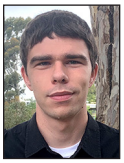
The Road to Less Trusted Code

Lowering the Barrier to In-Process Sandboxing

TAL GARFINKEL, SHRAVAN NARAYAN, CRAIG DISSELKOEN, HOVAV SHACHAM, AND DEIAN STEFAN

Tal Garfinkel is an independent researcher and consultant whose work focuses on the intersection of systems and security. He received his PhD from Stanford University in 2010 and is a co-founder of the USENIX Workshop on Offensive Technology. talg@cs.stanford.edu

Shravan Narayan is a fifth-year PhD student at UC San Diego working with Deian Stefan. His research focuses on in-process sandboxing, WebAssembly, browser security, and verified programming. He is the maintainer of the RLBox sandboxing framework. srn002@cs.ucsd.edu



Craig Disselkoen is a fifth-year PhD student at UC San Diego under Deian Stefan and Dean Tullsen. His research focuses on securing software through automatic vulnerability finding, program transformations, and secure runtimes. He is the author of the Haybale symbolic execution engine, written in Rust. cdisselk@cs.ucsd.edu

Hovav Shacham is a Professor of Computer Science at the University of Texas at Austin. His research interests are in applied cryptography, systems security, privacy-enhancing technologies, and technology policy. His work has been recognized with three "test of time" awards, including one at ACM CCS 2017 for his 2007 paper that introduced return-oriented programming. hovav@cs.utexas.edu

Firefox currently ships with a variety of third-party and in-house libraries running sandboxed using a new framework called RLBox. We explore how RLBox uses the C++ type system to simplify retrofitting sandboxing in existing code bases, and consider how better tooling and architecture support can enable a future where library sandboxing is a standard part of how we secure applications.

Users expect featureful software, and features, it hardly needs saying, come from code. The more features, the more code to implement them. And the more code, the more bugs—the more *security* bugs, in particular.

Whether it's the latest code rushed out before a marketing deadline, old code that hasn't been touched since the developer who wrote it retired, or a specialized module you licensed, attackers will scour them for bugs to use for exploiting your software and targeting your users.

The problem is especially acute with third-party open source libraries. You might care about one aspect of what the library does, but you ship the whole library, and bugs in any part of it can create security problems in your product. That is, unless you fork the library to remove the extraneous code, but who wants to maintain a fork forever? Worse, hackers who find a bug in a popular library can try to deploy it against every product that embeds the library—including yours.

Computer scientists have been thinking about software insecurity for 50 years, and they have come up with approaches to mitigate it. Rewrite your program (or parts of it) in a safer language! Refuse to ship new features and keep your program small! Formally verify the correctness of your software! "Privilege separate" your system by re-architecting it into multiple mutually distrusting processes! It's fair to say that none of these approaches has solved the problem. Insecure software is all around.

We believe that there is a practical path to improving software security. You can take software modules, including third-party libraries, and *sandbox* them to constrain what they can do—with low programmer effort, reasonable runtime overhead, and without wholesale rewriting or re-architecting—without even creating new OS processes. The sandboxed module will still have bugs, but those bugs will not (in most cases; see below) create security vulnerabilities in the enclosing program.

Consider an image decoding library like `libjpeg`. With sandboxing, we can restrict this library so it has access to the image it decodes and the bitmap it produces, *and that's it*. Or consider a spell-checking library like Hunspell. With sandboxing, we can restrict this library to just its dictionary and the text it checks. The application benefits from the library's features but doesn't inherit its security flaws.

Over the past two years we have worked with a team at Mozilla to build a tool, called RLBox, to support sandboxing and to migrate Firefox to a model where many third-party libraries run sandboxed. This new approach is now shipping in Firefox. Our experience suggests that once there is sufficient tooling support, then engineers can easily sandbox libraries, and they

The Road to Less Trusted Code: Lowering the Barrier to In-Process Sandboxing

Deian Stefan is an Assistant Professor of CSE at UC San Diego, where he co-leads the Security and Programming Systems groups. He received his PhD from Stanford University in 2016. Deian was a cofounder of Intrinsic, a web security start-up (acquired by VMware). His current research lies at the intersection of secure systems, programming languages, and verification. deian@cs.ucsd.edu

become increasingly comfortable with and excited by the opportunities this offers. For example, while the initial target of our sandboxing collaboration was a third-party font-shaping library, Graphite, now Firefox developers and security engineers are using RLBox to sandbox both third-party libraries and legacy Mozilla code in domains like media decoding, spell checking, and even speech synthesis.

We believe that the opportunities extend far beyond Firefox. After all, secure messaging apps (e.g., Signal, WhatsApp, and iMessage), servers and runtimes (e.g., Apache and Node.js), and enterprise tools (e.g., Zoom, Slack, and VS Code) also rely on third-party libraries for various tasks—from media rendering, to parsing network protocols like HTTP, image processing (e.g., to blur faces), spell checking, and automated text completion. With RLBox, these systems' developers are empowered to sandbox modules and limit the damage their bugs can cause.

Recent advances in compilers and processor architectures have made efficient in-process isolation increasingly practical. As it turns out, though, preventing a module from reading or writing memory outside its data region isn't enough. Our initial efforts in manually sandboxing Firefox libraries are a case in point. Firefox had been written under the assumption that the libraries were trustworthy. Even when isolated, they could return data values that would cause the (unsandboxed) Firefox code to take unsafe actions, a scenario that security researchers describe as a confused deputy attack. We tried to add code to manually check return values for consistency, but repeatedly found that we had missed cases and left open avenues for attack.

That's where RLBox comes in. Using the C++ type system, RLBox automatically generates the boilerplate code required for sandbox interaction, and identifies *all* places where the programmer will have to add data-checking code. With RLBox, programmers have a framework that makes it easy to sandbox libraries (1) *securely*, ensuring the interface between the untrusted library and the application code is correct, and (2) with *minimal engineering effort*, so that the cost of migrating libraries and applications to sandboxing is not prohibitive.

In the rest of this article we describe the experience that led to RLBox, how RLBox works, how it leverages the C++ type system to make sandboxing practical, and how our type-driven approach can be used in other domains (e.g., trusted execution environments). Then we outline how this approach can translate to languages other than C/C++. Finally, we end with a vision of what software development could look like with broader first-class support for sandboxing.

Before closing, we should note that sandboxing is not a panacea. Some components must be *correct*, not just isolated, for the system as a whole to be secure. The JavaScript just-in-time compilers used by Web browsers are a notorious example. With RLBox, you can sandbox everything else, and focus developer time on getting these few critical modules right.

The Road to RLBox: Library Sandboxing in Firefox

Firefox, like other browsers, relies on dozens of third-party libraries to decode audio, images, fonts, and other content. These libraries have been a significant source of vulnerabilities in the browser (e.g., most of the vulnerabilities found by recent work using symbolic execution were in third-party libraries [2]). With collaborators at Mozilla, we sought to minimize the damage due to vulnerabilities in libraries by retrofitting Firefox to sandbox these libraries.

When we began this project roughly two years ago, we thought the hardest part would be adapting Google's Native Client (NaCl), a software-based isolation (SFI) toolkit, to sandbox libraries. NaCl is designed for sandboxing programs, not libraries. This turned out to be the easy part. Since then, WebAssembly (Wasm) toolkits—in particular the Lucet Wasm compiler—have made this even easier [5].

In fact, the hardest part was the *last mile*, retrofitting Firefox to account for the now-untrusted libraries. Firefox was written assuming libraries are trusted. To add sandbox-

ing, we had to change its threat model to assume sandboxed libraries are untrusted, and harden the browser-library interface. Hardening this interface in turn required sanitizing data and regulating control flow between sandboxed libraries and the browser, thus ensuring that malicious libraries could not break out of their sandbox.

Our first attempt at sandboxing libraries in Firefox involved manually hardening the library-application interface—this did not go well.

Security challenges. To see how things can go wrong, let's consider updating the `fill_input_buffer` JPEG decoder function. `libjpeg` calls this function whenever it needs more bytes from Firefox. As seen on line 16 of Listing 1, Firefox also saves the unused input bytes held by `libjpeg` to an internal back buffer, which it sends to `libjpeg` along with the new input bytes.

```
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 (/* buffer is too small */) {
8:     JOCTET* buf = (JOCTET*) realloc(...);
9:     if (!buf) {
10:      decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
11:      ...
12:    }
13:    ...
14:  }
15:  ...
16:  memmove(decoder->mBackBuffer + decoder->mBackBufferLen,
17:      src->next_input_byte, src->bytes_in_buffer);
18:  ...
19: }
```

Listing 1

When sandboxing `libjpeg`, we need to make the following changes:

- Sanitize `jd`, otherwise the read of `jd->src` on line 2 could become a read gadget.
- Sanitize `src`, otherwise the write to `src->next_input_byte` on line 5 becomes a write gadget and the `memmove()` on line 16 becomes an arbitrary read gadget.
- Sanitize `jd->client_data` on line 3 to ensure it points to a valid Firefox `nsJPEGDecoder` object; otherwise invoking a virtual method on it will hijack control flow.
- Sanitize the nested pointer `mInfo.err` on line 10 prior to dereferencing, else it becomes a write gadget.
- Sanitize the pointer `decoder->mBackBuffer + decoder->mBackBufferLen` used on the destination address to `memmove()` on line 16 to prevent overflows of the unused byte buffer.
- Adjust pointer representations for `mInfo.err` and `decoder->mBackBuffer`—both NaCl and Wasm have different pointer representations and we must translate (swizzle) these pointers accordingly.
- Ensure that multiple threads can't invoke the callback on the same image; otherwise we have a data race that results in a use-after-free vulnerability on line 8.

If we miss any of these checks—and these are only a limited sample of the kind of checks required [4]—an attacker could potentially bypass our sandbox through a confused deputy attack. Adding these checks to the hundreds of Firefox functions that use `libjpeg` was tedious. Worse, we frequently found checks we had overlooked.

Engineering effort. The upfront engineering effort of modifying the browser this way was huge. Beyond adding security checks, we also had to retrofit all library calls, adjust data structures to account for machine model (ABI) differences between the application and sandbox (a common issue with SFI toolchains), marshal data to and from the sandbox, etc. Only then could we run tests to ensure our retrofitting didn't break the application. Finally, since Firefox runs on many platforms—including platforms not yet supported by SFI toolkits like NaCl and Wasm—we had to do this alongside the existing code that uses the library unsandboxed, using the C preprocessor to select between the old code and the new code. The patches to do all this became so complicated and unwieldy that we couldn't imagine anybody maintaining our code, so we abandoned this manual approach, built RLBox, and started anew.

The RLBox Framework

RLBox is 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 both mediate these flows and enforce security checks across the trust boundary.

RLBox mediates data flow using *tainted types*—it uses type wrappers to demarcate data originating from the sandbox, and ensure that application code cannot 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.

RLBox mediates control flow with explicit APIs for control transfers. Calls into the sandbox must use `sandbox_invoke(sbx_fn, args...)`. Callbacks into the application can only use functions registered with the `sandbox_callback(app_fn)` API. These APIs also impose a strict data flow discipline by forcing all sandbox function return values, and callback arguments, to be tainted.

As we show next, this tainted-type-driven approach addresses both the security and engineering challenges we outline above.

Using Tainted Types to Eliminate Confused Deputy Attacks

RLBox eliminates confused deputy attacks by turning unsafe control- and data-flows into type errors and, where possible, by performing automatic 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 tainted data must be validated before use.

Consider, for example, the JPEG decoder callback from before. RLBox type errors would guide us to (1) mark values from the sandbox as tainted (e.g., the `jd` argument and `src` variable on line 2, Listing 2) and (2) *copy and verify* (otherwise tainted) values we need to use (e.g., `jd->client_data` on line 3, Listing 2).

```
1: void fill_input_buffer (rlbox_sandbox& sandbox,
     tainted<j_decompress_ptr> jd) {
2:   tainted<jpeg_source_mgr*> src = jd->src;
3:   nsJPEGDecoder* decoder =
     jd->client_data.copy_and_verify(...);
4:   ...
5:   src->next_input_byte = new_buffer;
6:   ...
7:   if (/* buffer is too small */) {
8:     JOCTET* buf = (JOCTET*) realloc(...);
9:     if (!buf) {
10:      decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
11:      ...
12:    }
13:    ...
14:  }
15:  ...
16:  size_t nr = src->bytes_in_buffer.copy_and_verify(...));
17:  memmove(decoder->mBackBuffer + decoder->mBackBufferLen,
18:      src->next_input_byte.copy_and_verify(...), nr);
19:  ...
20: }
```

Listing 2

In Listing 2, we need to write validators as C++ lambdas to the `copy_and_verify` method used on lines 3, 16, and 18. As we describe in [4], validators fall into one of two categories: preserving application invariants (e.g., memory safety) or enforcing library invariants. On line 3, for example, we must ensure that `decoder` points to a valid `nsJPEGDecoder` object not used by a concurrent thread, while on line 16 we need to ensure that copying `nr` bytes won't read past the `mBackBuffer` bounds.

We must get validators right—a bug in a validator is often a security bug. In practice, though, validators are rare and short. The six libraries we sandboxed in [4] required 2–14 validators each, and these validators averaged only 2–4 lines of code. Most importantly, by making these validators explicit, RLBox makes code reviews easier: security engineers only need to review these validators.

What's missing in Listing 2 is almost as important: we don't write any security checks on lines 2, 5, and 10, for example. Instead, RLBox uses runtime checks to automatically swizzle and sanitize the `src`, `src->next_input_byte`, and `decoder->mInfo.err` pointers to point to sandbox memory.

Using Tainted Types to Minimize Engineering Effort

Manually migrating an application to use library sandboxing is labor intensive and demands a great deal of specific knowledge about the isolation mechanism. RLBox abstracts away many of these specifics, making migration relatively simple and mechanical.

Incremental migration. While RLBox automates many tasks, we still need to change application code to use RLBox. In particular, we need to add a trust boundary at the library interface by turning all control transfers (i.e., library function calls and callbacks) into RLBox calls, and we need to write validators to sanitize data from the library, as we saw above. Making these changes all at once is frustrating, error-prone—overlooking a single change might suddenly result in crashes or more subtle malfunctions—and hard to debug.

RLBox addresses these challenges with *incremental migration*, allowing developers to modify application code to use the RLBox API one line at a time. A full migration involves multiple steps and is explained further in our paper [4]. However, the key idea is that RLBox provides *escape hatches* which let developers temporarily disable some checks while migrating their application code. Thus, at each step, the application can be compiled, run, and tested.

RLBox provides two escape hatches:

1. The **UNSAFE_unverified API** allows developers to temporarily remove the tainted type wrapper (e.g., to run and test their code). As the application is ported, calls to `UNSAFE_unverified` APIs are removed or replaced with validator functions that correctly sanitize tainted data.

2. The **RLBox noop sandbox** provides a pass-through sandbox that redirects function calls back to the unsandboxed version of the library, while still wrapping data as if it were received from a sandboxed library. This allows developers to use the RLBox APIs and test data validation separately from the actual isolation mechanism.

Compile-time type errors guide the developer by pointing to the next required code change—e.g., data that needs to be validated before use, or control transfer code that needs to change to use the RLBox APIs. By the end of the process, the application is still fully

functional, all the escape hatches have been removed, and the application-library interface has fully migrated to using tainted types.

We found that incremental migration greatly simplified the code review process. In Firefox, we could commit and get reviews for partial migrations to the RLBox API, since the Firefox browser continued to build and run as before. Additionally, we could explicitly include security reviews when writing the data validators for tainted data.

Beyond migration, we also found the noop sandbox to be useful for selectively enabling library sandboxing in conditional builds. For example, while Firefox on Linux and OS X uses Wasm for isolation, the Lucet Wasm compiler's support for Windows is incomplete and thus Firefox uses the noop sandbox on Windows builds; once Windows support is complete, a single line change will allow us to take advantage of the sandbox. This is useful beyond Firefox too: developers of the Tor Browser (a downstream project of Firefox for anonymous web browsing) are interested in sandboxing more libraries than mainline Firefox, since Tor users typically have a higher security-performance threshold. Using the noop sandbox will allow Tor developers to contribute upstream changes to sandbox libraries in mainline Firefox, using the noop sandbox to avoid noticeable overhead. Tor developers can then selectively enable additional sandboxing (again) with a one-line change, rather than having to maintain a major fork.

ABI translations. Isolation mechanisms can have different machine models and ABIs from the rest of the application. For example, Wasm uses a 32-bit machine model meaning that pointers, `int`s, and `long`s are 32 bits. However, this is a different machine model from that used by the host application. Handling such differences manually is laborious and error-prone.

Consider line 10 from the previous `fill_input_buffer` example in Listing 2:

```
// mInfo is an object of type jpeg_decompress_struct
decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
```

If we port this manually, the resulting code would be:

```
auto err_field = adjust_for_abi_get_minfo_field(decoder
      ->minfo, "err");
auto err_field_swizzled = adjust_for_abi_convert_pointer
      (err_field);
auto msg_field = adjust_for_abi_get_err_field
      (*err_field_swizzled, "msg_code");
assert(in_sandbox_memory(msg_field));
      // Ensure pointer is in sandbox memory
auto msg_field_swizzled = adjust_for_abi_convert_pointer
      (msg_field); // Assign the value
*msg_field_swizzled = adjust_for_abi(JERR_OUT_OF_MEMORY);
```

In contrast, RLBox requires no changes other than marking `mInfo` as tainted. RLBox automatically transforms pointers, and accounts for the difference in the size of `long` and `pointers`:

```
// mInfo is an object of type tainted<jpeg_decompress_struct>
decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
```

RLBox is able to abstract and automatically reconcile ABI differences since all control and data flow goes through its APIs and tainted types.

Using Tainted Types Outside of Library Sandboxing

The security challenges we face when sandboxing libraries are not unique to library sandboxing. Developers have to handle untrusted data and control flow in many other domains—and our tainted-type approach can help. We give three examples:

TEE runtimes. Applications running in trusted execution environments (TEEs), like Intel's SGX and ARM's TrustZone, interface with untrusted code by design—TEEs even consider the OS untrusted. Getting this code right is hard. And, indeed, TEE runtimes contain similar bugs: Van Bulck et al. [1], for example, found that most frameworks, across several TEEs, were vulnerable to bugs RLBox addresses by construction.

OS kernels. Operating system kernels handle untrusted data from userspace. Bugfinding tools—from MECA at the start of the century [10] to Sys this year [2]—have found many vulnerabilities in kernels due to unchecked (or improperly checked) userspace data (notably, pointers). Frameworks like RLBox could automatically identify where userspace data needs to be checked and even perform certain checks automatically (e.g., much like we ensure that sandbox pointers point to sandbox memory, we can ensure that userspace pointers point to userspace memory). Indeed, Johnson and Wagner's bugfinding tool [3] even used type inference to find such kernel bugs.

Browser IPC layers. Modern browser architectures privilege separate different parts of the browser into sandboxed processes. Almost all separate the *renderer* parts—the portion of the browser that handles untrusted user content from HTML parsing, to JavaScript execution, to image decoding and rendering—from the *chrome* parts—the trusted portion of the browser that can access the file system, network, etc.—and restrict communication to a well-typed inter-process communication (IPC) layer. Like OS kernels, the browser chrome must validate all values coming from untrusted renderer processes; like kernels, browsers have been exploited because of unchecked (and improperly checked) untrusted data. Here, again, tainted types can help—and as a step in this direction, Mozilla started integrating tainted types into the Firefox IPC layer, as part of the IPDL (IPC protocol definition language) used to generate boilerplate code for sending and receiving well-typed IPC messages [7].

This list is by no means exhaustive; others have similarly observed that tainting can be used to catch and prevent bugs when handling untrusted data (e.g., see [9]).

Beyond RLBox

We have thus far discussed RLBox in its current form—a framework that uses the C++ type system, template metaprogramming, and SFI toolkits like Wasm to securely sandbox libraries typically written in C. In the future, we hope to see extensions to other languages, support for sandboxing libraries written in arbitrary languages, and the adoption of processor features that can further lower in-process sandboxing overheads.

Beyond C++

We implemented RLBox in C++ because Firefox is predominantly written in C++. To extend RLBox to other languages, we need to understand how to implement RLBox's tainted type system.

Our C++ implementation uses templates to implement the generic `tainted<T>` type and takes advantage of function and operator overloading to make most of the tainted type interface transparent. For example, RLBox overloads pointer dereferencing—the `->` and `*` operators—to allow dereferencing `tainted<T*>` values safely by automatically sanitizing the underlying pointer to point to sandbox memory (line 10 in Listing 2). We also use template metaprogramming to enforce a custom type discipline.

Many languages have features that are expressive enough to implement our tainted type system directly or as part of the language toolchain, for example, with compiler plugins.

Statically typed languages. RLBox is a natural fit for languages that already enforce type safety statically. Statically typed languages typically offer some form of generics or templates that can be used to implement tainted types. Many also allow function and operator overloading which, like C++, would allow us to provide safe operations on tainted types while preserving the original syntax of the language.

Rust is a particularly compelling language. First, Rust's raison d'être is safety—indeed, the language is used in many settings where assurance is paramount—and RLBox can complement Rust's safety by, for example, making it easy for Rust programmers to safely integrate C/C++ code into their projects, which today is considered unsafe. Second, Rust's macro system and support for generics and operator overloading via traits allows tainted types to be implemented directly in the language. Finally, Rust's affine types can even simplify certain RLBox validators, like the validators used to prevent time-of-check to time-of-use and double fetch attacks [4].

Dynamically typed languages. In dynamically typed languages like JavaScript and Python, we can enforce tainted types dynamically. This, of course, makes the incremental porting loop longer since type errors will only manifest at runtime. Luckily, many dynamically typed languages have typed extensions to precisely address this limitation. For example, TypeScript and Flow extend JavaScript with static type annotations.

Compiler plugins and toolkits. For languages not flexible enough to implement the RLBox tainted type system statically, we envision implementing the type system as part of language toolchains. For example, for C, we can implement RLBox as a Clang plugin (both to enforce the type system and to generate runtime checks). Alternatively, we can implement tainted types as part of interface description language (IDL) compilers. As mentioned above, for example, the Mozilla security team is integrating tainted types into the Firefox IPDL inter-process communication protocol IDL [7].

Beyond Software-Based Isolation

We designed RLBox to make it easy for developers to plug in different isolation mechanisms. This makes it easy to migrate code (e.g., by using the noop sandbox), as we have described. It also allows developers to use different isolation mechanisms that have different tradeoffs. For example, while in production we use Wasm for isolation, in [4] we evaluate two other isolation mechanisms: NaCl and traditional process-based isolation. These isolation mechanisms have different tradeoffs. Process isolation is simple but scales poorly—protection boundary crossing costs become prohibitive as the number of sandboxes exceed the number of available cores. Wasm and NaCl, on the other hand, scale to a large number of sandboxes and have cheap boundary crossings, but they impose an overhead on the sandboxed code.

At present, Wasm toolchains offer a practical and portable path to isolation. But this software-based isolation approach will inevitably be slower than running native code.

Hardware support for in-process isolation can offer solutions that are simple and more performant. Today, for example, Intel's Memory Protection Key (MPK) features incur roughly 1% overhead when used for in-process isolation [8], but this doesn't scale beyond 16 sandboxes. In the future, the CHERI capability-based system will similarly make in-process isolation—and memory safety more generally—cheap on ARM processors [6]. By making it easy to use these features transparently (e.g., for CHERI it can automatically adjust for ABI differences introduced by capabilities), RLBox could lower the barrier to adopting new hardware isolation features—and, we hope, this will encourage new hardware design for in-process isolation.

Bringing Sandboxing to the Developer Ecosystem

While RLBox has been a boon for our work in Firefox, it's just a starting point. Our hope is that library sandboxing will become a first-class activity in future development environments, and that RLBox's capabilities will ultimately be subsumed by standard parts of tomorrow's languages, toolchains, and package managers. We believe in many cases such support could allow the use of sandboxed libraries with a level of ease comparable to the use of unsandboxed libraries today.

FFIs and native code. Many popular safe languages such as Python, Ruby, and JavaScript make extensive use of native (typically C) code in their standard libraries and package ecosystems via foreign function interfaces (FFIs). Unfortunately, bugs in native code can completely break all high-level safety guarantees. Extending FFI interfaces and interface generation tools with first-class support for sandboxing native code is very natural—both because the FFI boundary is explicit and because developers are used to writing code that spans trust boundaries.

Package managers. In the ecology of package ecosystems there is constant competition between package authors to provide the best package for a given task. Security is among the ways that package authors have recently started differentiating their package from others. We have seen this clearly in the Rust ecosystem, where the presence (or absence) of unsafe code is one way that packages are compared.

Sandboxing is another way that package authors can provide differentiated value, by integrating sandboxing support into their library. This could look like authors distributing their packages with most or all of the work required to sandbox that package done upfront by the package author. Developers could then choose whether or not to enable sandboxing with minimal additional fanfare.

To facilitate this, the package author could specify a system level sandboxing policy (e.g., as a manifest file requesting access to parts of the file system or network), and developers could then choose if and how to grant these privileges when importing a package. Much of the work of writing validators for tainted types could also be mitigated by distributing validators as part of a sandboxed library. We even envision an ecosystem of sandbox interface declarations for existing packages, much like TypeScript type declarations for JavaScript packages, which will allow to developers to pull sandboxed interfaces much like they consume type declarations today.

Conclusion

Decades of attempts to detect and mitigate software vulnerabilities have yielded lackluster results. Even browsers, some of the most heavily targeted and scrutinized software, seem to provide an inexhaustible stream of exploitable vulnerabilities. In-process sandboxing can offer developers and security engineers another choice—moving code, especially legacy and third-party code, out of their trusted computing base by sandboxing it, thus mitigating the impact of a compromise.

We developed RLBox to make sandboxing practical. It is currently being used to sandbox third-party and in-house libraries in Firefox, and we hope that other C++ projects will choose to adopt it. Looking further, we hope to collaborate with developers of programming languages (and their toolchains and standard libraries), package managers, and processor architects to provide first-class support for in-process sandboxing. Small changes to make in-process sandboxing first-class can result in huge benefits for developers and security engineers.

References

[1] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, pp. 1741–1758: https://people.cs.kuleuven.be/~jo.vanbulck/ccs19-tale.pdf.

[2] F. Brown, D. Stefan, and D. Engler, "Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, pp. 199–216: https://www.usenix.org/conference/usenixsecurity20/presentation/brown.

[3] R. Johnson and D. Wagner, "Finding User/Kernel Pointer Bugs with Type Inference," in *Proceedings of the 13th USENIX Security Symposium (USENIX Security '04)*: https://www.usenix.org/event/sec04/tech/full_papers/johnson/johnson_html/.

[4] 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 *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, pp. 699–716: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[5] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: WebAssembly as a Practical Path to Library Sandboxing," arXiv, December 4, 2019: https://arxiv.org/abs/1912.02285.

[6] R. Grisenthwaite, "A Safer Digital Future, by Design," ARM Blueprint, October 18, 2019: https://www.arm.com/blogs/blueprint/digital-security-by-design.

[7] T. Ritter, "Support Tainting Data Received from IPC," Mozilla Bug 1610005, January 2020: https://bugzilla.mozilla.org/show_bug.cgi?id=1610005.

[8] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK)," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pp. 1221–1238: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[9] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," in *Proceedings of the 15th USENIX Security Symposium (USENIX Security '06)*, pp. 121–136: https://www.usenix.org/legacy/event/sec06/tech/full_papers/xu/xu_html/.

[10] J. Yang, T. Kremenek, Y. Xie, and D. Engler, "MECA: An Extensible, Expressive System and Language for Statically Checking Security Properties," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, pp. 321–334: https://web.stanford.edu/~engler/ccs03-meca.pdf.