Short paper: Superhacks

Exploring and preventing vulnerabilities in browser binding code

Fraser Brown
Stanford University

Deian Stefan
UC San Diego and Intrinsic

Abstract

In this paper, we analyze security vulnerabilities in the binding layer of browser code, and propose a research agenda to prevent these weaknesses with **(1)** static bug checkers and **(2)** new embedded domain specific languages (EDSLs). Browser vulnerabilities intermittently leak browsing data and sometimes allow attackers to completely compromise users' systems. Some of these security holes result from programmers' difficulties with managing multiple tightly-coupled runtime systems—typically JavaScript and C++. In this paper, we survey the vulnerabilities in code that connects C++ and JavaScript, and explain how they result from differences in the two languages' type systems, memory models, and control flow constructs. With this data, we design a research plan for using static checkers to catch bugs in existing binding code and for designing EDSLs that make writing new, bug-free binding code easier.

1. Introduction

Browsers are notoriously difficult to secure—vendors like Google and Mozilla invest millions of dollars and hundreds of engineers to fortify them. Google's Vulnerability Rewards Program, which pays bounties for bugs in products like the Chrome browser, has awarded over six million dollars since its inception [44, 51]. But, as both dangerous security breaches and more benign hacking contests demonstrate, browsers are still rife with exploitable bugs.[1]

While many factors contribute to browser insecurity, from ill defined security policies to bugs in JavaScript engines and sandboxing mechanisms, this paper focuses on bugs in the browser engine *binding layer*. Bindings allow code written in one language to communicate with code written in another. For example, Chrome's browser engine Blink uses C++ for performance-critical components like the HTML parser and rendering algorithms, while choosing JavaScript for application-specific functionality; binding code connects the two layers. This allows browser developers to take advantage of JavaScript for safety and usability, transitioning to C++ when they need, say, the support for input or output that JavaScript lacks.

Browser binding code is different from—and more complicated than—other multi-language systems, since it must reconcile different runtimes instead of making simple library calls (as a foreign function interface does). Furthermore, because the JavaScript running in browsers comes from many disparate, potentially malicious actors, binding code developers must struggle to write difficult defensive code. For example, their binding

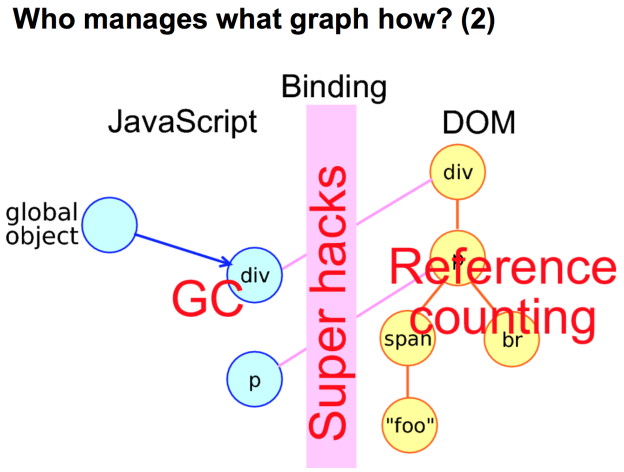

Figure 1: Slide from [29] illustrating the state of memory management code between JavaScript and C++ in Blink.

code must validate arguments when handling a down-call from JavaScript into C++, and grant access to an object only if this access is permitted by the same origin policy [2]. These checks are not unique to this *browser* layer, however; since browsers run in a necessarily adversarial environment, most of their components are defensive.

Certain challenges, though, are specific to writing secure browser binding-layer code, and these challenges stem from the impedance mismatch between C++ and JavaScript language abstractions; C++ and JavaScript's memory models, control structures, and types are different enough that incorrectly managing their interactions easily introduces bugs. For example, because JavaScript is garbage collected and C++ is not, developers may fail to free C++ objects when their reflected JavaScript objects are collected, creating memory leaks. On the other hand, they may free C++ objects when references to those objects still exist in either C++ or JavaScript, introducing use-after-free vulnerabilities

Unfortunately, the existing binding layer APIs actively work against developers attacking security challenges; Figure 1, taken from a core Chrome developer's presentation [29], refers to binding code as "super hacks." Rethinking and understanding the binding layer is crucial for users' security; it is also timely, as companies roll out new security-focused engines like Servo, as more developers depend on related platforms like Node.js, and as IoT enthusiasts expose hardware to JavaScript applications.

In this paper, we study the kinds of security vulnerabilities that appear in today's browser binding code (Section 2), an area that has received very little attention from the academic

[1] https://en.wikipedia.org/wiki/Pwn2Own

community. In response to these vulnerabilities, we propose a research agenda (Section 3) to address some challenges of writing secure binding code by **(1)** developing light-weight, browser-specific, multi-language static bug checkers and **(2)** designing new embedded domain specific languages that address the impedance mismatch between JavaScript and C++.

2. Understanding binding layer CVEs

We sieved through a list of Chrome CVEs to discover buggy patterns that checkers can recognize and the DSLs can disrupt [23].[2] This section categorizes and distills the binding-specific buggy patterns that we found. We ignore traditional bugs, like C++ buffer overflows, and focus on binding bugs for two reasons. First, traditional checkers can detect traditional C++ bugs—the problem is getting people to adopt them. Binding layer bugs, on the other hand, are checker-less; there are no checkers for anyone to adopt. Second, since new browsers, like Servo, forgo C++ altogether, C++-specific checkers and language designs are useless to them; any tools for these browser must depend on less language specific insights.

Most of the CVEs we surveyed fit into one of three categories: memory (mis-)management, type (mis-)representation, and (invalid) control flow. In this section, we illustrate how easy it is to introduce these vulnerabilities by creating the misinformed, fictional Blob object. We want to expose FileAPI-like Blobs [43] to JavaScript; we do not write them in JavaScript because C++ performs better, and Blobs should efficiently represent the binary data that we use to communicate with a server. The WebIDL [39] interface describing Blob is:

```
[Constructor(DOMString[] blobParts)]
interface Blob {
  readonly attribute unsigned long size;
  Blob slice(optional unsigned long start,
             optional unsigned long end)
  boolean equals(Blob other);
};
```

Chrome can expose this interface to JavaScript by using V8's `Set` function to set the `"Blob"` property on the global object. This allows users to create Blobs from an array of strings: `new Blob(["S", "H"])`. It also allows code to check the byte-size of a Blob (`blob.size`), compare two Blobs' contents (`blob1.equals(blob2)`), and extract subsets (`blob.slice(2)`).

Whenever someone calls the JavaScript Blob constructor, the V8 JavaScript engine creates a reflector object, a JavaScript object exposed to C++. Think of a reflector as a JavaScript facade; each time someone knocks on the door—makes a JavaScript call—they produce actions in the house itself—the underlying C++ object. When calling the Blob constructor from JavaScript, V8 creates the reflector and calls the underlying C++ binding constructor code with it. The constructor converts the `blobParts` JavaScript strings to raw C++ strings and creates a `Blob` C++ object to encapsulate them. Then, using the V8 binding API, it stores a pointer to the C++ object in a hidden internal field of the reflector object and returns it to JavaScript.

Invoking the `equals` method—or any method—on a reflected `Blob` object kicks off a V8 call to a C++ binding-layer function. This function extracts the C++ `Blob*` pointers from the hidden fields of the JavaScript receiver (`this`) and `other` argument. Then, it calls the C++ method `Blob::equals` and returns its boolean result, casted to a `v8::Value`.

This process is error-prone to explain in prose and even more so to execute; in the next three paragraphs, we will outline memory, type, and control-flow related bugs in `Blob` binding code.

Avoiding memory leaks means destroying C++ `Blob` objects each time their reflected JavaScript objects are collected. We do this by registering the correct callback with V8's garbage collector; this function should remove the C++ `Blob` whenever its reflector is collected. Calling the wrong callback (or nothing at all) will introduce a memory leak.

To prevent out-of-bounds vulnerabilities we must ensure that `Blob`'a `slice` function is called with valid bounds; though this process seems simple at first, we must cast correctly from JavaScript `number`s to C++ `unsigned long`s. Furthermore, since JavaScript allows methods to be called on arbitrary objects (e.g. `Blob.slice.call({})`), we must make sure that `Blob` C++ binding functions check that they are called on valid receivers. Otherwise, extracting a raw pointer from an internal hidden field and casting it to a `Blob*` could lead to an invalid memory access (or code execution, depending on the receiver).

Unfortunately, avoiding type-related problems is not enough; we must also make sure that C++ and JavaScript's interaction does not introduce unexpected and buggy control flow. For example, when using the V8 API to raise a JavaScript `TypeError` in the binding implementation of `slice`, we must ensure that the exceptional code follows JavaScript's control flow discipline and not C++'s stack discipline. Otherwise we might execute unintentional, potentially dangerous code [36].

These memory, type, and control flow issues (even for simple `Blob`) are not exhaustive; in the next sections, we present real-world CVEs that exploit more complicated bugs.

CVEs caused by memory mis-management The memory-related CVEs in this section appear in Google's rendering engine Blink, which creates a visual representation of a webpage from a URL. Blink must store DOM objects in order to render them onscreen. To allow application code to modify page content and structure, these objects are also exposed to JavaScript, much like our Blobs. The binding code for the DOM and other web APIs, however, is mostly generated: a WebIDL compiler takes WebIDL specifications and C++ templates as input and turns them into binding code [7].

Despite this, Blink has struggled to handle the differences between memory management on the Blink (reference counted) and V8 (garbage collected) sides of the binding; at one point, ten percent of their tests were leaking memory, and "almost all" of their crashes were due to use-after-free errors [28]. Not all of

[2] Although we focus on the binding layer in Chrome's Blink rendering engine in this paper, we don't mean to single them out unfairly; binding layers in other browsers are at least as complex and struggle with similar issues. See, for instance [31] for comments on the complexity of Mozilla's XPConnect.

these issues were a result of binding code—some were due to the inherent challenges of reference counting itself—but binding troubles certainly contributed to their 2012 decision to create Oilpan, a garbage collector for Blink.

Oilpan did not land until December 2015; in the mean time, CVE-2014-1713, a use-after-free vulnerability in Blink's binding code, led to a full execution-outside-sandbox exploit with a 100,000 dollar bounty [6, 10]. The source of this bug was in Blink's C++ template code for attribute setters [11]:

```
234 {{cpp_class}}* proxyImp =
      ↪   {{v8_class}}::toNative(info.Holder());
235 {{attribute.idl_type}}* imp =
      ↪   WTF::getPtr(proxyImp->{{attribute.name}}());
...
247 {{attribute.v8_value_to_local_cpp_value}};
...
251 ASSERT(imp);
```

This code looks strange at first, but, when building Chrome, the WebIDL compiler backend fills in template variables like `{{cpp_class}}` with information from WebIDL interface files, creating a normal C++ file. Once filled out, line 234 will create `proxyImp`, an object of whatever type the WebIDL file specifies. In the CVE, this was a Document object. On line 235, the object's location attribute will be assigned to the raw pointer `imp`. Unfortunately, the code on line 247 may perform an up-call (e.g. to call a JavaScript define `toString` function). At this point, malicious JavaScript can remove any references to the reflected attribute object and force V8 to GC. Since Blink registers GC callbacks to clean up the C++ objects backing reflected objects, this will result in `imp` getting freed—to the callback code, it appears like there are no live references to the object. Thereafer (line 251), using `imp` will result in a use-after-free crash. Making `imp` a `RefPtr`—as opposed to a raw pointer—fixes this problem, since the GC callback will only decrement the refcount of the `RefPtr`, instead of freeing blindly. Now, `imp` is not freed until both its JavaScript *and* C++ references are dead.

The use-after-free problem is pervasive in binding code. CVE-2015-6789 (high severity, fixed in 47.0.2526.80) is similar to the bug we just outlined. This vulnerability takes place in the `MutationObserverInterestGroup` private constructor, which keeps track of which `MutationObserver`s are observing (and reacting to) changes in the same DOM objects. The interest group tracks related observers using a hash map of raw pointers to `MutationObserver`s. Predictably, these observers may be touched and garbage collected by JavaScript, leading to more use-after-free vulnerabilities. The fix, again, is to use owning `RefPtr`s instead of `RawPtr`s—`RefPtr`s, with appropriate V8 GC callbacks, allow liveness information to pass between V8's collection and Blink's reference counting; garbage collection becomes a layer over a general reference counting scheme. One developer even suggested "hold[ing] `RefPtr`s to all objects in bindings code," but the change was deemed far too expensive. Instead, Blink is transitioning to garbage collection—consistent with V8—and, while this is addressing many issues, it would be interesting to also see what sorts of bugs this transition may also be introducing.

CVEs caused by incorrect types The vulnerabilities in this section all involve V8 code that either incorrectly casts to JavaScript types or uses JavaScript objects of the wrong type. Since JavaScript values at the binding layer are exposed as `v8::Value`s, code can fail by, for example, incorrectly casting such a value to a `v8::Function`, when it is in fact not a function, and then calling it. The JavaScript to C++ translation process can yield more unexpected errors, though, like CVE-2015-1217 [14, 15]:

```
134 String listenerSource = // code from m_source url
135 String code = "function () { ... return function(" +
      ↪   m_eventParameterName + ") {" + listenerSource +
      ↪   "\n};}";
136 v8::Handle<v8::String> codeExternalString =
      ↪   v8String(isolate(), code);
137 v8::Local<v8::Value> result = V8ScriptRunner::
      ↪   compileAndRunInternalScript (codeExternalString,
      ↪   isolate(), m_sourceURL, m_position);
138 ASSERT(result->IsFunction());
139 v8::Local<v8::Function> intermediateFunction =
      ↪   result.As<v8::Function>();
140 v8::Local<v8::Value> innerValue =
      ↪   V8ScriptRunner::callInternalFunction
      ↪   (intermediateFunction, thisObject, 0, 0,
      ↪   isolate());
```

This code snippet comes from `prepareListenerObject`, a function that creates a JavaScript reflector for an event listener. In the first line, the JavaScript source code that defines the listener is saved as `listenerSource`—this is the listener that the function is supposed to prepare and wrap. A `listenerSource` might appear as, say, an inline HTML event listener:

```
<a href="#" onclick="alert('clicked!');">button</a>
```

This code, which defines a listener that waits for a button click to produce a pop-up saying "clicked!", is saved as `listenerSource`. In the second line of the bug snippet, `listenerSource` is concatenated into a string that forms a JavaScript function. After the next two lines, the string is compiled and run, and its return value is saved as the V8 value `result`. `result` should be a function, since `code` defines a function that returns another function. In fact, for the inline event listener above, `result`'s code is equivalent to: `function() { alert("clicked!"); }`.

To be safe, the authors `ASSERT` that `result` is truly a function (line 138). Then, they cast the `result` to a JavaScript function type, and run the casted `intermediateFunction` to (hopefully) receive the event listener that the function is supposed to prepare. Unfortunately, asserts are turned off in release builds, so `result` will "called" regardless of whether it is a JavaScript function on not. Since event listeners are simply strings that the above code wrapped in a function body, a well crafted string can terminate the wrapping function block and return an arbitrary JavaScript value. Unsurprisingly, this will cause `callInternalFunction` to crash the browser. This bug was fixed as a high security vulnerability in Chrome 41.0.2272.76.

Incorrect types like this one have caused a number of serious Chrome vulnerabilities over the past two years. Bad casts, for example, caused both CVE-2015-1230 and CVE-2014-3199, high and low severity vulnerabilities respectively [12, 16]. In CVE-

2015-1230, the function `findOrCreateWrapper` is supposed to wrap an event listener in order to expose it to JavaScript. The object to be wrapped is passed as a `v8::Value`. Before wrapping the object, though, the function calls `doFindWrapper` to see if the object is already wrapped. `doFindWrapper` tries to extract wrapper contents from the object; if successful, the object is already wrapped and can be casted as an event listener. `doFindWrapper` extracts wrapper contents, though, by doing a lookup using the name of the wrapped object—in the case of event listeners, "listener." Sadly, `EventListener`s and `AudioListener`s are both named "listener," and so wrapped `AudioListener`s get casted as `EventListener`s, causing similar crashes [13]. This bug took eleven days to track down and fix.

Finally, calling functions on the wrong receivers is a perennial problem in binding code. CVE-2009-2935 (high severity, fixed in Chrome 2.0.172.43) allowed for potentially sensitive, cross-origin information to be exposed to calling JavaScript code [8]. Internally, the V8 function `Invoke` invokes a function on a receiver. Unfortunately, if this function is called on the document object (e.g. `document.foo()`), V8 sets the receiver as the global object. The receiver is *not* supposed to be the global object; rather, functions on the global object should be called on the `globalReceiver`, which is stored at a safe offset in the global object. This offset excludes the global object's `globalContext`, which might contain the sensitive, cross-origin informatoin. Traditionally, developers could use type systems to prevent such of bugs, making the `globalReceiver` a receiver type object—but in C++ code, there is no type checking for JavaScript types.

This was not an isolated incident: CVE-2016-1612 also results from incompatible recievers, receivers of the wrong type [21]. Finally, the fix for CVE-2015-6775 adds V8 Signatures to PDFium code to prevent vulnerabilities related to incompatible receivers [20]; according to V8 docs, Signatures "specify which receivers are valid to a function" [20]. Signatures may help with the wrong-receiver-type bugs, since they enforce type signatures for JavaScript functions in C++, but they are also difficult to understand and expensive (they require prototype chasing) [52]. Moreover, they do not prevent bad cast bugs.

CVEs caused by control flow The vulnerabilities in this section all involve V8 code that either incorrectly handles JavaScript control flow or fails due to JavaScript code violating C++ control flow invariants. CVE-2015-6764 is a high severity vulnerability fixed in Chrome 47.0.2526.73 [17]. The buggy code, given bellow, is in the `SerializeJSArray` function [18]:

```
430 BasicJsonStringifier::Result
    ↪  BasicJsonStringifier::SerializeJSArray
    ↪  (Handle<JSArray> object) {
431  uint32_t length = 0;
432  CHECK(object->length()->ToArrayLength(&length));
433  ...
434  Handle<FixedArray>
     ↪  elements(FixedArray::cast(object->elements()),
     ↪  isolate_);
435  for (uint32_t i = 0; i < length; i++) {
436   Result result = SerializeElement(isolate_,
      ↪  Handle<Object>(elements->get(i), isolate_), i);
437  }
438 }
```

This function turns a JavaScript array (`object`) into its JSON string representation. First, it initializes variable `length` to the array's length in lines 431 and 432. Then, starting on line 434, it creates a `FixedArray` out of the elements in `object` and iterates through that array from zero to `length`, calling `elements->get(i)` with each iteration.

Unfortunately, JavaScript arrays are object and the `get(i)` on line 436 may thus call a user-defined getter function. Since the getter can execute arbitrary code, it can reduces the array's length before returning to C++. Unfortunately, the C++ **`for`** loop still continues to iterate based on the cached `length` value from line 432, causing an out-of-bounds memory access.

This array iteration bug is not an anomaly; reasoning about when JavaScript objects in C++ code can call *back* into users' JavaScript functions means reasoning about the entire call chain. CVE-2016-1646 is very similar to the first bug in this section [9]; user code may change the length of an array during iteration. CVE-2014-3188 may invoke getters and setters (containing potentially arbitrary JavaScript code) in a switch statement that switches on the type of the object. These getters or setters may alter the type of the object after entering the case for a certain type, causing type confusion vulnerabilities like those discussed in the previous section. Finally, in some cases, arbitrary code is called at program points where it should not be, causing same-origin bypass. CVE-2015-6769 and CVE-2016-1679 both fall into this latter category [19, 22].

These vulnerabilities highlight two challenges related to control flow in binding code. The first and most obvious is that code in one language must take changes from the other into account (e.g. a JavaScript getter altering the length of an array that the C++ code may have cached). The second challenge is more subtle. Since different languages have different "colloquial" approaches to common tasks like iterating through a loop, translating these tasks from one language to another can introduce errors. In the case of the array bug, a natural JavaScript iteration technique uses a `forEach` loop, while the C++ version uses a **`for`** loop. A `forEach` loop is resilient to changes in length; a **`for`** loop is not. Both of these challenges, reasoning about complicated call chains and translating operations from one language into another, introduce subtle errors that are hard to avoid without some sort of safety net.

3. Finding and eliminating binding code bugs

Finding bugs In Section 2, we profile CVEs to determine whether they follow common patterns. Many of them do: for example, one class of bug results from casting values without checking their types. This kind of pattern is a static checker's sweet spot; checkers search for buggy patterns in source code and produce warnings when they enoucter those patterns, preventing vulnerabilities in production code. Moreover, static checkers may complement Chrome's existing testing infrastructure by giving developers more information about crash reports (e.g. by CloudFuzzer of ASan [1]), allowing them to fix bugs in days instead of weeks.

The challenge of static checking for binding code is that checkers are **a)** often limited to one language and **b)** not quick and easy to prototype. To effectively identify the types of errors in this paper, checkers must understand C++, V8 and Blink C++ colloquiallisms (e.g. `As<>()`), and IDL backend template code. Furthermore, information about which JavaScript code touches which C++ code can enhance analyses and suppress false reports.

Binding code checkers are not quick and easy to prototype because checker creators do not know which properties to check. Similarly, most people who write binding code are not familiar enough with frontends to write their own system-specific checkers easily or accurately. As a result, there are no static checkers specifically designed for browser binding code.

We believe that the micro grammar approach of [4] addresses both of these concerns: it supports flexible, cross-language checkers with very little developer overhead. Checkers in this system are often under fifty lines long, and adapting a checker from one language to another can take five to fifty more lines—far fewer than a traditional system requires.

Eliminating bugs We propose designing more secure binding APIs that **(1)** prevent buggy, unintended behavior, **(2)** make cross-language information explicit, and **(3)** are lightweight enough to easily change. First, while low-level binding APIs may be necessarily unsafe, we can build safe programming abstractions, like EDSLs, on top of them. Blink and V8 have already transitioned to using some DSLs embedded in C++. For example, to address the memory leaks and use-after-free bug in Chrome, the Blink team recently replaced `RefPtr<>` pointers with `Member<>` pointers which are managed by Oilpan, a C++ garbage collector [28]. We imagine a more through redesign: an API could provide constructs for explicitly defining reflected objects and methods on them (e.g. atop V8 Signatures), abstracting away raw C++ pointers and making the relationship between such stored pointers and other code explicit.

Second, we think that binding APIs, while enforcing safety, should also make it easier to determine which language is doing what. An EDSL could provide safe C++ iterators over JavaScript arrays (and other objects) by making the runtime-dependency of loop invariants explicit. Another could provide safe constructs for managing JavaScript control flow in C++, for example, by using `Either<>` types [41] to make errors explicit and thus force C++ code to abide by JavaScript control flow. This will not only prevent low-level control flow confusions, say, but also allow programmers to reason more soundly about their systems—sidestepping higher-level logic bugs.

Finally, we believe that DSLs must have a flexible language design, since they should change to support ever-evolving browser components. Furthermore, pliant languages can adapt to developers needs; a successful DSL is one that people use.

4. Related Work

Finding bugs in multi-language systems In [37], Li and Tan present a static analysis tool that detects reference counting bugs in Python/C interface code. Their results show that static checkers can help secure cross-language code. Still, their technique is not directly applicable to browsers' binding code, since porting checkers from one language to another takes a surprising amount of overhead [3]. Furthermore, checkers for browser binding code should be flexible enough to understand C++, V8, template code, and JavaScript, which we discuss in Section 3.

Tan and Croft find type safety bugs that result when developers expose C pointers to Java as integers [48]. Their work illustrates that static checkers can find type safety bugs in practice, as we suggest in Section 3. More broadly, Furr and Foster [25, 26] present a multi-language type inference systems for the OCaml and Java FFIs. Since JavaScript is dynamically typed, using type-inference approaches like these is difficult—but a similar analysis could help ensure that binding layer dynamic type checks are correct. Different static analysis techniques for finding bugs caused by mishandled exceptions in Java JNI code were presented in [33, 36, 48]. Some mishandled exceptions are preventable with safer binding-layer APIs (e.g. the V8's `Maybe` API [24, 27]). Static analysis might help find other bugs, though, like unexpected, crashing exceptions in native code.

Jinn [35] generates dynamic bug checkers for arbitrary languages from state machine descriptions of FFI rules. In doing so, Jinn finds bugs in all three categories in both JNI and Python/C code. While running similar checkers in production is probably prohibitively expensive, such a system would complement existing browser debug-runtime checks.

Securing binding code by construction Several projects develop formal models for multi-language systems and FFIs [34, 38, 47, 50]. These works not only help developers understand multi-language interaction but also allow developers to formally reason about *tools* for multi-language systems (like static checkers and binding layer EDSLs). We envision developing a similar formalization for JavaScript, perhaps based on S5 [42].

Janet [5] and Jeannie [30] are language designs that allow users to combine Java and C code in a single file, therefore building multi-language systems more safely. SafeJNI [49] provides a safe Java/C interface by using CCured [40] to retrofit C code to a subset that abides by Java's memory and type safety. While these language designs are well-suited for isolated browser features, it is unclear how these approaches would scale in a full browser engine. For browsers, we believe that DSLs scale better while providing similar safety guarantees. Of course, new DSLs require refactoring FFI code—but techniques like cross-language IDE plugins could help automate this [46].

Running different languages' runtimes in isolated environments addresses many security bugs in FFI code. Klinkoff *et al.* [32] presents an isolation approach for the .NET framework: they run native, unmanaged code in a separate sandboxed process mediated according to the high-level .NET security policy. Robusta [45] takes a similar approach for Java, but, to improve performance, they use software fault isolation (SFI) [53] for isolation and mediation within a single address space. These techniques are complimentary to our agenda.

References

[1] AddressSanitizer. Chromium. https://www.chromium.org/developers/testing/addresssanitizer.

[2] A. Barth. The web origin concept. Technical report, IETF, 2011. URL https://tools.ietf.org/html/rfc6454.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2), 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646374. URL http://doi.acm.org/10.1145/1646353.1646374.

[4] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–157. ACM, 2016.

[5] M. Bubak, D. Kurzyniec, and P. Luszczek. Creating java to native code interfaces with janet extension. In *Proceedings of the First Worldwide SGI Users Conference*, pages 283–294, 2000.

[6] Chromium. Stable channel update for chrome os: Friday, march 14, 2014. http://googlechromereleases.blogspot.com/2014/03/stable-channel-update-for-chrome-os_14.html, .

[7] Chromium. Idl compiler. https://www.chromium.org/developers/design-documents/idl-compiler, .

[8] Chromium. Issue 18639 and cve-2009-2935. https://bugs.chromium.org/p/chromium/issues/detail?id=18639, .

[9] Chromium. Cve2014-1646. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1646, .

[10] Chromium. Issue 352374. https://bugs.chromium.org/p/chromium/issues/detail?id=352374, .

[11] Chromium. Side by side diff for issue 196343011. https://codereview.chromium.org/196343011/diff/20001/Source/bindings/templates/attributes.cpp, .

[12] Chromium. Issue 395411 and cve-2014-3199. https://bugs.chromium.org/p/chromium/issues/detail?id=395411, .

[13] Chromium. Side by side diff for issue 424813007. https://codereview.chromium.org/424813007/diff/40001/Source/bindings/core/v8/custom/V8EventCustom.cpp, .

[14] Chromium. Issue 456192 and cve-2015-1217. https://bugs.chromium.org/p/chromium/issues/detail?id=456192, .

[15] Chromium. Side by side diff for issue 906193002. https://codereview.chromium.org/906193002/diff/20001/Source/bindings/core/v8/V8LazyEventListener.cpp, .

[16] Chromium. Issue 449610 and cve-2015-1230. https://bugs.chromium.org/p/chromium/issues/detail?id=449610, .

[17] Chromium. Issue 554946 and cve-2015-6764. https://bugs.chromium.org/p/chromium/issues/detail?id=554946, .

[18] Chromium. Side by side diff for issue 1440223002. https://codereview.chromium.org/1440223002/diff/1/src/json-stringifier.h, .

[19] Chromium. Issue 534923 and cve-2015-6769. https://bugs.chromium.org/p/chromium/issues/detail?id=534923, .

[20] Chromium. Issue 529012 and cve-2015-6775. https://bugs.chromium.org/p/chromium/issues/detail?id=529012, .

[21] Chromium. Issue 497632 and cve-2016-1612. https://bugs.chromium.org/p/chromium/issues/detail?id=497632, .

[22] Chromium. Issues 606390 and cve-2016-1679. https://codereview.chromium.org/1930953002, .

[23] C. Details. Google Chrome: CVE security vulnerabilities, versions and detailed reports. https://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224.

[24] B. English. ¡=v4: process.hrtime() segfaults on arrays with error-throwing accessors. https://github.com/nodejs/node/issues/7902.

[25] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–72. ACM, 2005.

[26] M. Furr and J. S. Foster. Polymorphic type inference for the jni. In *European Symposium on Programming*, pages 309–324. Springer, 2006.

[27] M. Hablich. API changes upcoming to make writing exception safe code more easy. https://groups.google.com/forum/#!topic/v8-users/gQVpp1HmbqM.

[28] K. Hara. Oilpan: Gc for blink. https://docs.google.com/presentation/d/1YtfurcyKFS0hxPOnC3U6JJroM8aRP49Yf0QWznZ9jrk, .

[29] K. Hara. A generational GC for DOM nodes. https://docs.google.com/presentation/d/1uifwVYGNYTZDoGLyCb7sXa7g49mWNMW2gaWvMN5NLk8, .

[30] M. Hirzel and R. Grimm. Jeannie: Granting java native interface developers their wishes. In *ACM Sigplan Notices*, volume 42, pages 19–38. ACM, 2007.

[31] B. Holley. Typed arrays supported in xpconnect. https://bholley.wordpress.com/2011/12/13/typed-arrays-supported-in-xpconnect/.

[32] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna. Extending. net security to unmanaged code. *International Journal of Information Security*, 6(6):417–428, 2007.

[33] G. Kondoh and T. Onodera. Finding bugs in java native interface programs. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 109–118. ACM, 2008.

[34] A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *Software Engineering and Formal Methods*, pages 215–230. Springer, 2015.

[35] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *ACM Sigplan Notices*, volume 45, pages 36–49. ACM, 2010.

[36] S. Li and G. Tan. Finding bugs in exceptional situations of jni programs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 442–452. ACM, 2009.

[37] S. Li and G. Tan. Finding reference-counting errors in python/c programs with affine analysis. In *European Conference on Object-Oriented Programming*, pages 80–104. Springer, 2014.

[38] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(3):12, 2009.

[39] C. McCormack. Web idl. *World Wide Web Consortium*, 2012.

[40] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[41] B. O'Sullivan, J. Goerzen, and D. B. Stewart. *Real world haskell: Code you can believe in*. " O'Reilly Media, Inc.", 2008.

[42] J. G. Politz, M. J. Carroll, B. S. Lerner, and S. Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *DLS*, 2012.

[43] A. Ranganathan, J. Sicking, and M. Kruisselbrink. File API. *World Wide Web Consortium*, 2015.

[44] G. A. Security. Chrome rewards. `https://www.google.com/about/appsecurity/chrome-rewards/index.html`.

[45] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the jvm. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 201–211. ACM, 2010.

[46] N. Sultana, J. Middleton, J. Overbey, and M. Hafiz. Understanding and fixing multiple language interoperability issues: the c/fortran case. In *Proceedings of the 38th International Conference on Software Engineering*, pages 772–783. ACM, 2016.

[47] G. Tan. Jni light: An operational model for the core jni. In *Asian Symposium on Programming Languages and Systems*, pages 114–130. Springer, 2010.

[48] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Usenix Security Symposium*, pages 365–378, 2008.

[49] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe java native interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, volume 97, page 106, 2006.

[50] V. Trifonov and Z. Shao. Safe and principled language interoperation. In *European Symposium on Programming*, pages 128–146. Springer, 1999.

[51] L. Tung. Android bugs made up 10 percent of google's $2m bounty payouts - in just five months. http://www.zdnet.com/article/android-bugs-made-up-10-percent-of-googles-2m-bounty-payouts-in-just-five-months/, January 2016.

[52] v8-users maling list. What is the difference between arguments::holder() and arguments::this()? `https://groups.google.com/forum/#!topic/v8-users/Axf4hF_RfZo`.

[53] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.