Tutorial: Sandboxing (unsafe) C code with RLBox

Shravan Narayan
UC San Diego
srn002@eng.ucsd.edu

Craig Disselkoen
UC San Diego
cdisselk@eng.ucsd.edu

Deian Stefan
UC San Diego
deian@cs.ucsd.edu

2021 IEEE Secure Development Conference (SecDev) | 978-1-6654-3170-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SecDev51306.2021.00017

***Abstract*—RLBox is a C++ framework for building secure systems from untrusted libraries. RLBox uses a static type system to (1) abstract isolation mechanisms like WebAssembly (2) make data and control flow across the application-library boundary explicit and safe, and (3) help developers retrofit their application with sandboxing. In this tutorial, we first give an overview of RLBox and demonstrate how the RLBox framework helps sandbox a (buggy) C library in a simple C++ application. Then, we walk through the process of using RLBox to sandbox libraries in larger C++ codebases like the Firefox Web browser.**

I. Introduction

Memory safety bugs are the single largest source of critical vulnerabilities in modern software. Recent studies by Microsoft [3] and the Chrome team [7] found that 70% of all critical vulnerabilities were caused by memory safety bugs.

One way to improve software security is to eliminate this whole class of bugs by construction. Safe languages (i.e., basically any language other than C or C++) and verification are a great way to do this. Another way, and the topic of this tutorial is to ensure that bugs do not impact security.

The most promising way to ensure that bugs in, say, a third-party library do not become security vulnerabilities is to run the library in a *software sandbox* [2]. Software sandboxing, or software-based fault isolation (SFI), uses runtime checks to ensure that the library is confined to its own region of memory — and cannot access the rest of the program memory.

Consider an image decoding library like libpng. With sandboxing, we can restrict this library so it *only* has access to the image it decodes and the bitmap it produces. The application benefits from the library's features, but doesn't inherit its security flaws. Bugs like the recent out-of-bounds read in libpng [4] would be *bugs*, not security vulnerabilities.

The idea that software sandboxing can be used to build secure systems from unsafe and untrusted components was present in Wahbe et al.'s original 1993 paper on SFI [8]. Over the last two and half years, we've been working with Mozilla on deploying SFI in production [1, 6] — to sandbox third-party libraries in the Firefox browser.

The hard part. In our experience the challenge with retrofitting a large system like Firefox with sandboxing is the *last mile*, ensuring that you properly verify and sanitize all the control and data-flow between Firefox and the now-untrusted library. This is hard because Firefox was written assuming libraries are trusted and thus the browser-library control and data flow are both unchecked and tightly coupled. Manually hardening the browser-library interface is slow and error-prone.

We address this with RLBox [6]. RLBox is a sandboxing framework that implements a coarse grained information flow type system to make the library-application boundary explicit. Type errors guide the migration effort, i.e., they elucidate when and where developers need to validate untrusted data that our framework could not check automatically.

In the tutorial, we'll look at an example showing why manually migrating existing applications to sandboxed libraries is difficult. Then, we will show how our RLBox sandboxing framework makes the migration process fast, easy, and secure. We will walk through how we can use RLBox in both simple examples as well large applications like Firefox (§IV).

II. Migrating to using sandboxed code is hard

To understand why migration is tedious and error-prone, consider retrofitting the Firefox browser to use a sandboxed libjpeg library to render JPEG images. To start, let's consider updating the Firefox `fill_input_buffer` decoder function:

```
1  void fill_input_buffer (j_decompress_ptr jd) {
2    struct jpeg_source_mgr* src = jd->src;
3    nsJPEGDecoder* decoder = jd->client_data;
4    ...
5    src->next_input_byte = new_buffer;
6    ...
7    if (/* check for errors */) {
8      decoder->mInfo.err->msg_code =
         ↪ JERR_OUT_OF_MEMORY;
9      ...
10   }
11   ...
12   memmove(decoder->mBackBuffer +
       ↪ decoder->mBackBufferLen,
13           src->next_input_byte,
               ↪ src->bytes_in_buffer);
14   ...
15 }
```

This function is called by the libjpeg library whenever it needs more image bytes from Firefox; unused input bytes are saved to an internal back buffer (line 12) for later use. To *securely* sandbox libjpeg, we must change this function to ensure that unsanitized data from the sandboxed libjpeg (e.g., function argument) cannot compromise the memory safety of the host Firefox process. Specifically, we need to:

- Sanitize `jd`, otherwise an attacker exploiting a libjpeg flaw could set `jd` to an arbitrary value. This would trick the host application into reading arbitrary memory with `jd->src` on line 2, giving the sandbox a "read gadget".
- Sanitize `src`, otherwise the attacker could similarly get a write gadget with `src->next_input_byte` on line 5, and then use the `memmove()` on line 12 as a read gadget.

978-1-6654-3170-5/21/$31.00 © 2021, Shravan Narayan. Under license to IEEE. 11
DOI 10.1109/SecDev51306.2021.00017
Authorized licensed use limited to: IEEE Xplore. Downloaded on July 26,2026 at 23:23:27 UTC from IEEE Xplore. Restrictions apply.

- Adjust pointer representations for `mInfo.err` — many SFI systems, including Native Client and WebAssembly, have different pointer representations, so we must translate (or swizzle) pointers accordingly.

This is only a partial list of necessary checks and modifications — as we describe in [6]. Worse, these checks are easy to miss and if (when) we do miss any necessary checks, an attacker can use this to break-out of the sandbox. But, beyond adding security checks, we also have to marshal data to and from the sandbox as well as adjust data structures to account for pointer representation and data layouts (ABI) differences between the application and sandbox. Only then can we run tests to ensure our retrofitting didn't break the application. In total, the upfront engineering effort of modifying a large application such as a browser this way is huge. We built the RLBox framework to address precisely this problem.

III. RLBox: The robust library sandboxing framework

RLBox is a sandboxing framework, implemented as a C++ library, designed to make it easier for developers to securely retrofit library sandboxing in existing applications. It does this by making data and control flow at the application-sandbox boundary explicit — using types — and by providing APIs to enforce security checks across the trust boundary.

RLBox mediates data flow using *tainted types* and control flow using APIs that make control transfers explicit. Tainted types are type wrappers that demarcate data originating from the sandbox, ensuring that application code does not use this data unsafely. For example, while application code can add two `tainted<int>`s (to produce another `tainted<int>`), it cannot branch on such values or use them as indexes into an array. Instead, the application must validate tainted values before it can use them. Similarly, the application must use the `sandbox_invoke` API to call functions in the sandbox. As we show next, this type-driven approach addresses both the security and engineering challenges we mentioned earlier.

Preventing sandbox breakouts. RLBox uses tainted types to (1) identify unsafe control- and data-flows, and (2) disallow unsafe flows by turning them into type errors or by automatically inserting security checks. Concretely, RLBox automatically sanitizes sandbox-supplied (tainted) pointers to ensure they point to sandbox memory, swizzles pointers that cross the trust boundary, and statically identifies locations where the developer must manually validate tainted data before it is used.

Minimizing engineering effort. RLBox is explicitly designed to simplify the task of migrating an application into a sandboxed form. First, RLBox uses tainted types to automatically handle ABI differences between the application and sandboxed library. Second, the framework allows developers to modify application code to use the RLBox API one line at a time and, in turn, allow developers to compile and test their application after each step. Finally, after each migration step, RLBox uses compile-time type errors to guide the developer to the next required change. Put together, this type-driven approach allows developers to migrate applications more easily and ensure that their modified application remains functionally equivalent in the process.

IV. Tutorial format

The tutorial will consist of two parts. The first part will cover the basics of sandboxing and RLBox. We will demonstrate how to use RLBox to secure a simple C++ application by isolation its dependencies (and their memory safety bugs). This will be hands-on: We will provide the code, compilers, and toolchains for the audience to follow along in real-time and get a feel for the RLBox API themselves.

In the second part, we will use RLBox to sandbox a library in a real application — the Firefox browser. The goal of this part is to help audience members understand of how they would deploy RLBox in their own (large) application. To this end, we will also work with the attendees to get them started sandboxing libraries in applications of their choosing.

Audience. This tutorial is for developers familiar with C++ who have an interest in secure system building, sandboxing, or browser security.

Open source. The material and code are open source and available online. The RLBox framework and documentation are available at https://rlbox.dev and https://docs.rlbox.dev, respectively. The WebAssembly sandbox and RLBox plugin we will use in the tutorial are available at https://wasm2c.rlbox.dev.

Acknowledgements

Parts of this tutorial are based on material presented at the USENIX Security Symposium 2020 [6], the Applied Research Competition at CSAW 2020 [5], among others. RLBox is a many-person effort; we thank our amazing collaborators, including Nathan Froyd, Tal Garfinkel, Bobby Holley, Mike Hommey, Sorin Lerner, Eric Rahm, Tom Ritter, Hovav Shacham, and Peter Van der Beken. This work is supported in by the NSF under Grant Numbers CCF-1918573 and CAREER CNS-2048262; and, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] N. Froyd. Securing Firefox with WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, Feb. 2020.

[2] T. Garfinkel, S. Narayan, C. Disselkoen, H. Shacham, and D. Stefan. The road to less trusted code: Lowering the barrier to in-process sandboxing. *USENIX ;login:*, 45(5), 2020.

[3] Matt Miller, Micorosoft Security Response Center. Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, Feb. 2019.

[4] Matt Miller, Micorosoft Security Response Center. Trends, challenge, and shifts in software vulnerability mitigation. https://bugs.chromium.org/p/project-zero/issues/detail?id=2164, Feb. 2021.

[5] S. Narayan. CSAW 2020 - applied research competition. https://github.com/shravanrn/shravanrn.github.io/raw/master/csaw-rlbox-award.pdf, Nov. 2020.

[6] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *USENIX Security Symposium*, 2020.

[7] The Chromium Projects. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety.

[8] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

Authorized licensed use limited to: IEEE Xplore. Downloaded on July 26,2026 at 23:23:27 UTC from IEEE Xplore. Restrictions apply.