Systems, Rennes, France, 2022, pg.132 - 148

Acknowledgements

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government, Minitry of Science and ICT (MSIT) (RS-2023-00277326), the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul national University in 2024, and Inter-University Semiconductor Research Center (ISRC); in part by the Institute of Information & Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00438729, Development of Full Lifecycle Privacy-Preserving Techniques using Anonymized Confidential Computing); and in part by Korea Planning & Evaluation Institute of Industrial Technology (KEIT) grant funded by the Korea governement (MOTIE) (No. RS-2024-00406121, Development of an Automotive Security Vulnerability-based Thread Analysis System (R&D)); and in part by IITP under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081) grant funded by the Korea government (MSIT).