NodeSentry: Least-privilege Library Integration for Server-Side JavaScript

Willem De Groef
iMinds–DistriNet
KU Leuven, Belgium
Willem.DeGroef@cs.kuleuven.be

Fabio Massacci
University of Trento
Fabio.Massacci@unitn.it

Frank Piessens
iMinds–DistriNet
KU Leuven, Belgium
Frank.Piessens@cs.kuleuven.be

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 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 implementation of NodeSentry, and present its practical evaluation. For hundreds of concurrent clients, NodeSentry has the same capacity and throughput as plain Node.js. Only on a large scale, when Node.js itself yields to a heavy load, NodeSentry shows a limited overhead.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services

Keywords

Web security, JavaScript

1. INTRODUCTION

Services offered on the web have a standard conceptual architecture: a client (or tenant) accesses a web application which talks to one or more databases [8]. In order to serve multiple clients, the traditional approach (represented by e.g., Apache and IIS) has been to duplicate the entire path for each client at the process level. In order to cope with increasing demands, modern services (e.g., Salesforce, SAP-ByDesign) have evolved to multi-tenancy event-driven architecture: different tenants access the same pipe which takes care of the different events by an event-driven program [41].

The major reason behind the success of event-driven programs is that they offer developers a much finer control (and therefore better performance) than switching between application processes [41, 17]. Among the various event-driven programming languages, Node.js is a widely successful platform that combines the popular JavaScript language with an efficient runtime tailored for a cloud-based event architecture [34].

JavaScript has many advantages for web development [14]. It is the de facto dominant language for client-side applications and it offers the flexibility of dynamic languages. In particular it allows the easy combination or mash-up of content and libraries from disparate third parties. Such flexibility comes at a price of significant security problems [26, 33], and researchers have proposed a number of solutions to contain them: from sandboxing (e.g., Google's Caja or [35, 2]) and information flow control [9, 10] to instrumenting the client with a number of policies [30], or trying to guarantee control-flow integrity at a web-firewall level [6]. Bielova presents a good recent survey on JavaScript security policies and their enforcement mechanism within a web browser [4]. These proposals are appropriate for client-side JavaScript but cannot be lifted to server-side code. At first, they assume that the client is not running with high-privileges; second they command a significant overhead acceptable at client side but not at server side. For example, Meyerovich's et al., [30] report some of the best micro-benchmarks for client side JavaScript and still report an overhead between 24% to 300% of the raw time.

Security problems are magnified at server side: 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. Figure 2 shows the libraries integrated in one of the most popular web application servers based on Node.js. Verifying such a massive amount of third-party code, especially in a

ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664276

language as dynamic and flexible as JavaScript, is close to impossible [43, §6].

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 paper targets this gap.

1.1 Contributions

This paper proposes a solution to the problem of least-privilege integration of libraries with the following contributions:

1. NodeSentry, a novel server-side JavaScript security architecture;
2. Policy infrastructure that allows 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 NodeSentry's implementation and its policy infrastructure in Node.js;
4. Practical evaluation of the performance of our solution.

In summary we show that for hundreds of concurrent clients NodeSentry is essentially close to its theoretical optimum, between 250-500 concurrent clients NodeSentry exhibits an increasing drop in capacity and after 500 move in synch with Node.js's own drop in performance reaching 50% of the theoretical optimum (while Node.js is at 60%).

The rest of this paper is structured as follows. Section 2 sketches the necessary background on Node.js and the security problems of its ecosystem of third-party libraries. Section 3 describes the exact threat model and gives a general overview of our solution, called NodeSentry. Section 4 discusses how NodeSentry can be used in practice and how it protects against real-life attacks. In Section 5, we exemplify several real-life policies and Section 6 gives insight into the implementation. Section 7 discusses the quantitative evaluation of the performance. Finally, Section 8 discusses related work, and Section 9 summarizes the contributions.

2. BACKGROUND ON Node.js LIBRARIES

Node.js by itself only provides core system functionality like e.g., accessing the file system or network communication. Developers that want to build applications must therefor often rely on third-party libraries. They are distributed as packages, structured according to the CommonJS package format and installable via the de facto standard `"npm"` package manager (by itself a JavaScript package). The official package registry contains more than 70 thousand packages and has more then 290 million downloads each month. Such libraries are statically or dynamically loaded in order to provide the corresponding services.

Node.js module loading system is very easy to use. Via the built-in `require` function, modules living within the base system, in a separate file or directory, can be included in the application. The 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. On line 2 of

```
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') }
```

Figure 1: Code excerpt that shows how different system functionalities are exposed within the Node.js environment by requiring specific libraries.

Figure 1, the variable `path` will be an object with several properties including `path.sep` that represents the separator character or 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 string 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 a `require(lib)` function where `lib` is dynamically evaluated.

The resulting ecosystem is such that almost all applications are composed of a large numbers 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.

Figure 2 shows a bird's eye view of the library used by the `"npm-www"` JavaScript package maintainer. One of the single nodes of this package tree, is the sub library `"st"` (the fourth node from the left) which is developed specifically to manage static files hosting for the backend of the web site [1]. As you can see, the `"st"` library further relies on access to the `"http"` and `"url"` package to process URLs and on the `"fs"` package to access the file system.

The quote below from a blog post of a Node.js developer clearly explains the sharing principles of the Node.js ecosystem[2]:

> I'm working on my own project, and was looking for a good static serving library. I found the best one, but sadly it was melded tightly to the npm-www project... glad to see it extracted and modularized!

Unfortunately, the resulting `"st"` turned out to be vulnerable to a directory traversal bug[3] which allowed it to serve essentially all files on the server, and thus leading to a potential massive compromise of all activities.

How can one check libraries for potential vulnerabilities? Server-side JavaScript code is not subject to changes as client-side code, so one may hope that static analysis might work.

[1] http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers
[2] https://github.com/isaacs/st/issues/3
[3] https://nodesecurity.io/advisories/st_directory_traversal & http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3744

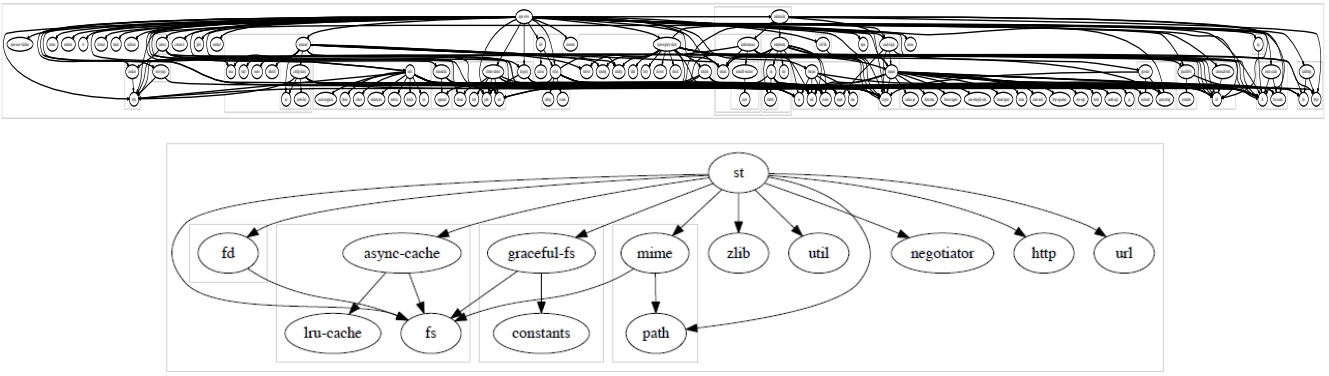

Figure 2: The code that runs http://npmjs.org, which is a Node.js package itself (top image), loads a large number of third parties libraries (which may use further libraries). The fourth node from left is the `"st"` library which further uses additional libraries (bottom image). Static verification is close to impossible.

Unfortunately, the dynamic functionalities and the usage of exceptions alone make static analysis of JavaScript packages extremely difficult: only a handful of frameworks for static analysis can deal with exceptions and dynamic calls [18, 16]. Further, the large quantity of libraries to be considered (and modeled) is another major hurdle. For example JAM requires modeling such dependencies in Prolog [15]. Runtime monitoring seems the only alternative *if* it can scale up to hundreds or thousands of concurrent requests. For client-side JavaScript, for *one* client, an effective implementation like ConScript already tallies a minimum 25% up to 300% overhead.

3. THREAT MODEL AND SOLUTION

The server-side scenario, discussed earlier, 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)*, as for example the `"st"` library. They might end up using malicious objects or doing something they were not intended to do.

The purpose of our security model is to shield the potential untrusted 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 the policies for *all* other possible libraries to be isolated from the malicious one by far outweigh the effort of writing the alleged benign functionalities 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. 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.

The key idea of our proposal is to use a variant of an inline reference monitor [38, 13] as modified for the Security-by-Contract approach for Java and .NET [11] in order to make it more flexible. Namely, we do not embed the monitor into the code as suggest by most approaches for inline reference monitors but inline only the hooks in a few key places, while the monitor 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 which do not call security relevant actions are unaffected.

Further, and maybe most importantly, we do not limit ourselves to purely raising security exceptions and stopping the execution but support policies that specify how to "fix" the execution [12, 5, 10]. This is another essential requirement for server side applications which must keep going.

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 [31, §9] and further refined in [44]. The goal of a membrane is to fully isolate two object graphs [31, 44]. 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.

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 [44], it instantly achieves the goal of transitively revoking all references as advocated by Miller [31].

The NodeSentry-handler intercepts the object references received by the semi-trusted library and can check them for compliance with the policy. Our policy decision point can be seen as a simple automaton: if the handler receives a request for an action and can make the transition then the object proxied by the membrane is called and the (proxied) result is returned; if the automaton cannot make a transition on the input (i.e., the policy is violated), then a security countermeasure can be implemented by NodeSentry or, in the worst case scenario, a security exception will be automatically raised.

We have identified *two* possible points where the policy

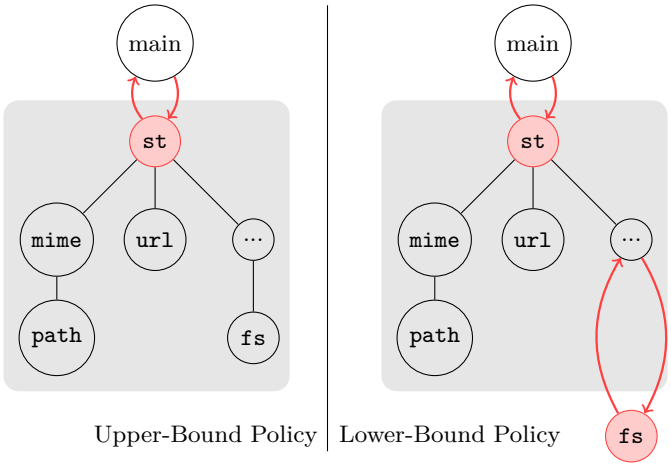

Figure 3: NodeSentry allows policies to be installed on the public interface of the secure library (*Upper-Bound policies*) and/or on the public interface of any depending library (*Lower-Bound policies*).

hooks can be placed that fall together 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. The choice of the location determines two type of policies:

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 it. 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.

Lower-Bound policies can be installed on the public interface of any depending library, both built-in core libraries (like e.g., `"fs"`) or 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 3 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).

A developer wishing to use NodeSentry only needs to replace the `require` call to the semi-trusted library with a `safe_require`. This approach makes it possible to implement

```
1  // code snippet from st.js
2  // get a path from a url
3  Mount.prototype.getPath = function (u) {
4      u = path.normalize(url.parse(u).pathname
5              .replace(/^[\/\\]?/, '/'))
6              .replace(/\\/g, '/')
7  // ...
```

Figure 4: The `"st"` library has a potential security issue because it does not check the file path for potential directory traversal.

a number of security checks used for web-hardening , like e.g., enabling the HTTP Strict-Transport-Security header [20], set the Secure and/or HttpOnly Cookies flags [3] or configure a Content Security Policy (CSP) [39], in quite a modular way without affecting the work of rank-and-file JavaScript developers. This is described in the next section whereas we illustrate some policy examples more in detail in Section 5.

4. USAGE MODEL

Here we describe the usage model [24] of the NodeSentry library. The developer (such as the one whose blog entry we have cited) has found an appropriate library for her application.

She may now use the library to serve files to clients. As mentioned, the library has a potential potential directory traversal issue, as shown in Figure 4. By itself, this may *not* be a vulnerability: if a library provides a functionality to manage 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 below becomes a serious security vulnerability.

An HTTP request for `/%2e%2e/%2e%2e/etc/passwd`, sent by an attacker towards a server using the `"st"` library to serve files, could expose unintended files.

It is of course possible to modify the original 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 go fix it without breaking other dependent libraries, or just have other priorities altogether.

In all these scenarios, which are the majority of the cases, the application of NodeSentry is the envisaged solution. The `"st"` library is considered semi-trusted and a number of default web-hardening policies are available in the NodeSentry policy toolkit.

The only adjustment is to load the NodeSentry framework and to make sure that `"st"` is safely required so that the policy becomes active, as shown in Figure 5.

The policy rules in Figure 6 can then be activated in the policy section and 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 `IncomingMessage.url`), we first test it on the presence of a (encoded) dot character. If so, we return a new URL that e.g., points to a file that contains a warning message.

```
require("nodesentry");
var http = require("http");
var st= safe_require("st");
var handler = st(process.cwd());
http.createServer(handler).listen(1337);
```

Figure 5: After loading the NodeSentry framework, policies can be (recursively) enforced on libraries by loading them via the newly introduced `safe_require` function.

```
if (method === "IncomingMessage.url") {
    var regex = new RegExp(/%2e/ig);
    if (regex.test(origValue))
        return "/your_attack_is_detected.html";
    else
        return origValue;
}
```

Figure 6: If application code requests the URL of the incoming request, a pointer to a different page is returned whenever malicious characters are detected.

```
if (method === "ServerResponse.write")  {
    var h = "Strict-Transport-Security";
    var v = "max-age=3600; includeSubDomains";
    response.setHeader(h, v);
    // move on with the real
    // ServerResponse.write call
}
```

Figure 7: Before a server response is sent towards a client, the policy first adds the HSTS header, effectively mimicking the behaviour of `helmet.hsts()`.

```
if (method === "fs.writeFile" ||
    method === "fs.write" ||
    method === "fs.writeFileSync" ||
    method === "fs.writeSync" ||
    method === "fs.appendFile" ||
    method === "fs.appendFileSync")  {
    // simply return
    return
}
```

Figure 8: A possible policy that wants to prevent a library from writing to the file system must cover all availalbe write operations of the `"fs"` library.

5. POLICY EXAMPLES

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, a single large monolithic policy being 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 implements OWASP's check on file system management (e.g., escaping, file traversal etc.). 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 and thus avoid paying an unnecessary performance price.

As a simple example for the potential of NodeSentry we describe how we implemented the checks behind the 'helmet' library[4], a middleware used for web hardening and implementing various security headers for the popular `"express"` framework.

It is used to, e.g., enable the HTTP Strict Transport Security (HSTS) protocol [20] in an `"express"`-based web application by requiring each application to actually use the library when crafting HTTP requests. Figure 7 shows a NodeSentry policy that adds the HSTS header before sending the outgoing server response.

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

The example in Figure 8 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 so that the real method call never gets executed, and thus effectively prevent writing to the file system. It is possible to change this behavior by e.g., throwing an exception or *chrooting* to a specific directory.

6. IMPLEMENTATION DETAILS

This section reports on our development of a mature NodeSentry prototype, which is designed to work with the latest Node.js versions and relies on the upcoming ES Harmony JavaScript standard. Membranes require this standard, in order to implement fully transparent wrappers, and 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. The main goal of wrapping a library's public API with a membrane, is to be sure that each time an API is accessed, our enforcement mechanism is invoked in a secure and transparent manner.

We rely on the ES Harmony reflection module shim by Van Cutsem[5] and its implementation of a generic membrane abstraction, which is used as a building block of our implementation and is shown in Figure 9. The current prototype runs seamlessly on Node.js v0.10.

Our first stepping stone is to introduce the `safe_require` function (see Figure 10) that virtualizes the `require` function so that any additional library, called within the membrane, can be intercepted by the framework.

This 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

[4]https://github.com/evilpacket/helmet

[5]https://github.com/tvcutsem/harmony-reflect

```
function newMembrane (ifaceObj, policyObj) {
    return require("membrane")
        .makeGenericMembrane(ifaceObj, policyObj)
        .target;
}
```

Figure 9: 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.

```
1  function safe_require (libName) {
2      var loadLib = function () {
3          var mod = new Module(libName);
4          var customRequire = membranedRequire(ctxt);
5
6          mod.require = function (libName) {
7              return customRequire(libName); };
8
9          return mod.loadLibrary();
10     };
11     return newMembrane(loadLib().exports, policy);
12 }
```

Figure 10: 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.

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.

Line 11 of Figure 10 shows how the public interface object (exports), gets membraned with a given policy. This line makes it possible to enforce Upper-Bound policies.

Lower-Bound policies can be enforced because of the custom require function that is given to the context in which a library gets loaded (line 4 in Figure 10). Because we provide the context with our own require function, we can intercept all its calls from any depending library. At interception time, we can decide if it is necessary to membrane the public interface object of that depending library (line 8 of Figure 11). 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 (line 11 of Figure 11).

7. EVALUATION

Performance is king for server-side JavaScript and the main goal of our benchmarking experiment is to verify the impact of introducing NodeSentry on the two major performance drivers. We define performance as *throughput*, i.e., the amount of tasks or total requests handled by our server, or as *capacity*, i.e., the total amount of concurrent users/requests handled by our server. These are standard measures for high performance concurrent servers [19].

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 requests. The *entire* code of the server, besides the libraries

```
function membranedRequire (lib) {
    return function (lib) {
        var libexports;

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

        if (lowerBoundPolicyNeeds(lib)) {
            return newMembrane(libexports, policy);
        } else {
            return libexports;
        }
    }
}
```

Figure 11: In order to enforce a (Lower-Bound) policy between a library and a depending library, its interface object must be wrapped within a membrane.

```
// set to false for plain Node.js
var enable_nodesentry = true;

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

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

var handler = st(process.cwd());
http.createServer(handler).listen(1337);
```

Figure 12: The streamlined benchmark application implements a bare static file hosting server, by using the popular "st" and "http" libraries.

"http" and "st", is shown in Figure 12. 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 with NodeSentry enabled.

Each experiment (for plain Node.js and for Node.js with NodeSentry enabled) 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 13. 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), after few seconds the number of clients step up and quickly reaches the total number N (ramp-up phase). The number of clients then remains constant until the end of the experiment (peak phase) with N clients continuously sending concurrent requests for files.

The experimental setup consists of two identical machines[6] interconnected in a switched gigabit Ethernet network. One machine is responsible for generating HTTP requests by spawning multiple threads, representing individual users. The second machine runs Node.js v0.10.28 and acts as the server.

[6]Each machine has 32 Intel© Xeon™ CPUs ES-2650 and 64GB RAM, running Ubuntu 12.04.4 LTS.

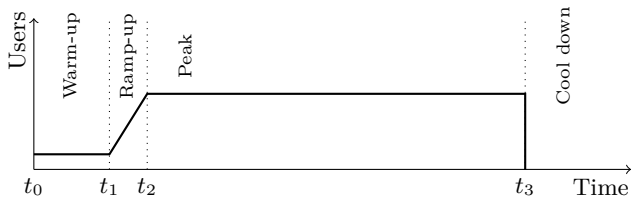

Figure 13: In our experimental set-up, the load profile of the experiment varies between a minimum (the warm-up phase) and a maximum (the peak phase) of concurrent users. This is repeated for $N = 1..1000$ concurrent users sending requests to our server.

The results of the experiment are summarized in Figure 14. The left graphics reports the throughput: how many requests the system is able to concurrently serve as the number of clients increases. This value is represented on the y-axis while the number of clients is represented on the x-axis. The diagonal black line plots the theoretical maximum: all requests by all clients are served in the given time horizon. Each blue square represent the summary of the performance of pure Node.js for the corresponding number of clients. The red circles denote the performance of NodeSentry for the same number of clients. The solid lines shows the interpolation curve with the *glm* method in R with a polynomial of grade 2. The gray shaded area represent the 95%-confidence interval computed by the function.

The right graphics reports the *capacity*: the number of concurrent requests handled at each time instance. The coding of lines and data follows the same criteria as for throughput: the blue squares and the blue line represents Node.js data points and interpolated values whereas the red line and the red circles represent the data points for NodeSentry.

For the first 200-250 all systems are able to serve requests at essentially the theoretical maximum capacity of the local benchmarking system. The system can comfortably host the intended amount of threads/concurrent users without slow-down. The results in Figure 14 indicate that NodeSentry's loss in capacity starts from around 200-250 concurrent users whereas the capacity of a plain Node.js instance starts to degrade at around 500 concurrent users.

NodeSentry gradually loses capacity until it stabilizes at approx 40% loss over the plain Node.js capacity and then moves in synchrony with NodeSentry after 500 users. It starts gaining again after approx 800-900 users and reduces the gap to 10%. Therefore, we can conclude that after 500 the losses of capacity are no longer due to NodeSentry but are directly consequence of the loss of capacity of Node.js. The sprint-up at 1000 clients can be easily explained: the main Node.js system is strained to keep up with performance, it has lost already 40% of its capacity over the theoretical maximum. In such stressful conditions, the additional constraints posed by NodeSentry's policy monitor are a drop in the sea.

We do not report data beyond 1000 users (albeit we tested it) because the behavior of *plain* Node.js started to exhibit significant jitters. It showed that largely beyond 1000 the actual capacity of our system set-up was dominated by other factors (OS process swaps, network processes, caches, etc. etc.). Setting up a benchmarking system that can smoothly process 10.000 users and beyond is an interesting direction for future work.

We've also measured the impact on the capacity of a server between using only policy hook (`"fs"` inside the membrane) and two policy hooks (`"fs"` outside the membrane).The results shown in Figure 15 indicate that there is no significant loss of capacity by bounding the semi-trusted library at the different policy points and thus tightening the policy rules.

At first, we stress again that *up to 200 clients there is no difference in performance*, which brings us almost at the same level of performance for an industrial security events monitoring system, suitable for deployment at a small business [23]. This strikingly compares with traditional approaches for JavaScript client side security in which even for *one* client there can be a performance penalty up to 300%.

For a larger number of clients there is a trade-off between performance and security. Such trade-off is still limited (less than 50% overhead) and decreases when other conditions stretch the performance of the system. Just as in normal program code, developers must take care to write efficient policy code. However, since policy code is written in plain JavaScript, it can benefit from efficiency measures in the underlying JavaScript engine, like e.g., a JIT compiler. Further, we believe that there are at least three ways to optimize the performance. The overhead is mostly due to the peculiarities of membranes: the overhead cost of the actual invariant enforcement mechanism, e.g., its use of a *shadow object*, the run-time post-condition assertions of the trap functions of the membrane handler, and the reliance on a self-hosted implementation of *Direct Proxies* in JavaScript [44, §5&6]. These would require a significant engineering effort that would not be justified for a research implementation.

8. RELATED WORK

There is a large body of work on JavaScript security, but the main focus has been overwhelmingly on client-side security. A very comprehensive survey of many of the recent works has been provided by Bielova [4] who describes a variety of JavaScript security policies and their enforcement mechanism within a web browser context. Therefore we refer to her work for additional details and only focus here on the few works that are closest to our own contribution.

JavaScript security in general.

Restricting third-party components within a web browser or web application by mediating access to specific security-sensitive operations, has seen a lot of attention since its rise the last decade.

BrowserShield [36] is a server-side rewriting system that rewrites certain JavaScript functions to use safe equivalents. These safe equivalents are injected into each web page via the BrowserShield JavaScript libraries. BrowserShield makes use of proxies for injecting its code into a web page. Self-protecting JavaScript [28, 35] is a client-side wrapping technique that applies an advice around JavaScript functions. The wrapping code with its advices are provided by the server and executed first, to make sure to operate in a clean, non-tampered environment. Browser-Enforced Embedded Policies (BEEP) [22] is a server system that injects a policy in a web page. The browser will call this policy script before loading any another script, giving the policy the opportunity

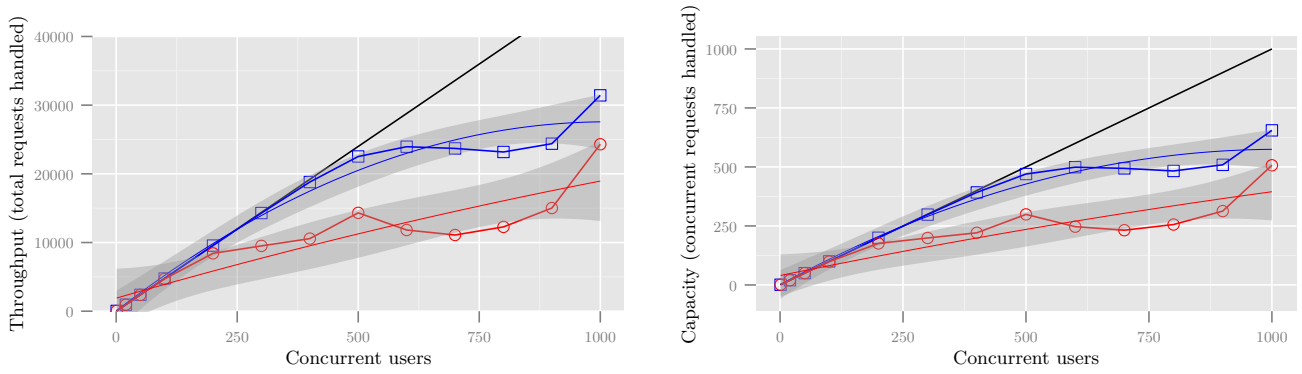

Figure 14: The solid black line is the theoretical performance of concurrent requests served in the fixed time horizon. The red circles represent the actual performance of plain Node.js with NodeSentry; the blue squares the performance of pure Node.js. Up to 200 clients the performance is optimal. Between 500-1000 we have a slight drop that is anyhow below 50%.

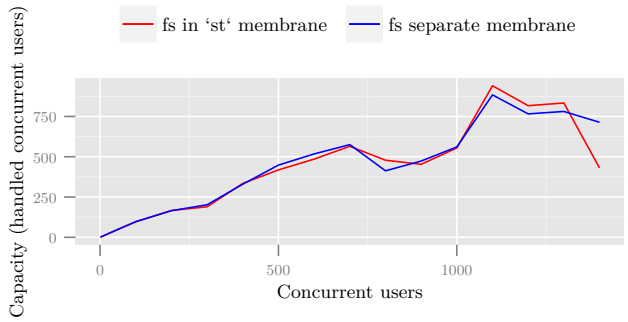

Figure 15: Tightening security by adding both an Upper-Bound policy and a Lower-Bound policy does not affect capacity, as demonstrated with the comparison of `"fs"` inside or outside the `"st"` membrane.

to vet the script about to be loaded. The loading process will only continue after approval of the policy. ConScript [29] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript [28, 35], except that ConScript uses deep advice, thus protects all access paths to a function. The price for using deep advice is the need for client-side support in the JavaScript engine. WebJail [42] offers the integrator the possibility to define a policy, in a specific, non-JavaScript way, that restricts the behavior of a third-party component in an isolated way. Agten et al., [2] present JSand, a server-driven sandboxing framework to enforce server-specified security policies in a client's browser. Richards et al. [37] present a security infrastructure for dealing with the gadget attacker threat model, by allowing the specification of access control policies on parts of a JavaScript program via leveraging the concept of delimited histories with revocation. Fredrikson et al., [15] have developed an off-line mechanism for the analysis of JavaScript applications that identify the place where policy hooks can be implemented by ConScript, thereby relying heavily on model checking technologies. Stefan et al. [40] introduce COWL, a JavaScript confinement system based on a new label-based mandatory access control API within web browsers that can be used by developers to indicate how to restrict the communication between compartments and external servers.

Security platforms for managed code.

Livshits [27] provides a taxonomy of runtime taint tracking approaches, in order to preventing web application vulnerabilities such as cross-site scripting and SQL injection attacks.

Wei et al., [45] propose a new architecture that decomposes a web service into two parts, executing in a separate protection domain. Only the trusted part can handle security-sensitive data.

Burket et al., [7] developed GuardRails, a source-to-source tool for building secure Ruby on Rails web applications, by attaching security policies, via annotations, to the data model itself. GuardRails produces a modified application that automatically enforces the specified policies.

Hosek et al., [21] developed a Ruby-based middleware that (1) associates security labels with data and (2) performs transparent label tracking, across a multi-tier web architecture in order to prevent harmful data disclosure.

Nguyen-Tuong et al., [32] propose a fully automated approach to harden PHP-based web applications via precise taint tracking of data and checking specifically for dangerous content only in parts of commands and output that came from untrustworthy sources.

Web application firewalls (WAF).

Krueger et al., [25] describe a technique, based on anomaly detectors, that replace suspicious parts in HTTP requests by benign data.

ModSecurity [1] is a firewall that detects malicious behavior by pattern matching HTTP requests with an existent rule base. A similar proxy-based approach has been proposed by Braun et al., [6] who implemented a policy enforcement mechanism to guarantee the control flow integrity of web applications.

9. CONCLUSIONS

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

Yet, these very 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 have developed NodeSentry, a novel server-side JavaScript security architecture that supports such least-privilege integration of libraries.

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 [44].

In order to show the practical effectiveness of NodeSentry we have evaluated its performance in an experiment where a server must be able to provide files concurrently to an increasing number of clients up to thousands of clients and tens of thousands of file requests. Our evaluation shows that for up to 250 clients NodeSentry has the same server capacity and throughput of plain Node.js, and that such capacity is essentially the theoretical optimum. At 1000 concurrent clients in a handful of seconds, when plain Node.js's already dropped capacity barely above 60% of the theoretical optimum, NodeSentry is able to attests itself at 50%.

Our complete prototype implementation (including the full source code, test suites, code documentation, installation/usage instructions, and the `"st"` example) is available at https://distrinet.cs.kuleuven.be/software/NodeSentry/ or directly installable via `npm install nodesentry`.

Acknowledgments

We thank our shepherd Joshua Schiffman and the anonymous reviewers for their valuable feedback. This research is partially funded by the Research Fund KU Leuven, the EU-funded FP7 projects NESSoS and WebSand, the IWT-SBO project SPION and the Italian Project PRIN-MIUR-TENACE. With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission – Directorate-General Home Affairs. This publication reflects the views only of the authors, and the funders cannot be held responsible for any use which may be made of the information contained therein.

10. REFERENCES

[1] Modsecurity – the open source web application firewall. https://www.modsecurity.org/.

[2] 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.

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

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

[5] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the International Conference on Network and System Security (NSS)*, pages 97–104, 2011.

[6] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga. Control-flow integrity in web applications. In *Engineering Secure Software and Systems (ESSOS'13)*, pages 1–16. Springer, 2013.

[7] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. GuardRails: A Data-Centric Web Application Security Framework. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2011.

[8] F. Chong and G. Carraro. Architecture strategies for catching the long tail. Technical report, Microsoft Corporation, April 2006. Available on the web at http://msdn.microsoft.com/en-us/library/aa479069.asp.

[9] 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 ACM Conference on Computer and Communications Security (CCS)*, pages 748–759, 2012.

[10] 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.

[11] 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.

[12] D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 109–124, 2010.

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

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

[15] 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.

[16] 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.

[17] L. Griffin, B. Butler, E. de Leastar, B. Jennings, and D. Botvich. On the Performance of Access Control Policy Evaluation. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 25–32, 2012.

[18] 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.

[19] N. J. Gunther. *Guerrilla capacity planning – a tactical approach to planning for highly scalable applications and services.* Springer, 2007.

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

[21] P. Hosek, M. Migliavacca, I. Papagiannis, D. M. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch. SafeWeb: A Middleware for Securing Ruby-based Web Applications. In *Proceedings of the International Middleware Conference*, pages 480–499, 2011.

[22] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 601–610, May 2007.

[23] K. M. Kavanagh, M. Nicolett, and O. Rochford. Magic Quadrant for Security Information and Event Management. http://www.gartner.com/technology/reprints.do?id=1-1W1N1U4&ct=140627, June 2014.

[24] P. B. Kruchten. Architectural Blueprints – The "4+1" View Model of Software Architecture. *Journal of IEEE Software*, 12(6):42–50, 1995.

[25] T. Krueger, C. Gehl, K. Rieck, and P. Laskov. TokDoc: A Self-Healing Web Application Firewall. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 1846–1853, 2010.

[26] 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.

[27] B. Livshits. Dynamic Taint Tracking in Managed Runtimes. Technical Report MSR-TR-2012-114, Microsoft Research, 2012.

[28] J. Magazinius, A. Askarov, and A. Sabelfeld. A Lattice-based Approach to Mashup Security. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 15–23, 2010.

[29] L. Meyerovich, A. Felt, and M. Miller. Object views: Fine-grained sharing in browsers. In *Proceedings of the 19th international conference on World wide web*, pages 721–730. ACM, 2010.

[30] 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.

[31] 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.

[32] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, pages 372–382, 2005.

[33] 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.

[34] 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.

[35] 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.

[36] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)*, 1(11), September 2007.

[37] 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.

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

[39] 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.

[40] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, H. Dave, K. Brad, and D. Mazieres. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[41] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[42] 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.

[43] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[44] 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.

[45] J. Wei, L. Singaravelu, and C. Pu. A Secure Information Flow Architecture for Web Service Platforms. *IEEE Transactions on Services Computing*, 1(2):75–87, 2008.