CROSS LANGUAGE CALL SANITIZATION

1st Alexandre Grimaldi
Computer Science and Engineering dep.
Sungkyunkwan University
Seoul, Republic of Korea
alexandre.grimaldi@telecom-sudparis.eu

2nd João Vasco Reis
Artificial Intelligence
Sungkyunkwan University
Seoul, Republic of Korea
joao.vasco.sobral@tecnico.ulisboa.pt

3rd Luca B. Knaack
Computer Science and Engineering dep.
Sungkyunkwan University
Seoul, Republic of Korea
l.knaack93@gmail.com

***Abstract*—A lot of well-known software use multiple languages in their conception or during a language transition, increasing their complexity and the number of potential vulnerabilities exponentially (e.g Firefox, Tor, Microsoft Windows operating system, Google Fuchsia OS, docker, Kubernetes etc). We looked into how those vulnerabilities could be secured by known sanitizers and what means would have to be added to those known sanitizers to make them viable in detecting cross language call vulnerabilities.**

I. Introduction

The C programming language has been the language of choice for many programmers for half a century now when it comes to developing embedded systems or low level programs. This is noticeable in that a lot of today's software contains C code or C++ code. However, C is not perfect. The freedom it gives limits the security of the language in this respect. In particular, the memory safety. Modern languages like Rust or Go are therefor increasingly used to replace legacy C-code and make it more secure, with their inherent security features. However, large code bases such as those of programs like Firefox or the Tor browser cannot be written completely in another language from one moment to the next. The result is programs written in two languages, although other reasons for software written in multiple languages exist. Recently published studies (Mergendahl et al., 2022) have shown that the use of multiple languages poses problems. Different underlying security assumptions lead to incompatible risk models and a program that uses code from two languages becomes vulnerable to so-called Cross Language Attacks (CLA). Hereby the attacker is able to perform an attack not possible in only one of the languages by skillfully navigating between them and exploiting their security assumptions.

II. CLA

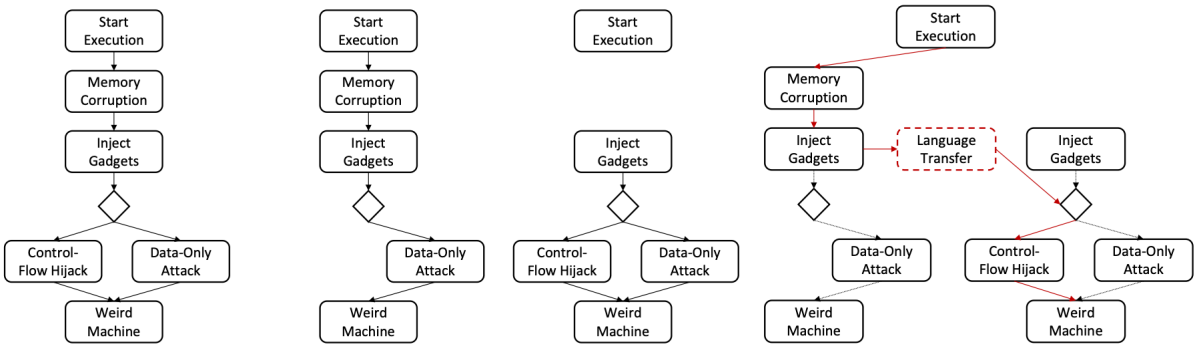
The basic sequence of a cross language attack is easy to understand. As an example, we will use software which switches between Rust and C code. If a function in C is called from a Rust program using the Foreign Functions Interface (FFI), the memory safety, which is anticipated by Rust, can no longer be guaranteed. If the metadata of a Rust vector (for example the size) is changed in C code, which is easily possible, the Rust code then has the possibility to make an out of bounds access, although this is not possible using only Rust code. (Mergendahl et al., 2022) demonstrates many other examples, and shows, that those cross language attacks are generally possible.

Fig. 1 shows the general problem with misaligned security assumptions schematically. The goal of an attack here is to start a weird machine (Shapiro et al., 2013). As the picture shows the attacker has "a way" to follow: from the C threat model, we can understand the easiest way to achieve the weird machine node. Starting the execution, trying to corrupt memory, injecting gadgets, Hijack Control-Flow and getting the weird machine are the most simple steps an attacker can have in an ideal situation. Getting a weird machine, would already be possible in C alone, but as C is commonly reinforced by CFI techniques, the attack is not possible simply in C (Control-Flow Hijack node is removed from the path), and just data-only attacks are possible through the memory corruption. The risk model of Rust (and Go respectively) makes it impossible to corrupt the memory, but no countermeasures against control-flow attacks are imposed, as the risk model is assuming that point in an attack can not be reached. However, if the features of both languages are used, an attacker can corrupt the memory in C, switch to Rust code after the corruption and continue with a control-flow attack to run the weird machine, taking by this way advantage from the vulnerabilities of each language. Finessing their protections(memory protections, Control-flow Protections), is possible as long as the languages used by the attacker do not have the same protection on the same node, otherwise it would be impossible to dodge it.

III. Adress Sanitizer - ASAN

Address Sanitizer (ASAN) is a memory error detector that helps to identify memory-related bugs in programs. It is a dynamic analysis tool that uses a combination of compile-time instrumentation and runtime memory analysis to detect memory errors in a program. ASAN is typically used to find buffer overflows, out-of-bounds accesses, and use-after-free errors in C and C++ programs.

When a program is compiled with ASAN, the compiler inserts special instrumentation code into the program's binary. This code tracks the use of memory in the program and checks for any illegal accesses or errors. When the program is run, ASAN monitors the memory accesses and looks for any violations of the rules that govern how memory can



(a) C threat model (b) C + CFI threat model (c) Rust/Go threat model (d) CLA threat model

Fig. 1. Language Threat Models (Mergendahl et al.)

be accessed. If a violation is detected, ASAN logs an error message and stops the program.

ASAN is relatively easy to use. To enable ASAN, you simply need to compile your program with the appropriate compiler flags. For example, if you are using GCC, you can use the '-fsanitize=address' flag to enable ASAN. Once a program is compiled with ASAN enabled, it can be run as it normally would, and ASAN will automatically detect and report any memory errors that it finds.

ASAN implementations do not exist solely for C and C++, however, with implementations also being published for languages like Go and Rust.

ASAN stores the state of each addressable or addressed memory byte in a memory region called shadow memory, using a special encoding. This encoding maps any memory byte, or multiple aligned ones, specifically to one byte in shadow memory.

ASAN, is a tool that uses shadow memory to efficiently detect and diagnose memory errors in computer programs. To do this, it divides the virtual address space into 8 equal parts and assigns one of these parts to the shadow memory. This allows ASan to use a small amount of memory to store information about a much larger amount of memory, making it efficient at detecting and diagnosing errors.

The address of a shadow byte can be determined by using the formula (Addr >>3)+Offset, where Addr is the address of the byte in the main memory, and Offset is a constant value determined by the ASan implementation. This formula maps the address of a byte in the main memory to the address of its corresponding shadow byte in the shadow memory.

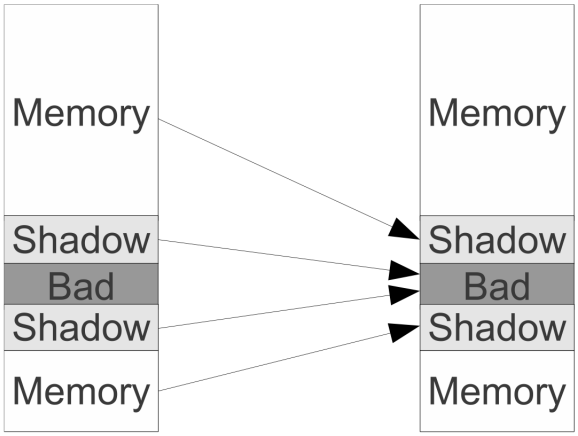


Fig. 2. Address Sanitizer memory mapping

IV. Related Work

Attacks can happen, once cross language calls are conducted. Cross Language Calls can allow simple attacks that would not be possible in one language on its own. (Mergendahl et al., 2022) provides a simple representation of how does one attack work, in many cases. Fig. 3 shows a simple representation of the clearest attack. From a sample code of Rust that calls a C/C++ function, we are able to analyze how can we create an out-of-bounds error, just by changing the length of the array. Some other examples leading to more errors are showed on the paper. Use-After-Free is also a common case, where the attacker is able to free the pointer passed as an argument, which will conduct to an incorrect use of the pointer afterwards. From that paper, it is clear that function calls must be done very carefully, and some precautions must be taken. Some methods can be used in order to prevent CLA attacks: granting memory isolation so that data is not referenced to unwanted destinations and Control Flow Guard [4] to prevent memory corruption vulnerabilities are some techniques applied to avert those attacks. An example would be Sandcrust [5], an easy-to-use sand-boxing solution for isolating code and data of a C library in a separate process. This isolation protects the Rust-based main program from any memory corruption caused by bugs in the unsafe library, which would otherwise invalidate the memory safety guarantees of Rust.

Rust usually considered a safe language, has been proven to have Unsafe Rust use. According to the paper (Peiming liu et al., 2020), even though 99 percent of the code is protected by Rust system code, 1 percent of code is enough to kill an entire program. This was the main reason that lead to the

```rust
fn rust_fn(cb_fptr: fn(&mut i64)) {
    // Initialize some data
    let mut x = Data {
        vals: [1,2,3],
        cb: cb_fptr,
    };

    unsafe{ vuln_fn(/*Ptr to x.vals*/) }

    // Uses corrupted function pointer
    (x.cb)(&mut x.vals[0]);
}
```

(a) Rust code that calls C/C++ to modify a Rust struct.

```c
// This function modifies a given array
// Can cause an OOB vulnerability
void vuln_fn(int64_t array_ptr_addr) {
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 3;
    int64_t array_value = get_attack();

    int64_t* a = (void *)array_ptr_addr;
    a[array_index] = array_value;
}
```

(b) C/C++ code that performs an Out-of-Bounds (OOB) error.

Fig. 3. Sample code to illustrate how CLA can circumvent the Rust type system to cause a OOB error. (Mergendahl et al.)

implementation of RustX, which intends to provide safety to "safe objects" once a memory corruption occurs. Fig. 4 enables us to see a brief resume of how the software is working. It starts by identifying the objects that are being processed by non safe Rust code, and it "modifies the code", to a more safe one. Full memory-safety is never guaranteed, because of the cost but also, because it is impossible to insure such thing. A lot of data are required to identify some vulnerabilities within unsafe code, and all the new examples found must be taken in consideration. The way XRust identifies the unsafe code, comes from various examples discovered previously.

V. Proposed Cross Language Vulnerability fixes

Solution ideas for avoiding cross language attacks are numerous and varied. Most ideas fall into one of the two following categories by either aiming at securing intended interactions or preventing unintended interactions. Ideas to prevent unintended interactions include moving the components of each language into a separate process and virtualizing them to maximize the separation. However, this is cumbersome and misses the point of using two languages. Also, the more languages involved, the more difficult this approach would be. Options fitting in neither category could be for example the control-flow-guard for Rust, which would enable the two languages C (with CFI) and Rust to share their security assumptions and therefor disable the possibility of special cross language attacks. However, those defense ideas are highly specializing on just some attack vectors and therefor only temporarily effective, as they only target the symptoms and not the cause of cross language attacks. Securing Intended interactions is mainly aimed at verifying the interactions that take place via an FFI. It is possible using formal methods, but highly impractical for larger software.

The solution we anticipate, on the other hand, would be to use sanitizers that would monitor inter-language calls, to detect potential threats and report it. Here, we explored different theoretical ideas to use existing sanitizers or create cross language sanitizers. While in the end sanitizers have to specifically aim at the foreign function call interface of one particular language in most cases, we were trying to find common patterns, principles and rules that would be able to secure cross language calls of multiple language pairs.

When we looked at the 5 proposed attacks performed by Mergendahl et al., 2022, we see that all of them could be prevented by preventing out of bounds or use after free accesses. The attack in Figure 3 for example could be prevented, if the C rust code knew, that an access at index 3 would be out of bounds. The second proposed attack by Mergendahl et al. coerces "Go into causing a UaF error" (Mergendahl et al., 2022, Figure 14). This attack could be prevented by synchronizing the frees for both languages. The third attack corrupts a "Go function pointer to execute a weird machine and circumvent CFI" (Mergendahl et al., 2022, Figure 15). Again, a bounds check could prevent the error. The fourth attack is an "Example of C/C++ using an arbitrary write to corrupt the size of a Go slice" (Mergendahl et al., 2022, Figure 16). A bounds check would have detected this vulnerability, too. The fifth and last attack "illustrate[s] how CLA can corrupt even data intended to cross the FFI boundary" (Mergendahl et al., 2022, Figure 17). Similar to the second attack, synchronizing the frees in both languages would prevent this vulnerability.

VI. Trying Existing Sanitizers

As we processed the several pieces of code (e.g Cross-Language vulnerabilities) we had, we were trying to see the impact they had, when executed with sanitization by state of the art sanitizers.

At first, we had troubles to run the target sanitizers that we wanted to try. We began by ASAN, one of the most well-known sanitizers. When only compiling the shared library that we use to run C code inside the GO code, and disassembling the library, we were able to find many ASAN flags, meaning the instrumentation actually worked, but the GO binary was crushing upon running. The binary was crashing because it seems it did not recognize the ASAN flags. After a while, we found out that we actually needed to change the LDFLAGS of the cgo command at the top of the GO code to correctly link the instrumented library with the GO code. Upon running, we discovered that only one of the vulnerabilities of our sample was detected by ASAN. Moreover, this vulnerability was the only one completely vulnerable with only using the C code

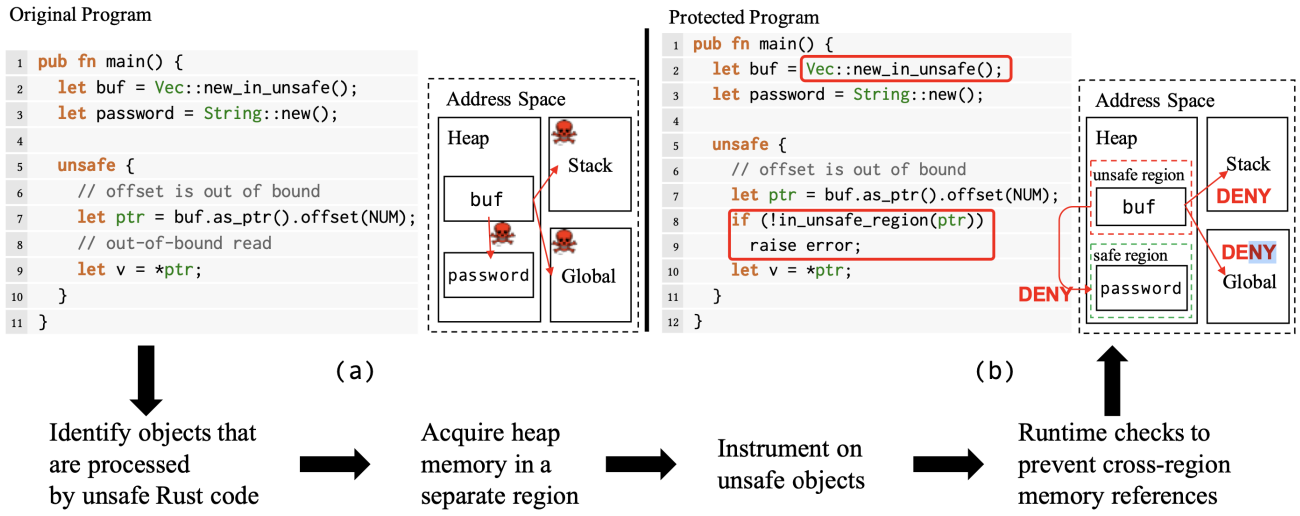


Fig. 4. A technical overview of XRust (using instrumentation-based memory isolation). (Peiming liu et al.

part, meaning ASAN does not actually detect any CLA from our experiments on our limited samples.

After this, we looked through the GO documentation and discovered that an ASAN flag already existed for the go build command. However, there were very few information about it and it created some bugs when we used it compared to the other way explained previously. Discarding the bugs created by the instrumentation, the results were the same, only the aforementioned vulnerability was detected.

An interesting thing to note is that the instrumentation was not done the same way depending on the chosen sanitization method (e.g. using a C compile flag or go build flag), comparing the disassembled binaries.

Then, we tried to implement LeakSanitizer, ThreadSanitizer, MemorySanitizer and UndefinedBehaviorSanitizer but none of them worked in this cross language context with C and GO either.

VII. Sanitizer Extension

Our results of testing existing sanitizers have shown, that those sanitizers could be used in a multi language setup. However they don't yet have the means to spot the cross language vulnerabilities unless they are written in just one languages, like the third vulnerability in Mergendahl et al., 2022.

When we analyzed five basic cross language call vulnerabilities, each one of them could be prevented with one of either two measures, as shown earlier:

1) Synchronizing knowledge about freed resources over language barriers
2) Synchronizing knowledge about the boundaries of resources over language barriers

For either of those knowledge synchronizations, shared meta data is important. When we look back at the implementation of common sanitizers like ASAN, we see, that they saved their meta data in so called shadow memory, and determine the location of the shadow memory using a mapping including an offset

We could use the same mapping method as ASAN, but at a different offset, to extend ASAN to save those aforementioned meta data we would need, into our own shadow memory.

Both of the measures can be saved in the same way by doing it as following: For each pointer, pointing at a memory location having an indexable data structure, we write the total boundary (e.g. the length of the array) into its associated shadow memory. Once the data structure is freed, we set the shadow memory to zero, indicating that it is freed or having a length of zero. The instrumentation that would check for valid indexes to access, would then compare the access index to the value stored in the shadow memory. If the access index is higher or equal as the shadow memory value, the access would not be possible. That would prevent out of bounds accesses as well as use after frees at the same time, thus, in theory, fixing all of our analyzed cross language vulnerabilities.

VIII. Conclusion

Evaluating the effectiveness of sanitizers has never been an easy task due to the difficulty of knowing how many errors were detected compared to the the total of existing bugs in a software, information that we do not possess. In our intended work, the task was even harder as we first aimed to find theoretical principles and rules before actually coming up with implementation ideas. Our first objective was to define how well state-of-the-art sanitizers can respond against cross language attacks, so we tried to make a classical comparison and evaluation of those different sanitizers on specific cases.

Cross language sanitization will be a crucial component of securing modern applications that use multiple programming languages. By properly sanitizing data and code from one language when it is processed by another language, we can help prevent attackers from exploiting differences between

languages to inject malicious code or data. Effective cross language sanitization can be achieved through a combination of different measures, some of which we showed in this work. Implementing these measures can help ensure that applications are secure and resistant to cross language attacks.

ASAN, or AddressSanitizer, is a memory error detection tool used in computer programming. It uses shadow memory, which is a copy of the contents of a computer's main memory (RAM), to detect and diagnose memory errors.

Since almost all the sanitizers, including ASAN, seem not to work with cross language attacks, we are able to conclude that the need for effective sanitizer implementation for cross language attacks is more urgent than ever, as many existing sanitizers are not capable of adequately protecting against these types of attacks. To protect against cross language vulnerabilities, it is essential to implement proper sanitization measures and regularly test the security of your application.

We therefor explained, how ASAN could be extended to save crucial meta data enabling instrumentation to prevent cross-language vulnerabilities.

IX. Future Work

Still, a lot of work can be done. As of now the shadow encoding of boundaries information, better said, the ASAN extension is not yet implemented. Seeing it work in production and testing it in real life scenarios against the proposed attacks will be the next, logical step in this work.

References

[1] Peiming Liu, Gang Zhao, and Jeff Wang, "Securing UnSafe Rust Programs with XRust," © 2020 Association for Computing Machinery, 2020.

[2] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," Proceedings 2022 Network and Distributed System Security Symposium, 2022.

[3] R. Shapiro, S. Bratus, S. W. Smith, "'Weird Machines"in ELF: A Spotlight on the Underappreciated Metadata', 7th USENIX Workshop on Offensive Technologies (WOOT 13), 2013.

[4] Andrew Paverd. (2020) Control flow guard for clang/llvm and rust.

[5] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, "Sandcrust: Automatic sandboxing of unsafe components in rust," in Proceedings of the 9th Workshop on Programming Languages and Operating Systems, 2017, pp. 51–57.

[6] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D. (2012). AddressSanitizer: a fast address sanity checker. USENIX Annual Technical Conference, 28