Short Paper: Superhacks

Exploring and Preventing Vulnerabilities in Browser Binding Code

Fraser Brown

Stanford University
mlfbrown@stanford.edu

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 may 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 [50, 58]. But, as both dangerous security breaches and more benign hacking contests such as Pwn2Own demonstrate, browsers are still rife with exploitable bugs.

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 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 the low-level control that JavaScript lacks.

PLAS '16, October 24, 2016, Vienna, Austria.
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4574-3/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2993600.2993602

Browser binding code is different from—and more complicated than—other multi-language systems such as foreign function interfaces (FFIs). This difference arises for three main reasons. First, browser binding code must reconcile different runtimes instead of making simple library calls. Second, since the binding layer sits between JavaScript application code and the rest of the browser, browsers often rely on binding-layer code to implement the same-origin policy [2]. For example, binding code must enforce isolation between the JavaScript running in a page and any cross-origin iframes the page may embed. Finally, the binding layer must defend against potentially malicious code, since the JavaScript running in browsers comes from many disparate parties.

A buggy binding layer introduces browser vulnerabilities. Even with existing sandboxing mechanisms, binding layer bugs might allow a third-party ad to: crash the user's tab; read sensitive data like cookies, local storage, or HTTP headers; and perform authenticated requests to other sites on the user's behalf. A binding-layer vulnerability could even allow an attacker to inject arbitrary code into a cross-origin page. Binding code must be extremely defensive to prevent these sorts of vulnerabilities. For example, it 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].

Preventing exploitable bugs in binding code is especially difficult because of the impedance mismatch between C++ and JavaScript language abstractions; C++ and JavaScript's memory models, control flow constructs, and types are different enough that incorrectly managing their interactions 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 which can ultimately lead to crashes. 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 that may lead to memory disclosures.

Unfortunately, the existing binding layer APIs actively work against developers tackling security challenges; Figure 1, taken from a core Chrome developer's presentation [30], refers to binding code as "super hacks." Rethinking and understanding the binding layer is crucial for users' security; it is also timely, as companies experiment with new security-focused layout engines like Servo, as more developers depend on related platforms like Node.js, and as engineers explore the Internet of Things by exposing hardware to JavaScript applications.

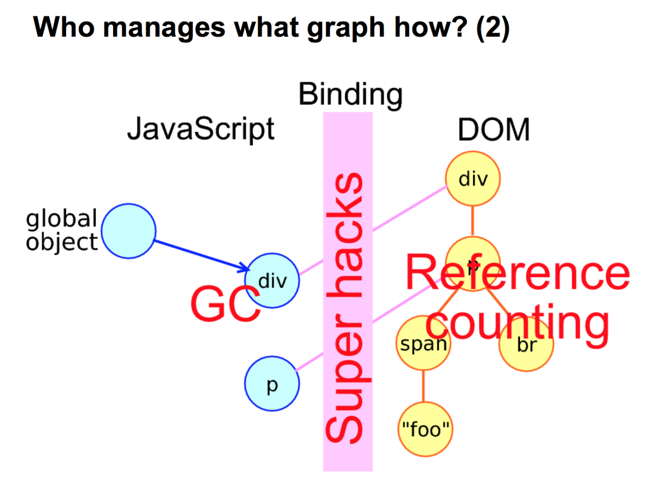

Figure 1: Slide from [30] illustrating the state of memory management code between JavaScript and C++ in Blink.

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 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 Common Vulnerabilities and Exposures (CVEs) to discover buggy patterns that checkers can recognize and that EDSLs can disrupt [25].[1] This section categorizes and distills the binding-specific buggy patterns that we found. We ignore traditional bugs, like C++ buffer overflows, and focus instead on binding bugs for two reasons. First, traditional checkers can detect traditional C++ bugs—the problem is checker adoption. Binding layer bugs, on the other hand, are checker-less; there are no checkers for anyone to adopt. Second, since new browsers components, like Servo, forgo C++ altogether, C++-specific checkers and language designs are useless to them; any tools for these browsers must depend on less language specific insights.

2.1 How to incorrectly expose an object to JavaScript

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 exposing FileAPI-like `Blob` objects [49] to JavaScript. `Blobs` should efficiently represent the binary data that we use to communicate with a server, so we want to implement them in better-performing C++. The WebIDL [44] interface describing `Blob` is:

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

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 `Blob` subsets (`blob.slice(2)`).

Whenever someone calls the JavaScript `Blob` constructor, the V8 JavaScript engine creates a reflector object, a JavaScript wrapper for the C++ object. Think of a reflector as a JavaScript facade; each time someone knocks on the door—makes a JavaScript call—they produce actions in the building itself—the underlying C++ object. When calling the `Blob` constructor from JavaScript, V8 creates the reflector and calls the underlying C++ binding constructor code on 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 that object 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 difficult to explain in prose and even more difficult to execute. In the next three paragraphs, we outline how memory, type, and control flow related bugs arise in `Blob` binding code.

Avoiding memory leaks means destroying C++ `Blob` objects each time their reflected JavaScript objects are collected. We do this, in binding code, 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) introduces a memory leak.

To prevent out-of-bounds vulnerabilities we must ensure that `Blob`'s `slice` function is called with valid bounds; though this process seems simple at first, we must cast correctly from JavaScript `numbers` 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).

[1] Although we focus on the binding layer in Chrome's Blink rendering engine in this paper, we do not mean to single them out unfairly; binding layers in other browsers are at least as complex and struggle with similar issues. For instance, see [33] for comments on the complexity of Mozilla's XPConnect.

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 that of C++. Otherwise we might execute unintentional, potentially dangerous code [40].

These memory, type, and control flow issues (even for simple objects like `Blob`) are far from exhaustive; in the next sections, we present real-world CVEs that exploit more complicated bugs.

2.2 CVEs in the wild

To estimate the impact of binding bugs on Chrome's security, we surveyed the most recently publicly available bountied CVE reports—March, April, and May of 2016. During this time, binding code was responsible for ten high severity security vulnerabilities out of thirty-two total high severity security vulnerabilities.[2] This number may be low. A few reports are still closed, since Google's fourteen-week view-restriction on issue reports is a minimum, not a guarantee. Furthermore, some reports are so complicated—they touch so many parts of the Chrome codebase—that it is difficult for non-Chrome engineers to determine their origin. Therefore, we counted binding code vulnerabilities conservatively. Even so, binding code-related CVEs clearly make an impact on Chrome's security. We illustrate the contents of some especially interesting CVEs from 2009 to Spring 2016 in the next sections.[3]

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 Document Objects Model (DOM) objects in order to render them on screen. 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 [8], automating some of work we described when explaining `Blobs`.

Despite using generated code, 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 [31]. Not all of 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.

[2] By reporting period: 4/9, 1/3, 2/4, 1/2, 1/3, 0/3, 1/8

[3] Information from googlechromereleases.blogspot.com

Oilpan did not land until December 2015; in the meantime, CVE-2014-1713, a use-after-free vulnerability in Blink's binding code, contributed to a full execution-outside-sandbox exploit with a 100,000 dollar bounty [7, 10]. The source of this bug was Blink's C++ template code for attribute setters [11]:

```
    /* blink/Source/bindings/templates/attributes.cpp */

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);
```

The WebIDL compiler backend fills in the template variables like `{{cpp_class}}`, creating a normal C++ file. After code generation, line 234 creates `proxyImp`, an object of the type specified by the WebIDL file. In this CVE, `proxyImp` is a `Document` object. On line 235, the `Document`'s location attribute is assigned to the raw pointer `imp`. Unfortunately, the code on line 247 may perform an up-call (e.g. a call to a JavaScript-defined `toString` function). Malicious JavaScript can remove any references to the reflected attribute object during this call and force V8 to GC. Since Blink registers GC callbacks to clean up the C++ objects backing reflected objects, this will result in `imp` being freed—to the callback code, it appears as if there are no live references to the object. Thereafter (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 `MutationObservers` 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 `MutationObservers`. 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 `RefPtrs` instead of `RawPtrs`—`RefPtrs`, 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] `RefPtrs` 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 addresses many issues, it will be interesting to see what bugs the transition may introduce.

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. JavaScript values at the binding layer are exposed as `v8::Values`. Later code must cast these `Values` to other types (like `v8::Function`); incorrect casts, like casts to `Functions` when `Values` are not functions, cause crashes. The JavaScript to C++ translation process can also yield more serious and unexpected errors. The code for CVE-2015-1217 [14, 15] highlights one such case:

```
   /* blink/Source/bindings/core/
      v8/V8LazyEventListener.cpp */

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 inline 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 an event listener of an anchor element:

```
<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`. Note that `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 be "called" regardless of whether it is a JavaScript function on not. Since inline 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.

Type mishandling like this led to a number of 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, 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 cast to 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, `EventListeners` and `AudioListeners` are both named "listener," and so wrapped `AudioListeners` get cast to `EventListeners`, 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 potentially sensitive, cross-origin information to be exposed to calling JavaScript code [9]. In the CVE, 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 to the global object. The receiver is *not* supposed to be the global object; rather, functions on the global object should be called on the global proxy object (formerly `globalReceiver`), which safely proxies set/gets to the actual global object [46]. Normally, developers could use type systems to prevent such bugs, making the `globalReceiver` a special `receiver` type object—but in C++ code, there is no enforcement for JavaScript types.

This CVE was not an isolated incident: CVE-2016-1612 also results from receivers of incompatible types [21]. Finally, the fix for CVE-2015-6775 adds V8 Signatures to PDFium code to prevent vulnerabilities related to incompatible receivers [20]. Signatures "specify which receivers are valid to a function" [20]. They reduce wrong-receiver-type bugs, since they enforce type signatures for JavaScript functions in C++, but they are both expensive (they require prototype chasing) [59] and difficult to understand. Moreover, they do not prevent bugs due to bad casts.

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 below, is in the `SerializeJSArray` function [18]:

```
/* v8/src/json-stringifier.h */

BasicJsonStringifier::Result
  ↪  BasicJsonStringifier::SerializeJSArray
  ↪  (Handle<JSArray> object) {
 uint32_t length = 0;
 CHECK(object->length()->ToArrayLength(&length));
 ...
 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.

JavaScript arrays are objects and therefore the `get(i)` on line 436 may 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 (OOB).

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; user code may change the length of an array during iteration [22]. CVE-2014-3188 may invoke getters and setters (containing potentially arbitrary JavaScript code) after caching the object type, which is used in a later switch statement [22]. Since the getters and setters may alter the type of the object before entering the switch block, this can, for example, lead the C++ code corrupting or disclosing memory. 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, 23].

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 bugs, 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

To better secure browsers, we propose both statically checking binding code and devising new programming models that make it easier to write safe binding code.

3.1 Finding bugs

In Section 2, we profiled 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 encounter those patterns. Moreover, static checkers may complement Chrome's existing testing infrastructure by giving developers more information about crash reports (e.g. by CloudFuzzer or ASan [1]), allowing them to fix bugs in days instead of weeks.

The challenge of static checking for binding code is that checkers are **(1)** often limited to one language and **(b)** not quick and easy to prototype. To effectively identify the types of errors considered in this paper, checkers must understand C++, V8 and Blink C++ colloquialisms, and IDL 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 static checking systems to write their own system-specific checkers easily or accurately. As a result, to the best of our knowledge, 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 is possible in five to fifty more lines—far fewer than a traditional system requires.

3.2 Eliminating bugs

New programming models can eliminate classes of bugs by construction. We propose designing a collection of secure binding APIs that **(1)** prevent buggy, unintended behavior, **(2)** make cross-language information explicit, and **(3)** are flexible enough to adjust. Finally, we hope to design EDSLs with static checking in mind, so that they are easier to check than the original C++ code that they replace. This way, checkers can shoulder some of the EDSLs' burdens: they can provide warnings against buggy patterns that are too hard to prevent by construction.

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 EDSLs 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 the Oilpan C++ garbage collector [31]. Similarly, they introduced `Maybe` types to make it easier to write safe code in the presence of JavaScript exceptions [26, 29]. 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. One 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 [47] to make errors explicit and thus force C++ code to abide by JavaScript control flow. This not only prevents low-level control flow confusions, for example, but also allows programmers to reason more soundly about their systems, sidestepping higher-level logic bugs.

Finally, we believe that EDSLs 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 EDSL is one that people use.

4. Related Work

4.1 Finding bugs in multi-language systems

Memory management bugs Li and Tan present a static analysis tool that detects reference counting bugs in Python/C interface code [41]. 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 3.1.

Type safety bugs Tan and Croft find type safety bugs that result when developers expose C pointers to Java as integers [53]. Their work illustrates that static checkers can find type safety bugs in practice, as we suggest in Section 3.1. Exposed pointer errors, however, are unlikely in Chrome binding code, since V8 provides hidden fields for C code to store raw pointers cross context. Still, since these errors would be catastrophic, it may be worthwhile to implement Tan and Croft's technique for browser binding code.

More broadly, Furr and Foster [27, 28] 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. Moreover, they would be applicable to other JavaScript runtimes (e.g. Node.js) where type checks are not generated and are, instead, implemented manually.

Control flow bugs Kondoh, Tan, and Li present different static analysis techniques for finding bugs caused by mishandled exceptions in Java JNI code [36, 40, 53]. Safer binding-layer APIs would address some of the issues these works tackle. For example, the recent V8 API addresses similar memory management and exception-catching concerns by construction [26, 29]. Still, we believe that static exception analysis might find bugs in binding code where exceptions are raised in native code.

More generally, it is possible to translate multi-language programs into an intermediate language and apply off-the-shelf analysis tools to their translation [6, 38, 42, 54]. For large codebases like Chrome, this approach is as feasible as the analysis approach is scalable.

Jinn [39] generates dynamic bug checkers for arbitrary languages from state machine descriptions of FFI rules. In doing so, Jinn finds bugs in both the JNI and Python/C code. While running similar checkers in production is probably prohibitively expensive, this kind of system would complement existing browser debug-runtime checks.

4.2 Writing secure binding code by construction

Formal models Several projects develop formal models for multi-language systems and FFIs [37, 43, 52, 57]. 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 [48].

New language designs Janet [5] and Jeannie [32] are language designs that allow users to combine Java and C code in a single file, therefore building multi-language systems more safely. SafeJNI [55] provides a safe Java/C interface by using CCured [45] 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 EDSLs scale better (because they do not require as many dynamic checks), while providing similar safety guarantees. But, of course, new EDSLs require refactoring existing FFI code.

4.3 Mitigating the effects of binding bugs

Fault isolation Running different languages' runtimes in isolated environments addresses many security bugs in FFI code. Klinkoff *et al.* [35] present 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 [51] takes a similar approach for Java, but, to improve performance, uses software fault isolation (SFI) [61]. These techniques are complimentary to our agenda.

Browser redesigns Redesigning the browser to run iframes in separate processes would address many of the vulnerabilities that lead to SOP bypasses. Unfortunately, as illustrated by Chrome's ongoing efforts [24] and several research browsers (e.g., Gazelle [60], IBOS [56], and Quark [34]) this is not an easy task. Re-designs often break compatibility and have huge performance costs. Still, we believe that re-design efforts are complimentary to our agenda.

Acknowledgements

Deian Stefan should be an author on this paper, but the SIGPLAN rules disallow chairs from submitting papers to conferences they are chairing. We thank the PLAS reviewers for their helpful comments. Thanks to Dawson Engler, Andres Nötzli, and Diana Young. This research was supported in part by Intel award 1505728 and NSF award 120750.

References

[1] AddressSanitizer. Chromium. https://www.chromium.org/developers/testing/addresssanitizer, 2016.

[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.

[4] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *ASPLOS*. ACM, 2016.

[5] M. Bubak, D. Kurzyniec, and P. Luszczek. Creating Java to native code interfaces with Janet extension. In *Worldwide SGI Users Conference*, 2000.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[7] 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, 2016.

[8] Chromium. IDL compiler. https://www.chromium.org/developers/design-documents/idl-compiler, 2016.

[9] Chromium. Issue 18639 and CVE-2009-2935. https://bugs.chromium.org/p/chromium/issues/detail?id=18639, 2016.

[10] Chromium. Issue 352374. https://bugs.chromium.org/p/chromium/issues/detail?id=352374, 2016.

[11] Chromium. Side by side diff for issue 196343011. https://codereview.chromium.org/196343011/diff/20001/Source/bindings/templates/attributes.cpp, 2016.

[12] Chromium. Issue 395411 and CVE-2014-3199. https://bugs.chromium.org/p/chromium/issues/detail?id=395411, 2016.

[13] Chromium. Side by side diff for issue 424813007. https://codereview.chromium.org/424813007/diff/40001/Source/bindings/core/v8/custom/V8EventCustom.cpp, 2016.

[14] Chromium. Issue 456192 and CVE-2015-1217. https://bugs.chromium.org/p/chromium/issues/detail?id=456192, 2016.

[15] Chromium. Side by side diff for issue 906193002. https://codereview.chromium.org/906193002/diff/20001/Source/bindings/core/v8/V8LazyEventListener.cpp, 2016.

[16] Chromium. Issue 449610 and CVE-2015-1230. https://bugs.chromium.org/p/chromium/issues/detail?id=449610, 2016.

[17] Chromium. Issue 554946 and CVE-2015-6764. https://bugs.chromium.org/p/chromium/issues/detail?id=554946, 2016.

[18] Chromium. Side by side diff for issue 1440223002. https://codereview.chromium.org/1440223002/diff/1/src/json-stringifier.h, 2016.

[19] Chromium. Issue 534923 and CVE-2015-6769. https://bugs.chromium.org/p/chromium/issues/detail?id=534923, 2016.

[20] Chromium. Issue 529012 and CVE-2015-6775. https://bugs.chromium.org/p/chromium/issues/detail?id=529012, 2016.

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

[22] Chromium. Issue 594574 and CVE-2016-1646. https://bugs.chromium.org/p/chromium/issues/detail?id=594574, 2016.

[23] Chromium. Issues 606390 and CVE-2016-1679. https://bugs.chromium.org/p/chromium/issues/detail?id=606390, 2016.

[24] Chromium. Out-of-process iframes. http://www.chromium.org/developers/design-documents/oop-iframes, 2016.

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

[26] B. English. $\leq$v4: process.hrtime() segfaults on arrays with error-throwing accessors. https://github.com/nodejs/node/issues/7902.

[27] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *PLDI*. ACM, 2005.

[28] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *ESOP*. Springer, 2006.

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

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

[31] K. Hara. Oilpan: GC for Blink. https://docs.google.com/presentation/d/1YtfurcyKFS0hxPOnC3U6JJroM8aRP49Yf0QWznZ9jrk, 2016.

[32] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *ACM SIGPLAN Notices*, volume 42. ACM, 2007.

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

[34] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, 2012.

[35] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna. Extending .NET security to unmanaged code. *Journal of Information Security*, 6(6), 2007.

[36] G. Kondoh and T. Onodera. Finding bugs in Java native interface programs. In *Symposium on Software Testing and Analysis*. ACM, 2008.

[37] A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *SEFM*. Springer, 2015.

[38] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. IEEE, 2004.

[39] 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. ACM, 2016.

[40] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *CCS*. ACM, 2009.

[41] S. Li and G. Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *ECOOP*. Springer, 2014.

[42] P. Linos, W. Lucas, S. Myers, and E. Maier. A metrics tool for multi-language software. In *SEA*, 2007.

[43] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3), 2009.

[44] C. McCormack. Web IDL. *World Wide Web Consortium*, 2012.

[45] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37. ACM, 2002.

[46] M. D. Network. Split object. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Split_object.

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

[48] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. *ACM SIGPLAN Notices*, 48(2): 1–16, 2013.

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

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

[51] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the JVM. In *CCS*. ACM, 2010.

[52] G. Tan. Jni light: An operational model for the core JNI. In *ASPLAS*. Springer, 2010.

[53] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *Usenix Security*, 2008.

[54] G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM SIGPLAN Notices*, volume 42. ACM, 2007.

[55] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java native interface. In *Secure Software Engineering*, volume 97, 2006.

[56] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *OSDI*, 2010.

[57] V. Trifonov and Z. Shao. Safe and principled language interoperation. In *ESOP*. Springer, 1999.

[58] 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.

[59] v8-users maling list. What is the difference between Arguments::Holder() and Arguments::This()? https://groups.google.com/forum/#!topic/v8-users/Axf4hF_RfZo.

[60] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle Web Browser. In *USENIX security*, 2009.

[61] 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 *Security and Privacy*. IEEE, 2009.