Hindawi
Security and Communication Networks
Volume 2019, Article ID 9629034, 21 pages
https://doi.org/10.1155/2019/9629034

Research Article

A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries

Neline van Ginkel,[1] **Willem De Groef**,[1] **Fabio Massacci**,[2] **and Frank Piessens**[1]

[1] *imec-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium*
[2] *Department of Information Engineering and Computer Science (DISI), University of Trento, Via Sommarive 9, 38123 Trento, Italy*

Correspondence should be addressed to Neline van Ginkel; neline.vanginkel@cs.kuleuven.be

Received 27 December 2018; Accepted 17 March 2019; Published 2 May 2019

Guest Editor: Gabriele Tolomei

The popularity of the JavaScript programming language for server-side programming has increased tremendously over the past decade. The Node.js framework is a popular JavaScript server-side framework with an efficient runtime for cloud-based event-driven architectures. One of its strengths 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 an entire server. In order to support the secure integration of libraries, we developed 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. We discuss the design and implementation of NODESENTRY and present its performance and security evaluation.

1. Introduction

Over the past decade, JavaScript has become one of the most widely used programming languages. It has been the number one scripting language for client-side web-scripts for a long time. However, since the mid-2000s, there has also been a growing interest in server-side JavaScript. An important driver is the enormous engineering efforts invested both in JavaScript as a programming language and in the underlying virtual machines and JIT compilers. The huge competition between the main browser vendors to build the fastest browser has produced JavaScript engines that run significantly faster than their predecessors. Another driver is the fact that many web developers are already familiar with client-side JavaScript, as part of writing front-ends of web applications. The step to server-side JavaScript can potentially allow an organization to take better advantage of the available talent pool.

The Stack Overflow web site, one of the most popular platforms for users to ask and answer questions on software development matters, organizes a yearly survey amongst its visitors. The 2016 survey [1] had 56,033 participants and shows that those developers use JavaScript more often than any other programming language. Even back-end developers are more likely to use it than any other language.

JavaScript has many advantages for web development [2]. In particular it allows the easy combination or *mash-up* of content and libraries from disparate third parties. This flexibility comes however at the price of significant security threats [3, 4], and researchers have proposed a number of client-side solutions to contain them—we discuss these approaches in more detail in Section 9.

These security threats are at least as significant at the server side: applications run without sandboxing, often in the same shared-memory address space, 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. Section 2.2 gives an overview of attack vectors for server-side JavaScript applications.

One particular security issue with server-side JavaScript is the fact that JavaScript makes it very easy to include third-party libraries into your application. Vulnerabilities

```
1  var mime = require("mime")
2  var path = require("path")
3  var fs;
4  try { fs = require("graceful-fs") }
5  catch (e) { fs = require("fs") }
```

Listing 1: The module loading system in Node.js.

in such included libraries undermine the security of any application that includes them. As a solution to this problem, some of the authors of this paper designed and published the NodeSentry security framework [5, 6], a server-side JavaScript security framework for confining third-party libraries. NodeSentry is implemented as an extension of Node.js, the most popular server-side JavaScript implementation.

This journal version of the NodeSentry conference paper extends the conference publication with a more detailed description of the framework, additional and updated benchmarks, an extensive security evaluation, and a more extensive discussion and comparison with related work.

Contributions. In summary, this paper proposes and evaluates a solution to the problem of secure integration of JavaScript libraries with the following contributions:

(1) NodeSentry, a novel server-side JavaScript security architecture;

(2) policy infrastructure that allows one to subsume and combine common web hardening techniques and measures, common and custom access control policies on interactions between libraries and their environment, including any dependent library;

(3) description of the key features of the implementation of NodeSentry and its policy infrastructure in Node.js;

(4) practical performance evaluation of an implementation of NodeSentry;

(5) an extensive, systematic security evaluation, with a focus on secure deployment and integration within existing code bases.

The rest of this paper is structured as follows. First we briefly describe Node.js and some of its security issues in a background section. Then we define the problem we address in this paper in Section 3. We describe the design, usage model, and implementation of our solution in Sections 4, 5, and 6, respectively. In Section 7 we evaluate our solution in terms of performance and security. The paper ends with a discussion (Section 8), a comparison with related work (Section 9), and a conclusion (Section 10).

2. Background

2.1. Node.js and Its Ecosystem of Third-Party Libraries. Node.js is an open-source, cross-platform runtime environment for developing server-side web applications, developed by Ryan Dahl in 2009.

The runtime environment that drives Node.js is built upon Google's V8 engine and runs on most operating systems including OS X, Linux, and Microsoft Windows. Most of the basic modules, e.g., for file system access and networking, are written in JavaScript.

Node.js is based on an event-driven architecture with asynchronous I/O in mind and is meant to optimize throughput and scalability in I/O bound and/or real-time web applications.

Node.js's architecture is designed to bring event-driven programming to web server development. It makes it easy for developers to create high performance, highly scalable server software, without having to struggle with threading. Using a simplified model of event-driven programming, one that uses callbacks, prevents having to work with concurrency, as is often the case with other server-side programming languages.

The standard library of Node.js is quite extensive: it supports functions including system I/O, all types of networking (ranging from raw UDP or TCP to HTTP and TLS), cryptography, data streams, and handling binary data. In 2010, the npm package manager for Node.js was introduced to make it easier to publish and share Node.js libraries. The npm tool can be used to access the online npm registry (https://npmjs.com), to organize the installation, and to manage third-party Node.js libraries. After installing a Node.js library, it can be loaded anywhere in the application by calling the `require` function, available in every Node.js context. At the time of writing, the official npm registry hosts over half a million libraries.

Loading works by reading the JavaScript code (from memory or from disk), executing that code in its own name space, and returning an `exports` object, which acts as the public interface for external code.

The Node.js module loading system is very easy to use in practice. In Listing 1, on line 2, the variable `path` will be an object with properties including `path.sep` that represents the separator character and the function `path.dirname` that returns the directory name of a given file path.

Libraries can also be dynamically loaded at any place in a program. For example on line 4, the program first tries to load the `graceful-fs` library. If this load fails, e.g., because it is not installed, the program falls back into loading the original system library `fs` (line 5). In this example constant strings are provided to the `require` function but this is not necessary. A developer can define a variable `var lib='fs'` and later on just call `require(lib)` where `lib` is dynamically evaluated.

```
1  var http = require("http");
2
3  let server = http.createServer((request, response) => {
4          if (request.method === "POST") {
5                  let data = "";
6                  let appendChunk = (chunk) => { data += chunk; }
7                  let fetchStockInfo = () => {
8                          let stockQuery = eval("(" + data + ")");
9                          // do something with the parsed data
10                         getStockPrice(stockQuery.symbol);
11                         ...
12                 };
13
14                 request.addListener("data", appendChunk);
15                 request.addListener("end", fetchStockInfo);
16         }
17 });
18 server.listen(1337, "127.0.0.1");
```

Figure 1: Example code of a Node.js application vulnerable to an injection attack. Just as in a client-side context, the call to `eval`, on line 8, must be considered dangerous [7] and makes the example vulnerable to an injection attack.

The resulting ecosystem is such that almost all applications are composed of a large number of libraries which recursively call other libraries. The most popular packages can include hundreds of libraries: `jade`, `grunt`, and `mongoose` make up for more than 200 included libraries each (directly or recursively); `express`, a popular web package includes 138, whereas `socket.io` can be unrolled to 160 libraries.

As a concrete example of how this can impact security, consider the `npm-www` JavaScript package maintainer application. This application uses around 100 libraries. One of these is the library `st` which is developed specifically to manage static file hosting for the back-end of the web site. The `st` library itself relies on access to around 10 other libraries, such as the `http` and `url` package, to process URLs, and the `fs` package to access the file system. Unfortunately, the `st` library turned out to be vulnerable to a directory traversal bug (https://www.npmjs.com/advisories/36 & http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3744) which allowed it to serve essentially all files on the server, thus leading to a potential massive compromise of all activities (http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers).

2.2. Server-Side JavaScript Security. The Node.js community always has a strong focus on the scalability of the platform, and security had a rather low priority—see, for example, the Node Security Project in Section 7.2 and the fact that a server-side JavaScript application by default does not run in a shielded environment. However, script injection vulnerabilities are just as easily introduced in a server-side application as in a client-side application. The impact of a successful injection attack can also be far more critical and damaging.

Ojamaa and Duuna [8] discuss several potential security weaknesses or pitfalls of the Node.js platform. They base their findings on their own experience with Node.js and on general web application security knowledge, like, for example, OWASP. They highlight issues including the fragility of Node.js applications, as any programming mistake in the single-threaded event loop might terminate the whole application, or the fact that there might be malicious installation scripts in an external Node.js package. Many of the issues have to do with the fact that server-side JavaScript is still JavaScript. Just like on the client-side, it is possible to (unwillingly) introduce bugs into JavaScript programs that might lead to, for example, an injection vulnerability. Figure 1 shows example code of a HTTP server implementation that uses the `eval` function to dynamically evaluate user provided JSON data.

Exploitation of server-side JavaScript looks more like triggering a SQL injection than like performing a cross-site scripting attack. There is no need for an attacker to set up a victim, for example, via a social engineering e-mail, like what is normally done for a reflected or DOM-based cross-site scripting attack.

By simply sending carefully, arbitrarily crafted (in our example case HTTP) requests, the attacker can manipulate the global state of the server process.

The defenses against server-side injection attacks have therefore a lot in common with typical SQL injection protection. Validation of user input is the most obvious and by far the simplest but most effective defense. Avoiding the `eval` function at all costs, is also something very well known and recommended by security experts [9]. In our example case, shown in Figure 1, JSON parsing should have been done via a safer alternative such as `JSON.parse`.

The security researcher Bryan Sullivan has presented an overview of the most relevant types of attacks for server-side JavaScript injection attacks [10]. We highlight three of them to give the reader a flavor of what types of attacks might be expected from an attacker and the amount of skill that is required for a successful attack.

Denial-of-Service. Due to the single-threaded event loop architecture of Node.js, any time consuming operation will block the main thread. No new network connections will be accepted as long as the main thread is busy. As many use cases for server-side applications are I/O bound, Node.js has adopted the concept of nonblocking I/O by the extensive

use of callbacks. A denial-of-service attack for our example could be easily triggered by sending, for example, code for an infinite loop `while(1)` or by exiting the current process via `process.exit()`. The end result is a server process that gets stuck, uses 100% of its processor time, or is otherwise unable to accept, process, or respond to any other incoming request.

This attack is much more effective than a regular distributed denial-of-service attack. Instead of flooding the target with millions of requests, only a single HTTP request is sufficient to completely disable the target victim server.

File System Access. One of the built-in functionalities of Node.js is its API for file system access. Via this API it is possible to read, write, and append to potentially any file on the file system and to list the contents of directories. As an example, an attacker could dynamically load the `fs` library via the appropriate attack payload and write arbitrary binary executables to the target server, by sending the command `require('fs').writeFileSync('/usr/local/bin/foo', 'data in base64 encoding','base64');`.

Execution of Arbitrary Code/Binaries. After dropping a binary executable on the target server, the only thing that is left to do for a successful attack is executing the binary. Node.js includes a `child_process` module that provides the ability to spawn arbitrary child processes. Via the attack payload `require('child_process').spawn(filename);` it would be possible to execute the previously written executable on the target server. At this point, any further exploitation is only limited by the attacker's imagination.

Defenses. How can an application developer that includes third-party libraries check those libraries for potential vulnerabilities? This is unfortunately a hard problem. Static analysis of JavaScript code is known to be hard because of the complexity of the language [11, 12]. Furthermore, the large quantity of libraries to be considered and modeled is another major hurdle. For example, JAM requires modeling such dependencies in Prolog [13].

Hence, this paper pursues an approach based on runtime monitoring. Our objective is to build a practical mechanism that an application developer can use to confine third-party libraries included in his application.

3. Problem Statement

The problem we address in this paper is the confinement of nonmalicious but potentially vulnerable third-party libraries in Node.js applications. We want to design a security framework, NodeSentry, that enables developers to include third-party libraries more securely by limiting the privileges given to such libraries. When loading a third-party library, the developer can enforce a policy on the interactions of that library with its clients and with other libraries.

3.1. Threat Model. We consider an attacker that can interact with a Node.js web application. The attacker knows what third-party libraries the application includes, and will try to exploit vulnerabilities in these third-party libraries. This is a realistic scenario, since many web application frameworks can be recognized (fingerprinted) by specific aspects of their output. An attacker with knowledge of a vulnerability in a commonly used third-party library can try to exploit that vulnerability and, if successful, conclude that the third-party library is indeed used in the web application.

NodeSentry is intended to confine such *nonmalicious libraries, although potentially vulnerable and exploitable (semitrusted)*, such as the `st` library. The objective of our security solution is to limit the damage that an attacker can do by exploiting vulnerabilities in such semitrusted libraries. For example we may want to filter access by the semitrusted 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 the policies for *all* other possible libraries to be isolated from the malicious one by far outweighs the effort of writing the alleged benign functionality of the malicious library from scratch.

Given the fact that NodeSentry has a programmatic policy and that policy code can effectively modify how the enforcement mechanism functions, it could be possible to introduce new vulnerabilities into the system via a badly written policy, e.g., if the policy code interacts with clients' requests. However, we consider the production of safe and secure policy code an interesting but orthogonal—and thus out-of-scope—issue, for which care must be taken by the policy writer to prevent mistakes/misuse.

4. NodeSentry

The key idea for NodeSentry is to use a variant of an inline reference monitor [14, 15] as modified for the Security-by-Contract approach for Java and .NET [16, 17] in order to make this monitor more flexible. We do not directly embed the monitor into the code, as suggested by most approaches for inline reference monitors, but inline only hooks in the required places, and these hooks call a policy decision point (implemented as a JavaScript object).

Further, we do not limit ourselves to purely raising security exceptions and stopping the execution but support policies that specify how to "fix" the execution [18–21]. This is another essential requirement for server-side applications which must keep going.

4.1. Membranes. In order to maintain control over all references acquired by the library, e.g., via recursive calls to `require`, NodeSentry applies the *membrane* pattern, originally proposed by Miller [22, §9] and further refined by Van Cutsem et al. [23]. The goal of a membrane is to fully isolate two object graphs [22, 23]. This is particularly important for dynamic languages in which object pointers may be passed along and an object may not be aware of who still has access to its internal components. The membrane also allows to intervene whenever code tries to cross the boundary between object graphs.

Intuitively, a membrane creates a shadow object that is a "clone" of the target object that it wishes to protect.

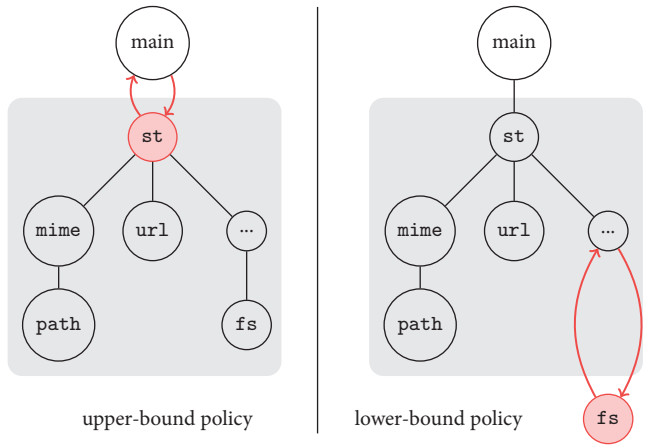

Figure 2: NodeSentry allows policies to be installed both on the public interface of the secure library (*upper-bound policies*) and on the public interface of any depending library (*lower-bound policies*).

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 [22].

4.2. Policies. The NodeSentry handler intercepts all interactions that cross the membrane, and all these interactions can be checked for compliance by a policy decision point provided by the application developer using NodeSentry.

This policy decision point can be seen as a standard security automaton: if it receives an action to check and the security automaton can make the corresponding transition, then the object proxied by the membrane is called and the (proxied) result is returned; if the automaton could not make a transition (i.e., the policy is violated), then a security countermeasure can be implemented by the policy decision point or, in the worst case scenario, a security exception will be automatically raised.

Hence, the notion of *policy* in NodeSentry has two aspects: on the one hand, the policy specifies what goes inside the membrane, and on the other hand the policy specifies the allowed cross-membrane interactions.

With respect to placement of the membrane, we distinguish two useful types of policies. First, the membrane can be placed around the *public interface of the library itself* with the outer world, thus putting the library (and all libraries it depends on) in the membrane. But second, it can be useful to take some of the depending libraries outside the membrane, to specify the allowed interactions on the *public interface of that depending library*. This is often useful for built-in, core libraries, but can also be done for other more trusted third-party libraries.

With respect to specifying allowed interactions, this leads to two different types of policies:

① *Upper-bound policies* are set on each member of the public interface of a library itself with the outer world. Those interfaces are used by the rest of the application to interact with the third-party library. It is the ideal location to do all kinds of security checks when specific library functionality is executed, or right after the library returns control.

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 add extra security headers to the server response towards a client. Another example of a useful policy would be to block specific clients from accessing specific files via the web server.

② *Lower-bound policies* can be installed on the public interface of any depending library, typically on built-in core libraries (like, e.g., `fs`) but also on any other third-party library.

Such a policy could be used to enforce, e.g., an application-wide *chroot jail* or to allow fine-grained access control such as restricting reading to several files or preventing all write actions to the file system.

Figure 2 illustrates these two types of policies with the red arrows and highlights the isolated context or membrane with a grey box. All interactions that cross the grey box boundary will be instrumented by NodeSentry for policy checks. Hence the choice of the membrane position is a trade-off between performance (fewer membrane crossings means fewer runtime checks) and security (more membrane crossings may offer opportunities for a more fine-grained policy).

A developer wishing to use NodeSentry only needs to replace the `require` call to the semitrusted library with a `safe_require`. This approach makes it possible to implement a number of security checks used for web hardening like, e.g., enabling the HTTP Strict-Transport-Security header [24], set the Secure and/or Http-Only Cookies flags [25], or configure a Content Security Policy (CSP) [26], in quite a modular way without affecting the work of rank-and-file JavaScript developers.

```
1  // the function transforms the given url into
2  // a path on the local system
3  Mount.prototype.getPath = function (u) {
4      u = path.normalize(url.parse(u).pathname
5      .replace(/^[\/\\]?/, "/"))
6      .replace(/\\/g, "/")
7      // ...
8  };
```

Listing 2: Simplified code snippet from the 'st' library.

```
1  require("nodesentry");
2  var http = require("http");
3  var st= safe_require("st", /* policy object */);
4  var handler = st(process.cwd());
5  http.createServer(handler).listen(1337);
```

Listing 3: Safely require a library.

5. Usage Model

We first describe the usage model [27] of NodeSentry for a fictive developer that has chosen to use the `st` library in her application to serve files to clients. In Section 5.1 we give an overview of the different steps of NodeSentry while it enforces a policy to secure the library.

The `st` library version < 0.2.5 has a potential directory traversal issue because it did not correctly check the file path for potential directory traversal. The snippet in Listing 2 shows a simplified version of the code.

By itself, this may *not* be a vulnerability: if a library manages files, it should provide a file from any point of the file system, possibly also using '. .' substrings, as far as this is a correct string for directory. However, when used to provide files to clients of a web server based on URLs, the code snippet becomes a serious security vulnerability.

An attacker could expose unintended files by sending, for example, a HTTP request for `/%2e%2e/%2e%2e/etc/passwd` (%2e is the URL encoding for . (dot)) towards a server using the `st` library to serve files.

It is of course possible to modify the original code, within the `st` library's source code, to fix the bug, but this patch would be lost when a new update to `st` is done by the original developers of the library. Getting involved in the community maintenance of the library so that the fix is inserted into the main branch may be too time demanding, or the developer may just not be sufficiently skilled to get it fixed without breaking other dependent libraries, or just have other priorities altogether.

The developer could instead merge the "fix" into the main code trunk but this "fix" might also be an actual "bug" for other developers that want to use the `st` library for other purposes.

In all these scenarios, the application of NodeSentry is the envisaged solution. The `st` library is considered semitrusted and a number of default web hardening policies are available in the NodeSentry policy toolkit. In the evaluation in Section 7.2, we go into more detail on secure deployment and how useful and practical NodeSentry is to fix real-life security issues.

The only adjustment is to load NodeSentry and to make sure that `st` is safely required so that the policy, given as a parameter object, becomes active.

The code snippet in Listing 3 is an example of an upper-bound policy decision point, as shown in Figure 2. After loading NodeSentry, policies can be (recursively) enforced on libraries by loading them via the newly introduced `safe_require` function. In our running example, when the policy for the requested URL detects malicious characters, it returns a pointer to a different page that could show a warning message. This functionality (a feature we call *policy execution correction*) is important in a server-side context where terminating the server with a security exception is undesirable.

If the policy in Listing 4 would be activated, all URLs passed to `st` would be correctly filtered. The policy states that if a library wants to access the URL of the incoming HTTP request (via the method `IncomingMessage.url`), we first test it for the presence of a directory traversal attack. If so, we return a different URL that points to a warning HTML page. In both a benign or malicious situation, a call to `IncomingMessage.url` would return a URL string and does not break the original contract of the API.

```
1  if (method === "IncomingMessage.url") {
2      var regex = new RegExp(/[\\/]\.\.[\\/]/ig);
3
4      if (regex.test(origValue.replace('%2e', '.'))
5          return "/your_attack_is_detected.html";
6      else
7          return origValue;
8  }
```

Listing 4: Example of a policy on a property lookup.

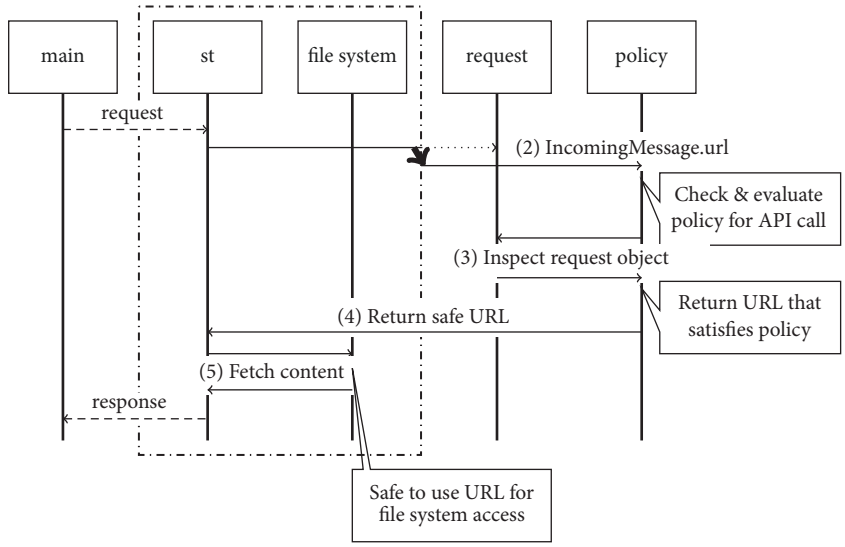

Figure 3: Interaction diagram of the running example from Section 5. The membrane is shown as the dash-dotted line. The interception of the API call `IncomingMessage.url` to read the requested URL is shown as a fat arrow.

5.1. Interactions Exemplified. Figure 3 shows the interaction diagram of the running example. The Node.js main event loop handles an incoming request and passes it to the `st` library. Next, the library needs to parse the requested URL in order to serve the corresponding file from the file system. The call for `IncomingMessage.url` crosses the membrane and gets forwarded to the policy object for evaluation. During the evaluation, the policy checks the requested URL and makes sure that it returns a safe URL to the `st` library. Finally, the library continues its normal behavior: reading the requested file (or a safe alternative) from the file system and sending back the response to the main method.

6. Implementation

This section reports on our development of a mature prototype that works with a standard installation of Node.js.

The crux of our implementation relies on the membrane pattern. We wrap a library's public API with a membrane to get full mediation, i.e., to be sure that each time an API is accessed, our enforcement mechanism is invoked in a secure and transparent manner. We detail on this in the first subsection.

In the second subsection, we discuss how we coped with the problem of safely requiring libraries. NodeSentry needs to know which libraries are recursively loaded. Therefore we designed a custom module loader, relying for a part on the original module loader and allowing us to specify a custom `require` wrapper function.

In the third subsection, we go into detail on how to exactly write policies and how these policy objects interact with a membrane. In NodeSentry, policies are written as objects that define the custom behavior of fundamental operations on objects.

6.1. Membranes. NodeSentry works with the latest Node.js versions and relies on the ES6 JavaScript standard. Membranes require this standard, in order to implement fully transparent wrappers. Membranes also build on WeakMaps, to preserve object identity between the shadow object and the real object (1) across the membrane and (2) on either side of the membrane.

```
1  function requireWrapper (require) {
2      return function require_log (path) {
3          console.log("require('" + path + "')");
4          return require(path);
5      }
6  };
```

Listing 5: Require wrapper to log which libraries are loaded.

```
1  function newMembrane (ifaceObj, policyObj) {
2      return require("membrane")
3                .makeGenericMembrane(ifaceObj, policyObj)
4                .target;
5  }
```

Listing 6: Wrapping an interface with a policy.

We rely on the ES6 Reflect and Proxy API by Node.js and use the implementation of a generic membrane abstraction by Van Cutsem (https://github.com/tvcutsem/harmony-reflect/blob/master/examples/generic_membrane.js), which is used as a building block of our implementation and is available via the `membrane` library, as shown in the code snippets below. The current prototype of NodeSentry runs seamlessly on a standard Node.js v6.0 or higher. An older prototype, which uses the shim module by Van Cutsem (https://github.com/tvcutsem/harmony-reflect) for the Reflect and Proxy modules, runs on v0.10 or higher.

In Listing 5 we show an example of a custom `require_log` wrapper function that logs to the console which libraries are loaded and relies on another `require` function (i.e., the default, built-in function) to effectively load a library into memory. The result of the `require(path)` call on line 4 is a JavaScript object `ifaceObj` with properties representing the application interface of the library.

We rely on a generic implementation, available via the `membrane` library, to wrap a membrane around a given `ifaceObj` with the given handler code in `policyObj`, as shown in Listing 6.

6.2. Safely Requiring Libraries. While loading a library with `safe_require`, the original `require` function is replaced with one that wraps the public interface object with a membrane and a given (upper-bound) policy.

Our first stepping stone is to introduce the `safe_require` function. Its main goal is to virtualize the `require` function so that any additional library that will be loaded as a dependency can be intercepted.

At the heart of the `safe_require` function, as shown in Listing 7, is the `loadLib` function (line 3) that initializes a new module environment and loads it with a custom `membranedRequire` function. This function will make sure that every call for a dependent library will be intercepted and that the library itself is properly wrapped, even in a recursive way. This extra indirection in the library loading process allows us to enforce lower-bound policies on the public interface of any depending library. We elaborate more on this in a later paragraph.

Finally, the API object (`exports`) gets wrapped in a new membrane, based on a given `policy`, as shown on line 12. This line in particular makes it possible to enforce upper-bound policies on the public interface of the library.

This whole operation does not normally cost any additional overhead since it is only done at system start-up and is therefore completely immaterial during server operations. If `require` is called dynamically we can still catch it. Either way, each time the function is called we can now test whether a library we want to protect has been invoked.

Lower-bound policies are enforced by overwriting the `require` function with the `membranedRequire` function, which is shown in Listing 8. By controlling the loading context of a library and providing it with our own `require` function, we can intercept all its calls and those from any depending library. At interception time, if the library has been identified as needing control from a lower-bound policy, we wrap the public interface object of that depending library with a membrane (see line 10 in Listing 8). If decided so, all interactions between the library and its depending library are effectively subject to the lower-bound policy. If not, the original interface objects get returned (see line 13).

6.3. Policy Objects. In NodeSentry, a `policyObj` is a regular JavaScript object that holds code that represents a security policy. As shown in the previous section, that code is used in a wrapper around the original Node.js API calls. This wrapper basically hijacks the original call just like an advice function in aspect-oriented programming. This way,

```
function safe_require (libName, policyObj) {

    function loadLib () {
        var mod = new Module(libName);
        // enforce lower-bound policies
        mod.require = membranedRequire(policyObj);

        return mod.loadLibrary();
    };

    // enforce upper-bound policies
    return newMembrane(loadLib().exports, policyObj);
}
```

Listing 7: The implementation of the `safe_require` function.

```
function membranedRequire (policyObj) {
    return function (libName) {
        var libexports;

        // [...] load the requested library
        // and assign to libExports

        if (lowerBoundPolicyNeeds(libName)) {
            // enforce lower-bound policies
            return newMembrane(libExports, policyObj);
        } else {
            // enforce no policy
            return libExports;
        }
    }
}
```

Listing 8: The implementation of the `membranedRequire` function.

we allow custom code execution before and after the original call execution. The `policyObj` keeps track of which code to execute before/after which Node.js API call.

We have designed a simple domain-specific language (DSL), based on method chaining, that encodes this behavior and that allows policy writers to express a policy in JavaScript.

The current version of the DSL supports policies that can modify return properties of objects (using the `on` method on a policy) and policies that can execute custom functions before or right after an actual API call before it returns to the actual call site (using the methods `before`, `after`). It can also execute any other function via the `do` construct.

More formally, the DSL is structured as follows:

(i) `let policy = new Policy(`*policy-name*`)`, where *policy-name* is a string, creates an empty policy with a given name.

(ii) The first way to add a rule to a policy object is

`policy.on(`*interception-point*`).return(`*wrapper-function*`)`

where

(a) *interception-point* is a string that denotes a property exported by the library that the policy applies to (note that properties can also return function values, i.e. methods);

(b) *wrapper-function* is a function that takes one parameter.

The effect of this rule is that, on invocation of the getter for the intercepted property at runtime, the policy intervenes and instead invokes the *wrapper-function* with the original property value as actual parameter.

```
let returnErrorPageIfAttack = (url) => {
    if (/%2e/ig.test(url) === true) {
        return "/your_attack_is_detected.html";
    } else {
        return url;
    }
};
let policy = new Policy("st example")
        .on("IncomingMessage.url")
        .return(returnErrorPageIfAttack)
        .build();
```

Listing 9: Implementation of the fix for the `st` library using NodeSentry.

The return value of this call to *wrapper-function* becomes the return value of the intercepted invocation and, hence, the value of the property seen by clients of the library.

This is a very powerful mechanism that can be used to replace the value of properties of primitive types (like integers or strings), but it can also replace methods of an object with another method that wraps the original one.

(iii) It is often useful to specify that side-effecting operations should happen on lookup of a property. The following DSL construct does just that:

`policy.on(`*interception-point*`).do(`*on-advice*`)`

where *on-advice* is a function that takes two parameters. On every invocation of the getter for the intercepted property, the function *on-advice* will be invoked with, as arguments, the receiver object and the property value. The return value of the function is ignored.

(iv) In the case where the intercepted property is a method, it is also convenient to be able to specify code that should be executed before and after every invocation of the method (as opposed to before and after every lookup of the property). Our DSL supports three mechanisms to specify this:

`policy.before(`*interception-point*`).do(`*before-advice*`)`

`policy.after(`*interception-point*`).do(`*after-advice*`)`

`policy.after(`*interception-point*`).return(`*transform*`)`

The first two constructs invoke *before-advice* (respectively, *after-advice*) before (respectively, after) each invocation of the intercepted method. The function *before-advice* gets three arguments: the receiver of the invocation, the value of the method (as a function value), and the arguments array of the invocation.

The function *after-advice* in addition gets as fourth argument the return value of the invoked method.

The third construct just transforms the return value of the intercepted method using the provided *transform* function.

(v) Multiple calls that add rules to a policy object (on different interception points) can be chained, and then `policy.build()` is used to finalize the policy.

These features of the DSL are sufficient to understand the example policies in this paper. When reading the example policies, keep in mind that JavaScript is very flexible in argument passing: for instance, if an advice function only needs its first argument, it can be declared as a single-argument function even if it gets passed more arguments at runtime. These additional arguments will then just be ignored.

The full DSL supports a number of additional constructs, for instance, to specify the conditional invocation of rules.

Example Policy for the `st` *Example.* In Listing 9 we show the policy for the `st` example vulnerability mentioned in Section 5. We want to prevent an attacker from providing malicious input, without forwarding the input to the vulnerable `st` library.

Example Policy Enabling HSTS. As a simple example for the potential of NodeSentry we describe how we implemented the checks behind the `helmet` library, a middleware used for web hardening and implementing various security headers for the popular `express` framework (https://github.com/evilpacket/helmet).

It is used to, e.g., enable the HTTP Strict Transport Security (HSTS) policy [24] in an `express`-based web application by requiring each application to actually use the library when crafting HTTP requests. The HSTS policy is used to protect websites against protocol downgrade attacks.

The snippet in Listing 10 shows a NodeSentry policy that adds the HSTS header before continuing with sending the outgoing server response, via a call to

```
let addHSTSHeader = (response) => {
        let h = "Strict-Transport-Security";
        let v = "max-age=3600; includeSubDomains";

        return response.setHeader(h, v);
};

let policy = new Policy("HSTS Example")
        .before("ServerResponse.writeHead")
        .do(addHSTSHeader)
        .build();
```

Listing 10: A policy that automatically adds a HSTS header.

```
const https = safe_require("https", policy);
const fs = require("fs");
const options = { pfx: fs.readFileSync("server.pfx") };

https.createServer(options, (request, response) => {
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.end("Welcome on this web site");
}).listen(7777);
```

Listing 11: Using the HSTS policy from Listing 10.

```
HTTP/1.1 200 OK
Content-Type: text/plain
Strict-Transport-Security: max-age=3600; includeSubdomains
Date: Sun, 04 Dec 2016 13:50:02 GMT
Connection: keep-alive
```

Listing 12:

`ServerResponse.writeHead`, effectively mimicking the behavior of the original `helmet.hsts()` call.

The developer does not need to modify the original application code to exhibit this behavior. They only need to `safe_require` the library whose HTTPS calls they want to restrict. This can be done once at the beginning of the library itself, as customary in many Node.js packages.

In the code snippet in Listing 11, we initialize a HTTPS server by loading the `https` library with our example policy. The server needs access to an archive file for its key and certificate and sends back a static message when contacted on port 7777.

In Listing 12 are the HTTP response headers from a request made to the server from Listing 11 (https://localhost:7777/), clearly showing that the policy added the `Strict-Transport-Security` header.

Example Policy Preventing Write Access to the File System. The next example shows a possible policy to prevent a library from writing to the file system without raising an error or an exception. Whenever a possible write operation via the `fs` library gets called, the policy will silently `return` from the execution. The policy uses the `on` construct so that the real method call never gets executed, and thus effectively prevents writing to the file system.

It is possible to change this behavior by, e.g., throwing an exception or *chrooting* to a specific directory. A possible policy that wants to prevent a library from writing to the file system must cover all available write operations of the `fs` library and therefore requires in-depth knowledge of the internals of the built-in libraries. Such a policy is implemented in Listing 13.

Although our API is fairly simple and does not protect against unsafe or insecure policy code, we do provide some

```
//do not forward the call to the original API method
let doNothing = () => { return; }
let policy = new Policy("no writing to file system allowed")
        .on("fs.writeFile")
        .on("fs.write")
        .on("fs.writeFileSync")
        .on("fs.writeSync")
        .on("fs.appendFile")
        .on("fs.appendFileSync")
        .return(doNothing)
        .build();
```

Listing 13: Disabling access to the write operations in the `fs` library.

```
// toggle between plain Node.js and NodeSentry
var enable_nodesentry = true;

var http = require("http");
var st;

if (enable_nodesentry) {
  require("nodesentry");
  st = safe_require("st", null);
} else {
  st = require("st");
}

// actual benchmark application
var handler = st(process.cwd());
http.createServer(handler).listen(1337);
```

Figure 4: Our streamlined benchmark application implements a bare static file hosting server, by relying on the popular `st` and the built-in `http` libraries.

form of containment, as defined by Keil and Thiemann [28]. NodeSentry makes sure that the evaluation of a policy takes place in a sandbox so that it cannot write to other variables outside of the policy scope. Different from the work of Keil and Thiemann [28, §3.6], we rely on the built-in `vm` module of Node.js. As mentioned in Section 3, we do not explicitly protect against introducing new vulnerabilities via badly written policy code.

7. Evaluation

This section details our evaluation of both the performance cost and the security of NodeSentry. The main goal of our benchmark experiment is to verify the impact of introducing NodeSentry in an existing software stack. We evaluate the cost of both an empty policy and a meaningful policy.

We also evaluate secure deployment in terms of both effectiveness and ease of use. We show how NodeSentry can be used to secure real-world, existing vulnerable libraries, as mentioned in our threat model, and we try to give an indication as to how hard it is to weave the NodeSentry API within an existing code base.

7.1. Performance. Our first benchmark experiment aims to verify the impact of introducing NodeSentry on performance measured as *throughput*, i.e., the number of tasks or total requests handled by our server.

In order to streamline the benchmark and eliminate all possible confounding factors, we have written a stripped file hosting server that uses the `st` library to serve files. The *entire* code of the server, besides the libraries `http` and `st`, is shown in Figure 4. The only conditional instruction present in the code makes it possible for us to run the benchmark test suite at first for pure Node.js and then compare it with Node.js with NodeSentry enabled (with no specified policy). In a third benchmark we also implement a meaningful policy, as shown in Figure 5.

Each experiment (for plain Node.js, Node.js with NodeSentry with a `null` policy, and NodeSentry with a meaningful policy) consists of multiple runs. Each run measures the ability of the web server to *concurrently serve files to N clients*, for an increasingly large N, as illustrated in Figure 6. Each client continuously sends requests for files to the server throughout the duration of each experiment. At first only few clients are present (warm-up phase), and after few seconds the number of clients steps up and quickly reaches the total number N (ramp-up phase). The number of

```
var validUrl = function(url) {
        return url.indexOf("%2e") > -1 || url.indexOf("..") > -1;
};

var po = new node.Policy().on("IncomingMessage.url").return(function() {
                return "/redirect_to_404_page";
        }).if(function(incomingMessage, url) {
                return validUrl(url);
        }).build();
```

Figure 5: The policy implemented by the static file hosting server with a meaningful policy. This policy is implemented instead of the `null` policy in Figure 4.

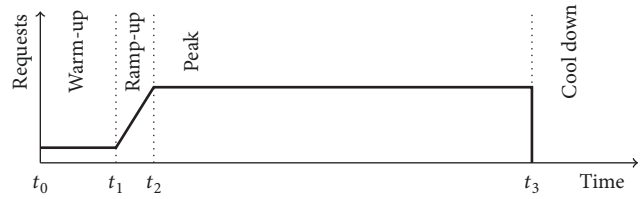

Figure 6: In our experimental setup, the load profile of the experiment varies between a minimum (the warm-up phase) and a maximum (the peak phase) of concurrent users. This is repeated for $N = 1..1000$ concurrent users sending requests to our server.

clients then remains constant until the end of the experiment (peak phase) with N clients continuously sending concurrent requests for files. We only measure the performance in the peak phase.

The experimental setup consists of two identical virtual machines with 8GB RAM and 8 virtual CPUs, in the same network. One machine is responsible for generating HTTP requests by spawning multiple threads, representing individual users. The second machine runs Node.js v8.7.0 and acts as the server. The load generating machine relies on a benchmarking framework developed by Heyman et al. [29]. The maximal load the framework could generate was 1200 concurrent users.

The results of the experiment are summarized in Table 1. This table reports the *throughput*: how many requests the system is able to concurrently serve as the number of clients increases. As can be seen, NodeSentry has a very low impact on throughput, even with a very high number of clients, and even if an actual policy is being enforced.

Table 1 suggests that, even at the highest load that our benchmarking framework can generate, the CPUs of the server are not fully loaded. Hence, we perform an additional experiment to measure the impact of NodeSentry on the request response time of individual requests. We used ApacheBench (`ab`) to measure the time needed per request, averaged over 100.000 requests. We ran `ab` on an identical virtual machine as those used in the experiments described earlier, with the Node.js server on a separate machine. We used a concurrency level of one, meaning we send only one request at a time, to prevent one request influencing the processing time of another request. The results are summarized in Table 2.

We can conclude from this experiment that the impact of the NodeSentry infrastructure (just the interception, with empty policy) on performance is negligible. Obviously, as soon as one implements policy logic, the performance cost depends on the policy logic, and the additional computation time shows up in the request response time. However, for the specific application and policy used in our benchmarks, the additional cost is small enough that it does not significantly impact application throughput even under a load of 1200 concurrent clients.

Finally, we also measured the impact of using NodeSentry on start-up cost. We measured the difference of using the regular `require` versus a `safe_require`, where we measure both using a `null` policy and a meaningful policy. We only measure the time it takes to execute the line of code containing the `require` or `safe_require` using the `console.time` functionality in Node.js. We started the process 100 times as warm-up and measured the average time for 1000 runs of the process. The results are summarized in Table 3. As expected `safe_require` takes slightly longer than `require`, but the impact is limited, especially since applications do not frequently `require` libraries except on start-up.

7.2. Secure Deployment. Securely deploying an existing Node.js application with NodeSentry is as simple as installing and loading the NodeSentry library, as clarified in Section 5.

Another aspect of secure deployment is the effectiveness of our security framework. We provide evidence of the security benefits that NodeSentry provides through the following experiment. The Node Security Project was a community initiative raising awareness about security-related problems within the Node.js ecosystem and maintained a list of advisories for all known, reported vulnerabilities of Node.js libraries, which is now integrated with npm (https://www.npmjs.com/advisories). We analyzed all the 73 vulnerabilities reported by the Node Security Project at the

Table 1: The throughput (total number of requests processed in 10 seconds) of the simple file server without NodeSentry, with NodeSentry with a `null` policy, and with NodeSentry with a sensible policy. The performance for all cases up to 800 users is near optimum, and there is no significant difference between the performance of the plain file server and the file server protected with NodeSentry.

Concurrent users	Node.js	NodeSentry	With Policy
1	10	10	10
10	99	99	100
20	199	199	200
50	500	499	499
100	999	997	993
200	1993	1997	1984
400	3959	3945	3897
600	5987	5878	5829
800	7777	7747	7706
1000	9948	9926	9336
1200	11710	11613	10556

Table 2: The average request response time.

Node.js	NodeSentry	With Policy
0.400ms	0.407ms	1.125ms

Table 3: The average time it takes to require the st library.

Node.js	NodeSentry	With Policy
16.696ms	19.075ms	20.939ms

Table 4: Summary of the reported vulnerabilities of the Node Security Project and their corresponding type of policy. About 95% are in scope for NodeSentry.

Type of policy	# Vulnerabilities involved
① Input filtering	31 (42%)
② Output filtering	7 (10%)
③ Additional logic	12 (16%)
④ Denial-of-Service filtering	19 (26%)
⑤ Out of scope	4 (5%)

time of the experiment (March 1, 2016) and investigated if and how these vulnerabilities could be mitigated by NodeSentry. To answer this question, we relied on the description of the vulnerability, as well as the proposed patch for the vulnerability if one was available (to determine if a NodeSentry policy could implement behavior that is semantically equivalent to the patch).

We have not implemented and tested policies for each issue individually, as this would require building test cases to confirm that benign functionality of the library is not prevented, as well as example attacks to confirm that the policy stops attacks. We do provide a full implementation of policies for a subset of the vulnerabilities below.

We classified the vulnerabilities in five separate categories, based on the type of policy required to fix the vulnerability. In defining the policies, we have tried to be as modular as possible: real system security policies are best given as collections of simpler policies, because a single large monolithic policy is more difficult to comprehend. The system's security policy is then the result of composing the simpler policies in the collection by taking their conjunction. This is particularly appropriate considering our scenario of filtering library actions: If the library may not be trusted to provide access to the file system, it may be enough to implement OWASP's check on file system management (e.g., escaping or file traversal). If a library is used for processing HTTP requests to a database, it could be controlled for URL sanitization. Each of those two libraries could then be wrapped by using only the relevant policy components, thus avoiding paying an unnecessary performance price.

We report on the result of the experiment below. For each of the 73 vulnerabilities, we report on whether the vulnerability can be mitigated by means of a NodeSentry policy, and if so by what type of policy.

The main results are summarized in Table 4. Out of the 73 vulnerabilities, only 4 could not realistically be mitigated with NodeSentry.

The complete list of vulnerable libraries, with a short explanation of the vulnerability type and their corresponding vulnerability category, can be found in the table in the appendix.

We now discuss each vulnerability category in more detail.

Vulnerability Categories. We have divided all 73 vulnerabilities into five separate categories, based on the type of policy that would fix their security issue. In the remainder of this section, we give details for each category and give an example policy for an existing vulnerability.

The first category contains libraries for which a policy is based on filtering incoming data before passing it on to a library. The second category contains libraries for which policies filter outgoing data, i.e., data coming from a library, after it has been processed. The third category combines all libraries that have policies that extend some functionality of the library, because they must be able to rely on original

```
let checkForValidPassword = (msg, fun, args, ret) => {
    if (args[0] === "access-key") {
        var input_key = ret;
        var configFile = require("./config");
        if (configFile.master.api.access_key !== input_key) {
            throw new Error("unauthorized access");
        }
    }
};

let policy = new Policy("tomato example")
        .after("IncomingMessage.header")
        .do(checkForValidPassword)
        .build();
```

Listing 14: Fixing a bug where the access key was not correctly checked in the `tomato` library.

```
let addUTFEncoding = (response, func, args) => {
    if (args[0] === 400) {
        let contentType = response.getHeader("content-type");
        if (contentType === null) {
            response.setHeader("content-type", "text/html; charset=utf-8");
        }
    }
};

let policy = new Policy("UTF8 encoding")
        .before("ServerResponse.writeHead")
        .do(addUTFEncoding)
        .build();
```

Listing 15: Fixing a bug where no content-type header was sent in the `express` library.

functionality of the library. The fourth category are denial-of-service vulnerabilities that cannot be handled correctly in all corner cases of their input. It is clear that a general policy implementation can only be coarse grained and only put some limit on the input. The fifth category contains libraries that have vulnerabilities that are too hard to fix with NodeSentry-style policies as they occur on a layer different from JavaScript (e.g., the vulnerability is located in a C library).

① *Input Filtering*: All policies within this category are based on the idea that the vulnerable library never gets access to the malicious input as it gets filtered before it can be effectively used. The examples from Section 5 fall within the category of input filtering.

Other examples of input filtering policies are the ones that filter incoming requests. The `tomato` library unintentionally exposed the admin API because it checked if the provided access key was *within* the configured access key, not equal to it. A possible policy for this vulnerability would implement a correct check and any unauthorized request would simply be filtered and left unanswered. The policy hooks in when the `tomato` library searches for the custom `access-key` HTTP header (Listing 14).

② *Output Filtering*: All policies within this category are based on the idea that the vulnerability in a library happens because their output can turn into malicious output in certain cases.

The `express` library did not specify a character set encoding in the content-type header while displaying a 400 error message, leaving the library vulnerable to a cross-site scripting attack. A NodeSentry policy for such a vulnerability could automatically attach the necessary header to the server response, right before sending it, effectively filtering and modifying the output (Listing 15). The policy only performs this operation if it detects a 400 error message being sent.

Another example for pure output filtering is the policy for the cross-site scripting vulnerability in `serve-index`, because the library did not properly escape directory names

```
let escapeDirectoryNames = (result) => {
        // an open source HTML sanitization library
        // linked to on the OWASP website
        var bleach = require("bleach");
        return bleach.sanitize(result);
};

let policy = new Policy("escape directoy names")
        .after("fs.readdir")
        .return(escapeDirectoryNames)
        .build();
```

Listing 16: Fixing a bug where directory names were not escaped in the `serve-index` library.

```
let verifyCorrectAlgorithm = (jwtObj, func, verifyArgs) => {
        var jws = require("jsws");
        var jwtString = verifyArgs[0];
        var options = verifyArgs[2];
        var header = jws.decode(jwtString).header;
        if (!~options.algorithms.indexOf(header.alg)) {
                throw new Error("invalid algorithm");
        }
};

let policy = new Policy("jsonwebtoken algorithm check")
        .before("jsonwebtoken.verify")
        .do(verifyCorrectAlgorithm)
        .build();
```

Listing 17: Fixing a bug where the algorithm used was not verified in the `jsonwebtoken` library.

when showing the contents of a directory. A NodeSentry policy could rely on a decent HTML sanitization library and filter and fix, if necessary, the resulting HTML of the library (Listing 16).

③ *Additional Logic*: Some policies need to extend the original behavior of a library, e.g., to strengthen certain conditional checks. Policies from this category are inherently specialized for one specific library.

A vulnerability in `jsonwebtoken` allowed an attacker to bypass the verification part by providing a token with a digitally signed asymmetric key based on a different algorithm from the one used by the library. The official patch for this security issue is to first decode the header of the token and explicitly verify whether the algorithm is supported (URL of the patch, as visited on November 4th, 2015: https://github.com/auth0/node-jsonwebtoken/commit/1bb584bc382295eeb7ee8c4452a673a77a68b687).

The exact same solution could be implemented with a policy for NodeSentry, which is in fact idempotent with the official patch. A NodeSentry policy wraps the `verify` API functionality, does the necessary check, and throws an error in case an invalid algorithm is specified (Listing 17).

④ *Denial-of-Service Filtering*: A denial-of-service filter is either a coarse-grained filter to limit the input to a specific regular expression or a very ad hoc filter that eliminates specific corner cases that would trigger the denial-of-service.

An example policy for the former case is the library `marked`. It was vulnerable to a regular expression denial-of-service (ReDoS) attack in which a carefully crafted message could cause a regular expression to take an exponentially long time to try to match the input. A quick fix might be to limit the length of the input to be matched.

An example of the latter case is the denial-of-service vulnerability in `mqtt-packet`. A carefully crafted network packet can crash the application because of a bug in the parser code. A quick fix could be to check for a valid protocol identifier and make sure that we catch the out of range exception when the vulnerability is triggered.

⑤ *Out of Scope*: Technically, there are no solid policies for libraries in this category. However, in some use cases it might be possible to construct a working policy but it would require an extensive case-by-case analysis and highly depends on the situation and context the library is used in.

For example `libyaml` relied on a vulnerable version of the original LibYaml C library. In this case, the patch against

the heap-based buffer overflow involved modifying C code to allocate enough memory for the given YAML tags. However, designing a policy that puts limits on the input of the wrapper library would severely limit the usefulness of the library in real-life.

Conclusions. Out of the original list of 73 vulnerable libraries, only 4 are out of scope and not generally fixable. This means that the majority of the vulnerable libraries could benefit from a security architecture like NodeSentry. About 43 vulnerabilities could be fixed with proper input filtering (31) or proper output filter (12). Only 7 vulnerabilities require a custom crafted policy. As input and output filtering policies are often generic (e.g., cross-site scripting or URL sanitization) and count for more than half of all our policies, the results seem to suggest that in practice even more libraries with *unknown* vulnerabilities could profit from NodeSentry. About one-fourth (19) of the vulnerabilities have to do with denial-of-service. In 13 cases, extremely long input can cause the regular expression implementation of Node.js to consume too much execution time. Limiting the input to a more reasonable size is probably the best fix for all of them, again suggesting that in the future more of these types of vulnerabilities will be automatically fixed. The other 6 cases require a truly custom fix.

Our analysis also suggests that NodeSentry could be used as a community-driven tool to provide (quick) patches to vulnerabilities before they are fixed in the original library. NodeSentry could even be the only way to enroll security patches, e.g., in case a library gets abandoned or if the original developers have no interest in fixing the issues. Enforcing general policies, like, e.g., the anti-directory traversal policy, could also prevent previously unknown vulnerabilities in libraries from popping up.

8. Discussion and Future Work

While our evaluation shows that NodeSentry can provide protection for a significant number of security threats, it also has some important limitations that we briefly discuss in this section, as well as how these limitations could be addressed with future work.

First, the privilege reduction that NodeSentry enforces on a third-party library depends on the policy provided by the application developer integrating the library. While the conference paper used the term *least-privilege* integration to describe the secure integration of libraries offered by NodeSentry, it would be more correct to describe this as *reduced-privilege* integration. There is no guarantee whatsoever that the remaining privileges are minimal in some sense. There is an independent line of research investigating approaches to infer least-privilege policies [30–32] and it would be an interesting topic for future work to try to integrate such policy inference. We expect this to be challenging, however, given the nature of the JavaScript language.

Second, an important disadvantage of NodeSentry is that it is a powerful tool and developers can easily make mistakes in writing policies that could result in new vulnerabilities. NodeSentry supports a kind of *aspect-oriented* programming: a policy programmer can inject arbitrary code at multiple points in the application. When used badly, this can negatively impact maintainability and understandability of the code. It would be beneficial to make sure that policy code has a kind of *precision* property; i.e., the code does *not* impact the execution unless some well-specified security property is violated. With that precision property, application developers do not need to worry about the impact of policy code on their application: the application is not affected when the program is not under attack. An interesting question for future work is how one could enforce such a precision property on NodeSentry policies.

Third, and related to the previous point, is the development of suitable policy languages for NodeSentry. The current prototype has an ad hoc domain-specific language implemented as a JavaScript API, but this policy language can definitely be further improved. For instance, policies in the current language may require updating when the API they protect is extended with new methods; i.e., there is no way to *quantify* over multiple methods, for instance, to forbid access to all write methods on a file system API. The design of a good policy language will need to balance expressivity, usability and understandability, and support for analysis (such as for enforcing the precision property discussed above).

We show in Section 7.2 that NodeSentry can resolve vulnerabilities in many cases. However, NodeSentry is by no means a silver bullet, and in some cases better alternative solutions are possible. For instance, if it is easy to solve the vulnerability directly in the library itself, that should be preferred, since this will fix the vulnerability for all users of that library. Another example is malicious libraries: if a developer considers a given library as possibly *arbitrarily* malicious, then a NodeSentry policy might have to be very defensive, checking, for instance, every return value of the library by recomputing it independently. While one can in principle write such a policy, it would obviously be less effort to write the desired library from scratch. The sweet spot for using NodeSentry is the protection against library behavior that leads to vulnerabilities in this specific application but might be acceptable behavior for other applications relying on that library. Another useful use case is patching of vulnerabilities in libraries when patching the library itself is not an option.

Finally, NodeSentry is a JavaScript specific security framework. An interesting question is whether the same approach can be ported to other programming languages or server-side frameworks. The key requirement seems to be that the language or framework should have good support for implementing the membrane object capability pattern. Investigating such ports to other platforms is a final very relevant path for future work.

9. Related Work

NodeSentry builds on two long-standing research lines. First, it is an application of the idea of *aspect-oriented programming* [33]: in aspect-oriented programming, a base-program can be enriched with aspects that specify additional

program functionality (*advice*) that must be executed at specific program points. The application of this idea for security has also been called *inlined reference monitoring* and been investigated intensively for integrating access control into application code [15, 34]. NodeSentry can be seen as an instance of this idea: policies provide advice that is executed when crossing the membrane.

Second, NodeSentry is an application of the ideas of the object-capabilities community. This community has been investigating the use of capabilities (unforgeable references) as a security mechanism for many years. The PhD thesis of Miller [22] provides an excellent history and overview of the field. In particular, Miller proposes a number of object capability patterns to address specific security problems, and the membrane pattern used in NodeSentry is a prominent example of these patterns. Miller's work on Caja [35] has contributed to the understanding of how JavaScript can be made a capability-safe language, and several authors have investigated the security properties that can be achieved using object capability patterns. Maffeis et al. [36, 37] were the first to study the isolation of JavaScript using object capability techniques in a formal setting. Devriese et al. [38] proposed a more advanced formalization that more completely captures the notion of capability safety. In particular, their technique is powerful enough to reason about isolated components with restricted communication between these components. Very recently, Swasey et al. [39] propose the first program logic that can compositionally specify and verify object capability patterns in JavaScript-like languages. The membrane pattern that we rely on in NodeSentry is one of the object capability patterns they specify in their logic. These foundational works on developing the formal basis for proving properties about object capability patters are complementary to our work on NodeSentry. It is conceivable that they could in the future be used to provide provable guarantees about the security of systems like NodeSentry.

With respect to related work on security architectures for JavaScript, there is a substantial body of work on securing JavaScript on the client-side, including approaches for sandboxing (e.g., based on Google's Caja or [40, 41]), approaches that do information flow control [20, 42, 43], as well as approaches that instrument the browser with a number of policies [44] or try to guarantee control-flow integrity at a web-firewall level [45]. Bielova presents a good survey on JavaScript security policies and their enforcement mechanism within a web browser [46]. While some of these client-side approaches also rely on object capability patterns to isolate JavaScript, they focus on isolating completely untrusted code that is not essential to the application's core functionality (for instance, isolating advertisement code). In contrast, NodeSentry confines semitrusted code of libraries whose functionality is essential to the application. It counters relatively basic attacks where an attacker tries to exploit a *vulnerability* in a nonmalicious but buggy library included in the application, and it does so by making it simple to instrument entry and exit points of the library with security checks. As a consequence, the performance cost of NodeSentry can be significantly smaller than that of the existing client-side approaches.

Compared to the amount of work on securing JavaScript on the client-side, there is surprisingly little work on securing JavaScript on the server side. The conference version of this paper [5] was to the best of our knowledge the first security architecture addressing the secure inclusion of JavaScript libraries on the server side. Very recently, Staicu et al. proposed Synode [47], an automated mitigation technique for injection attacks on Node.js. In contrast with NodeSentry, Synode only protects against injection attacks on the `eval` and `exec` functions in Node.js, two functions that allow a developer to execute arbitrary code, by using static analysis to generate runtime checks. NodeSentry on the other hand supports hand-written policies for any module-exported function or variable. This allows a developer to use NodeSentry for writing policies for limiting library functionality even if they do not use `eval` or `exec` functions and are not vulnerable to injection attacks. A developer can thus arbitrarily modify functionality of a library without the limitation to fix only code injection vulnerabilities.

10. Conclusion

In order to address the problem of secure integration of third-party libraries, we have developed NodeSentry, a novel server-side JavaScript security architecture.

We have illustrated how 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. We have described the key features of the implementation of NodeSentry which builds on the implementation of membranes by Miller and Van Cutsem as a stepping stone for building trustworthy object proxies [23].

We evaluated the performance impact of NodeSentry in an experiment where a server must be able to provide files concurrently to an increasing number of clients. Our evaluation shows that the performance cost of the enforcement infrastructure itself is negligible and that useful policies can be enforced with very low performance overhead.

We evaluated the security effectiveness of NodeSentry by analyzing all 73 reported vulnerable libraries on the Node Security Project website, and we showed that the vast majority of these vulnerable libraries could be protected with NodeSentry.

Appendix

A table with a complete list of all the reported vulnerabilities of the Node Security Project as used within the evaluation in this document can be found in Table 5.

An in-depth discussion of our findings can be found in Section 7.2.

Data Availability

The NodeSentry prototype is available on GitHub (https://github.com/WillemDeGroef/nodesentry/).

Table 5

Package	Vulnerability	Category
hapi-auth-jwt2	Authentication Bypass	③
moment	Regular Expression Denial of Service	④
i18n-node-angular	Denial of Service	④
i18n-node-angular	Content Injection	①
hawk	Regular Expression Denial of Service	④
is-my-json-valid	Regular Expression Denial of Service	④
mqtt-packet	Denial of Service	④
mapbox.js	Content Injection	①
jshamcrest	Regular Expression Denial of Service	④
jadedown	Regular Expression Denial of Service	④
bittorrent-dht	Remote Memory Disclosure	⑤
ws	Remote Memory Disclosure	③
mysql	SQL Injection	①
hapi	Route level CORS config	②
ecstatic	Denial of Service	⑤
hapi	Denial of Service	①
mustache	Content Injection	①
handlebars	Content Injection	①
keystone	Authentication Weakness	③
milliseconds	Regular Expression Denial of Service	④
tar	Symlink Arbitrary File Overwrite	⑤
send	Root Path Disclosure	①
gm	Command Injection	①
ansi2html	Regular Expression Denial of Service	④
uglify-js	Regular Expression Denial of Service	④
secure-compare	Insecure Comparison	②
mapbox.js	Content Injection via TileJSON attribute	①
bleach	Regular Expression Denial of Service	④
ms	Regular Expression Denial of Service	④
hapi	Incorrect handling of CORS preflight request headers	③
ldapauth	LDAP Injection	①
datatables	Cross-Site Scripting	③
ldapauth-fork	LDAP Injection	①
ulgify-js	Incorrect non-boolean comparisons	②
ungit	Command injection	①
geddy	Directory traversal	①
semver	Regular Expression Denial of Service	④
jsonwebtoken	Verification Bypass	②
marked	Regular Expression Denial of Service	④
marked	VBScript Content Injection	①
sequelize	SQL Injection in Order	①
serve-static	Open Redirect	①
serve-index	XSS	③
inert	Hidden Directories Always Served	①
fancy-server	Directory Traversal	①
dns-sync	Command Injection	①
bassmaster	JavaScript Execution in Bassmaster	①
crumb	CORS Token Disclosure	①
express	No Charset In Content-Type Header	③
hapi	File Descriptor Leak Can Cause DoS Vulnerability	④
hapi	Rosetta-flash Jsonp Vulnerability	③
libyaml	Heap-based Buffer Overflow When Parsing YAML Tags	⑤

Table 5: Continued.

Package	Vulnerability	Category
marked	Multiple Content Injection Vulnerabilities	①
nhouston	Directory Traversal	①
paypal-ipn	Validation Bypass	②
printer	Potential Command Injection on Untrusted Input	①
qs	Denial-of-Service Extended Event Loop Blocking	④
qs	Denial-of-Service Memory Exhaustion	④
remarkable	Content Injection	③
send	Directory Traversal	①
st	Directory Traversal	①
syntax-error	Potential For Script Injection	①
validator	isURL Regular Expression Denial of Service	④
validator	XSS Filter Bypass via Encoded URL	③
yar	Denial-of-Service	④
js-yaml	Deserialization Code Execution	②
hubot-scripts	Scripts Potential Command Injection in Email.coffee	①
tomato	API Admin Auth Weakness	②
codem-transcode	Potential Command Injection in Ffprobe Functionality	①
ep_imageconvert	Unauthenticated Remote Command Injection	①
libnotify	Command Injection in Libnotify.notify	①
connect	Middleware Reflected Cross-Site Scripting	③
validator	XSS Filter Bypasses	③

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work has been partly supported by the EU-FP7-NESSOS project and by the FWO-SBO Tearless project.

References

[1] Stackoverflow developer survey results 2016. http://stackoverflow.com/research/developer-survey-2016.

[2] D. Flanagan, *JavaScript: The Definitive Guide*, O'Reilly Media, Inc., 6th edition, 2011.

[3] S. Lekies, B. Stock, and M. Johns, "25 Million Flows Later Large-scale Detection of DOM-based XSS," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[4] N. Nikiforakis, L. Invernizzi, A. Kapravelos et al., "You are what you include: large-scale evaluation of remote Javascript inclusions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 736–747, ACM, Raleigh, NC, USA, October 2012.

[5] W. De Groef, F. Massacci, and F. Piessens, "NodeSentry: Least-privilege library integration for server-side JavaScript," in *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014*, pp. 446–455, USA, December 2014.

[6] W. De Groef, *Client- and Server-Side Security Technologies for JavaScript Web Applications [Ph.D. thesis]*, University of Leuven, 2016.

[7] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[8] A. Ojamaa and K. Düüna, "Assessing the security of Node.js platform," in *Proceedings of the 7th International Conference for Internet Technology and Secured Transactions, ICITST 2012*, pp. 348–355, UK, December 2012.

[9] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation, PLDI 2010*, pp. 1–12, ACM, June 2010.

[10] B. Sullivan, *Server-Side JavaScript Injection: Attacking and Defending NoSQL and Node.js. In Black Hat USA*, 2011.

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

[12] P. A. Gardner, S. Maffeis, and G. D. Smith, "Towards a program logic for JavaScript," in *Proceedings of the the 39th annual ACM SIGPLAN-SIGACT symposium*, p. 31, Philadelphia, PA, USA, January 2012.

[13] M. Fredrikson, R. Joiner, S. Jha et al., "Efficient runtime policy enforcement using counterexample-guided abstraction refinement," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2012.

[14] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[15] U. Erlingsson, *The Inlined Reference Monitor Approach to Security Policy Enforcement [Ph.D. thesis]*, Cornell University, 2004.

[16] L. Desmet, W. Joosen, F. Massacci et al., "Security-by-contract on the .NET platform," *Information Security Technical Report*, vol. 13, no. 1, pp. 25–32, 2008.

[17] L. Desmet, W. Joosen, F. Massacci et al., "A flexible security architecture to support third-party applications on mobile devices," in *Proceedings of the ACM workshop on Computer security architecture*, pp. 19–28, ACM, USA, November 2007.

[18] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proceedings of the 31st IEEE Symposium on Security and Privacy, SP 2010*, pp. 109–124, USA, May 2010.

[19] N. Bielova, D. Devriese, F. Massacci, and F. Piessens, "Reactive non-interference for a browser model," in *Proceedings of the 2011 5th International Conference on Network and System Security, NSS 2011*, pp. 97–104, Italy, September 2011.

[20] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Secure multi-execution of web scripts: Theory and practice," *Journal of Computer Security*, vol. 22, no. 4, pp. 469–509, 2014.

[21] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: Enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 2–16, 2005.

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

[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)*, pp. 154–178, 2013.

[24] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," Rfc 6797, 2012, http://tools.ietf.org/html/rfc6797.

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

[26] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International World Wide Web Conference (WWW '10)*, pp. 921–929, Raleigh, NC, USA, April 2010.

[27] P. B. Kruchten, "Architectural Blueprints - The "4+1" view model of software architecture," *Journal of IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.

[28] M. Keil and P. Thiemann, "TreatJS: Higher-order contracts for JavaScript," https://arxiv.org/abs/1504.08110.

[29] T. Heyman, D. Preuveneers, and W. Joosen, "Scalar: Systematic scalability analysis with the universal scalability law," in *Proceedings of the 2nd International Conference on Future Internet of Things and Cloud, FiCloud 2014*, pp. 497–504, Spain, August 2014.

[30] L. Koved, M. Pistoia, and A. Kershenbaum, "Access rights analysis for Java," in *Proceedings of 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '02*, pp. 359–372, ACM, New York, NY, USA, 2002.

[31] E. Geay, M. Pistoia, T. Takaaki, B. G. Ryder, and J. Dolby, "Modular string-sensitive permission analysis with demand-driven precision," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pp. 177–187, IEEE Computer Society, Vancouver, BC, Canada, May 2009.

[32] B. Hermann, M. Reif, M. Eichberg, and M. Mezini, "Getting to know you: Towards a capability model for Java," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015*, pp. 758–769, ACM, New York, NY, USA, September 2015.

[33] G. Kiczales, J. Lamping, A. Mendhekar et al., "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*, M. Aksit and S. Matsuoka, Eds., Lecture Notes in Computer Science, pp. 220–242, Springer, Berlin, Heidelberg, 1997.

[34] B. De Win, W. Joosen, and F. Piessens, *Developing Secure Applications through Aspect-Oriented Programming*, vol. 10, Addison-Wesley, 2004.

[35] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja, Safe active content in sanitized javascript, 2008," http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[36] S. Maffeis, J. C. Mitchell, and A. Taly, "Isolating JavaScript with filters, rewriting, and wrappers," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pp. 505–522.

[37] S. Maffeis, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *Proceedings of the 31st IEEE Symposium on Security and Privacy, SP 2010*, pp. 125–140, USA, May 2010.

[38] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy, EURO S and P 2016*, pp. 147–162, Germany, March 2016.

[39] D. Swasey, D. Garg, and D. Dreyer, "Robust and compositional verification of object capability patterns," *Proceedings of the ACM on Programming Languages, 1(OOPSLA)*, vol. 89, pp. 1–26, 2017.

[40] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting JavaScript," in *Proceedings of the 4th International Symposium on ACM Symposium on Information, Computer and Communications Security, ASIACCS'09*, pp. 47–60, Australia, March 2009.

[41] 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 28th Annual Computer Security Applications Conference (ACSAC '12)*, pp. 1–10, ACM, December 2012.

[42] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "FlowFox: A web browser with flexible and precise information flow control," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012*, pp. 748–759, USA, October 2012.

[43] D. Hedin and A. Sabelfeld, "Information-flow security for a core of JavaScript," in *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium, CSF 2012*, pp. 3–18, USA, June 2012.

[44] L. A. Meyerovich and B. Livshits, "CONSCRIPT: Specifying and enforcing fine-grained security policies for JavaScript in the browser," in *Proceedings of the 31st IEEE Symposium on Security and Privacy, SP 2010*, pp. 481–496, USA, May 2010.

[45] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga, "Control-Flow Integrity in Web Applications," in *Engineering Secure Software and Systems*, vol. 7781 of *Lecture Notes in Computer Science*, pp. 1–16, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[46] N. Bielova, "Survey on JavaScript security policies and their enforcement mechanisms in a web browser," *Journal of Logic and Algebraic Programming*, vol. 82, no. 8, pp. 243–262, 2013.

[47] C. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js," in *Network and Distributed System Security (NDSS)*, 2018.