A Security Architecture for Server-Side JavaScript: Extended Abstract

Willem De Groef Fabio Massacci Frank Piessens

July 8, 2014

Abstract

Node.js is a popular JavaScript server-side framework with an efficient runtime for cloud-based event-driven architectures. Its strength is the presence of thousands of third party libraries which allow developers to quickly build and deploy applications. These very libraries are a source of security threats as a vulnerability in one library can (and in some cases did) compromise one's entire server.

In order to support the least-privilege integration of libraries we develop NodeSentry, the first security architecture for server-side JavaScript. Our policy enforcement infrastructure supports an easy deployment of web-hardening techniques and access control policies on interactions between libraries and their environment, including any dependent library.

Introduction

JavaScript has many advantages for web development [9], as it is the de facto dominant language for client-side applications and it offers the flexibility of dynamic languages. Recently it also achieved an undeniable solid foothold at the server-side, illustrated by the rapidly growing community of developers and support from industry. Among the various event-driven programming languages, Node.js is a widely successful platform that combines the JavaScript language with an efficient runtime, tailored for a cloud-based event-driven architecture [17].

In particular, JavaScript supports the easy (and often necessary) combination or mash-up of content and libraries from disparate third parties. Such flexibility comes at a price of significant security and integrity problems [16], and researchers have proposed a number of solutions to address these problems, either using language-based sandboxing [18, 1], information flow control [5, 6, 3], or web browser modifications that enforce user- or server-provided access control policies [14, 19, 22]. Bielova [4] provides an extensive survey.

However, these proposals for client-side JavaScript cannot be lifted to server-side applications. Some client-side measures are embedded in a web browser, or they command a significant overhead acceptable at client-side but not at server-side. For example, Meyerovich's et al., [14] report some of the best micro-benchmarks for client-side JavaScript and still report an overhead between 24% to 300% of the raw time.

At the server-side, security problems are magnified: applications run without sandboxing and serve a large number of clients simultaneously. Server processes must handle load without interruptions for extended periods of time. Any corruption of the global state, whether unintentional or induced by an attacker, can be disastrous. Unfortunately, JavaScript features make it easy to slip and introduce security vulnerabilities which may allow a diversion of the control flow or even complete server poisoning. Hence, developers should be cautious when developing server applications in JavaScript. Yet the current trend is to build up one's application by loading (dynamically) a large number of third party libraries.

How to check all these libraries for potential vulnerabilities? Server-side JavaScript is more static than client-side JavaScript, so one may hope that static analysis might work. Unfortunately, the dynamic nature of the JavaScript language makes static analysis of JavaScript packages extremely difficult: only a handful of frameworks for static analysis can deal with exceptions and dynamic calls [12, 11]. Also the large number of libraries to be considered (and modeled) is a major hurdle. For example JAM requires modeling such dependencies in Prolog [10]. Runtime monitoring seems the only alternative *if* it can scale up to hundreds or thousands of concurrent requests.

How do we combine the flexibility of loading third-party libraries from a vibrant ecosystem with strong

security guarantees at an acceptable performance price? There is essentially no academic work addressing the problem of server-side JavaScript security. Our work targets this gap.

Threat Model

The server-side scenario assumes that libraries are actually executed on the server with server privileges. Hence, we assume *non-malicious libraries, although potentially vulnerable and exploitable (semi-trusted).* Because of vulnerabilities, they might end up doing something that they were not intended to do. A prototypical example is the `st` library, used to host static files, which was found to be vulnerable to a directory traversal attack.[1]

The purpose of our security model is to shield the semi-trusted libraries from *some* of the other libraries loaded in the package which may offer a functionality that we consider core. For example we may want to filter access by the semi-trusted library to the trusted library offering access to the file system.

We consider outright malicious libraries out of scope from our threat model, albeit one could use NodeSentry equally well to fully isolate a malicious library. We believe that the effort to write policies for *all* other possible libraries to be isolated from the malicious one would outweigh the effort of writing the alleged benign functionalities of the malicious library from scratch.

NodeSentry

The key idea of our proposal, called NodeSentry, is to use a variant of an inline reference monitor [20, 8] as modified for the Security-by-Contract approach for Java and .NET [7] in order to make it more flexible. Specifically, we do not embed the monitor into the code as suggested by most approaches for inline reference monitors but inline only hooks in a few key places, while the monitor itself is an external component. In our case this has the added advantage of potentially improving performance (a key requirement for server-side code) as the monitor can now run in a separate thread and threads that do not call security relevant actions are unaffected. Further, and maybe most important, we do not limit ourselves to purely raising security exceptions and stopping the execution but support policies that specify how to "fix" the execution; the ability to continue executing after security errors is an essential requirement for server-side applications.

In order to maintain control over all references acquired by the library, e.g., via a call to `"require"`, NodeSentry applies the *membrane* pattern, originally proposed by Miller [15, §9] and further refined by Van Cutsem and Miller [23]. The goal of a membrane is to fully isolate two object graphs [15, 23]. Intuitively, a membrane creates a shadow object that is a "clone" of the target object that it wishes to protect. Only the references to the shadow object are passed further to callers. Any access to the shadowed object is then intercepted and either served directly or eventually reflected on the target object through handlers. In this way, when a membrane revokes a reference, essentially by destroying the shadow object [23], it instantly achieves the goal of transitively revoking all references as advocated by Miller [15].

Security Policies

We have identified *two* possible points where the policy hooks can be placed. These coincide with two distinct types of policies: on the *public interface of the library itself* with the outer world, on the *public interface of any depending library* (both built-in, core libraries and other third-party libraries), or in both places.

Upper-bound policies are set on each member of the public interface of a library with the outer world. Those interfaces are used by the rest of the application to interact with it. It is the ideal location to do all kinds of security checks before the library code is executed, or right after the library returns. For example, these checks can be used (i) to implement web application firewalls and prevent malformed or maliciously crafted URLs from entering the library or (ii) to implement a number of security checks used for web-hardening, like e.g., enabling the HTTP Strict-Transport-Security header [13], set the Secure and/or HttpOnly Cookies flags [2] or configure a Content Security Policy (CSP) [21]. Another example of a useful policy would be to block specific clients from accessing specific files via the web server.

Lower-bound policies can be installed on the public interface of any depending library, both built-in core libraries (like e.g., `"fs"` for file system access)

[1] https://nodesecurity.io/advisories/st_directory_traversal & http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3744

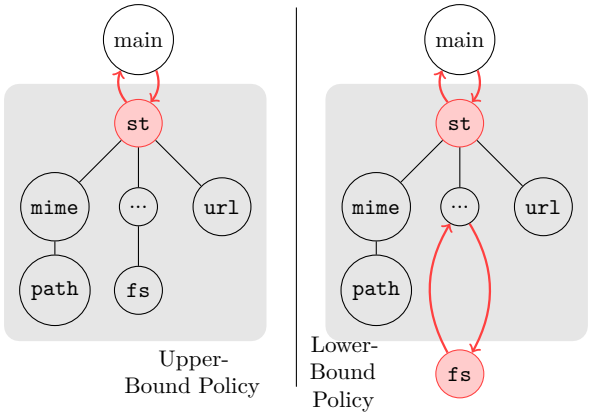

Figure 1: NodeSentry allows policies to be installed on the public interface (*Upper-Bound*) of the library to be secured (indicated by the gray rectangle that shows the inner dependency graph) and on the public interface of any depending library (*Lower-Bound*)

or any other third-party library. Such a policy could enforce e.g., an application-wide *chroot jail* or a fine-grained access control policy such as restrict reading to a specific set of files or prevent all write actions.

Figure 1 depicts interactions with these two types of policies with the red arrows and highlights the isolated context or membrane with a grey box. The amount of available policy points is thus a trade-off between performance (less points mean less checks) and security (more points mean a more fine-grained policy).

Conclusions

Among the various server-side framework, Node.js has emerged as one of the most popular frameworks. Its strengths are the use of JavaScript, an efficient runtime tailored for cloud-based event parallelism, and thousands of third-party libraries.

Yet, these libraries are also a source of potential security threats. Since the server runs with full privileges, a vulnerability in one library can compromise one's entire server. This is indeed what recently happened with the `"st"` library used by the popular web server libraries to serve static files.

In order to address the problem of least privilege integration of third party libraries we develop NodeSentry, a novel server-side JavaScript security architecture that supports such least-privilege integration of libraries, and that builds on the implementation of membranes.

Our enforcement infrastructure can support a simple and uniform implementation of security rules, starting from traditional web-hardening techniques to custom security policies on interactions between libraries and their environment, including any dependent library.

References

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 1–10, 2012.

[2] A. Barth. RFC 6265: HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265, 2011.

[3] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *Principles of Security and Trust*, pages 159–178. 2014.

[4] N. Bielova. Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser. *Journal of Logic and Algebraic Programming*, 2012.

[5] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. https://distrinet.cs.kuleuven.be/software/FlowFox/.

[6] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure Multi-Execution of Web Scripts: Theory and Practice. *Journal of Computer Security*, 22(4):469–509, 2014.

[7] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the. net platform. *Information Security Technical Report*, 13(1):25–32, 2008.

[8] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Cornell University, 2003.

[9] D. Flanagan. *JavaScript: the definitive guide.* " O'Reilly Media, Inc.", 2002.

[10] M. Fredrikson, R. Joiner, S. Jha, T. Reps, S. Hassen, and V. Yegneswaran. Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2012.

[11] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for javascript. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2012.

[12] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150, 2010.

[13] J. Hodges, C. Jackson, and A. Barth. Rfc 6797: Http strict transport security (hsts). http://tools.ietf.org/html/rfc6797, 2012.

[14] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 481–496, 2010.

[15] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[16] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 736–747, 2012.

[17] A. Ojamaa and K. Düüna. Assessing the Security of Node.js Platform. In *Proceedings of the International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 348–355, 2012.

[18] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 47–60, 2009.

[19] G. Richards, C. Hammer, F. Z. Nardelli, S. Jagannathan, and J. Vitek. Flexible Access Control for JavaScript. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2013.

[20] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[21] S. Stamm, S. Brandon, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2010.

[22] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[23] T. Van Cutsem and M. S. Miller. Trustworthy Proxies: Virtualizing Objects with Invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 154–178, 2013.