Web API Verification: Results and Challenges

Arjun Guha
Brown University
Providence, RI
arjun@cs.brown.edu

Benjamin Lerner
Brown University
Providence, RI
blerner@cs.brown.edu

Joe Gibbs Politz
Brown University
Providence, RI
joe@cs.brown.edu

Shriram Krishnamurthi
Brown University
Providence, RI
sk@cs.brown.edu

Modern Web applications use several APIs to build rich features. For example, the W3C Geolocation API provides access to a user's physical location for location-based personalization, Facebook Connect provides a single sign-on service and access to a user's social graph, Local Storage is a per-domain cache that enables offline access, and the Google Maps API allows applications to embed customized maps. Some of these APIs, such as Geolocation and Local Storage, are built into Web browsers, whereas others, such as Facebook Connect and Google Maps, are provided by Web services. These APIs are carefully crafted, since they give applications access to sensitive information. However, security bugs are constantly found that affect API providers, Web applications, and end users.

Some APIs are security-specific. For instance, ADsafe [2] and Caja [6] are systems designed to securely sandbox content combined from non-mutually-trusting authors on a single Web page. Through a combination of static and runtime checks, these systems aim to provide certain security guarantees to the content and/or the hosting page. In such cases, where the authors may be actively malicious, verifying the APIs is critical.

Other APIs were not designed with security in mind, but security concerns arose as they became widely implemented and programmers understood their full potential. Consider the XMLHttpRequest API, which allows JavaScript programs to send HTTP requests to Web servers. When first released, a malicious JavaScript program using XMLHttpRequest could trivially set special HTTP headers that caused Web servers to misinterpret requests. Web browsers now blacklist several headers. Mozilla Firefox has an Extension API, which third-party extensions use with great effect. However, third-party extensions run with the same privileges as Firefox's code. Mozilla thus has an arduous, error-prone review process to vet extensions.

In principle, these APIs are *reference monitors*, which should be small and verifiable. In practice, these APIs have several entry points with complex static and runtime checks, resulting in a large attack surface that a large attack surface, complex runtime checks, and several entry points that could be compromised. For example, the Firefox extension API has over $1,000$ types, some of which allow access to sensitive resources such as local files. ADsafe has only $1,800$ lines of code, but 95 runtime security checks. A missing or misused security check could compromise the safety of the entire system. Moreover, these APIs do not crisply state their security goals, which makes verification difficult. Designing and implementing these APIs securely, and verifying that this has been done, is hence an important challenge.

Summary of Prior and Current Work

Client-side Web APIs are invariably provided for JavaScript, the lingua franca of the browser. Verifying these reference monitors therefore requires being able to encode assumptions about the environment and to use these to check properties of JavaScript code. This need has been noted in the form of identifying a shortcoming: the external security review [1] of Caja stated,

"[Caja is] hard to review. No map states invariants and points to where they are enforced, which hurts maintainability and security."

Naturally, static type systems address these needs very nicely. In particular, we have been working on adapting our existing JavaScript type checker [3, 4] to serve as a verification engine. This has especially required relaxing the notion of progress provided by the type system, since (in this context) termination through error is generally harmless—a halted component cannot further leak information or attack another component.

In previous work we verified ADsafe, a reference monitor that makes it safe to embed untrusted, third-party content on Web pages [7]. Our current work focuses on verifying that Firefox extensions do not misuse security-sensitive APIs in private browsing mode.

The completed and ongoing projects use a JavaScript type-checker as a verification tool. Type-checking JavaScript presents two challenges. First, JavaScript is a dynamic, untyped scripting language. Programmers freely use reflection, control-flow, and state to reason informally about types [4]. Although JavaScript only supports prototype-based objects, programmers use them to encode modules, classes, and other high-level features [3]. Second, different security APIs admit different classes of errors. For example, the ADsafe widgets should never use local storage, but Firefox extensions can use local storage when they are not in private browsing mode. Thus, a JavaScript type-checker that wants to support such a diversity of uses must be flexible enough to admit varied programming patterns and different error conditions.

We present an overview of the security architecture of ADsafe and Firefox Extensions, two Web APIs that we've studied in detail. We show how types can succinctly express the security invariants of these APIs. We highlight the novel features of our JavaScript type-checker that make type-based verification possible.

Challenges

Validating JavaScript Sub-Languages Unfettered JavaScript is not amenable to rich reasoning. Therefore, various industrial groups have developed other techniques, such as *linting* [2], *cajoling* [6], and *SES* [5] to restrict JavaScript. Cajoling and linting are complex techniques that do not come with any proofs or specifications, but are instead defined by their implementations. Since these processes effectively place syntactic restrictions on programs, we advocate expressing them as type systems. In prior work [7] we showed how to do this for ADsafe's *linting*. It remains to be seen whether *cajoling*, *SES*, and other techniques can be tackled in the same way.

From Programs to Pages to Sites Authors of Web APIs wish to reason about sophisticated safety properties that are beyond the scope of our type-checker and other tools for Web programs. For example, Web sandboxes such as ADsafe and Google Caja try to ensure that untrusted code can only read and draw to a single region (an HTML `<div>`) of a Web page. To verify that an implementation meets this specification, we must reason not only about JavaScript code, but also HTML markup, CSS stylesheets and even server-side content. Verifying realistic programs that contain both code and markup, and doing so in the context of server-side behavior, is beyond the scope of most existing tools.

Shrinking Attack Surfaces Web APIs support a plethora of features, have a large number of entrypoints, and thus have very large attack surfaces. For example, the Web browser DOM API has over 100 interfaces, while the Firefox Extension API has over $1,000$ interfaces. Security

vulnerabilities could lurk in the design or implementation of any of these interfaces. Shrinking the size of these APIs would mitigate the threat of attacks; the challenge is to do so without losing essential features. Another challenge is to make security and privacy behavior better manifest. For instance, Firefox expects extensions to check a global flag to determine whether they are in private browsing mode, and alter their behavior accordingly. A better design might be to use a capability-based approach [6] that provides the right capabilities in the first place, instead of expecting API users to make checks they are likely to (and, in practice, do) forget.

Establishing Non-Interference Web API designers are often interested in properties akin to non-interference of information flows. For example, ADsafe's author is interested in "restrictions required to prevent unintended collusion" between multiple programs. We found that a pair of untrusted, embedded widgets can communicate by simply reading and writing to shared state. The Web browser environment has several shared, mutable data structures, and restricting access to them would require a complete redesign of ADsafe. Such design problems would be caught much earlier were Web APIs built with security in mind.

Verifying Mixed-Language Implementations Some Web APIs, such as ADsafe, are fully implemented in JavaScript, whereas others, such as the Firefox extension API, are built with a mix of C++ and JavaScript code. Thus, understanding their behavior requires cross-language reasoning. Often, this is latent in the model of the environment: for instance, in verifying ADsafe we had to assume we knew all the APIs that could lead to `eval`-like behavior. We encoded our understanding by mining the knowledge of experts, but the ground truth is contained in the C++ code of the browser.

References

[1] Ihab Awad, Tyler Close, Adrienne Felt, Collin Jackson, Ben Laurie, Felix Lee, Ka-Ping Lee, David-Sarah Hopwood, Jasvir Nagra, Eric Sachs, Mike Samuel, Mike Stay, and David Wagner. Caja external security review. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/Caja_External_Security_Review_v2.pdf`.

[2] Douglas Crockford. ADSafe. `www.adsafe.org`, 2011.

[3] Arjun Guha, Joe Gibbs Politz, and Shriram Krishnamurthi. Fluid object types. Technical Report CS-11-04, Brown University, 10 2011. Under review.

[4] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming*, 2011.

[5] Mark M. Miller. Secure EcmaScript. `http://code.google.com/p/es-lab/wiki/SecureEcmaScript`, 2011.

[6] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[7] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.