Brigadier: A Datalog-based IAST framework for Node.js Applications

Angel Luis Scull Pupo
Vrije Universiteit Brussel
angel.luis.scull.pupo@vub.be

Jens Nicolay
Vrije Universiteit Brussel
jens.nicolay@vub.be

Elisa Gonzalez Boix
Vrije Universiteit Brussel
egonzale@vub.be

***Abstract*—The NODE.JS runtime, in combination with Node Package Manager (NPM), is a popular ecosystem for building server-side web applications. Both JavaScript's flexible and dynamic character and the vast amount of NPM libraries available can speed up the development of web applications. However, JavaScript and NODE.JS lack security mechanisms and abstractions. Despite the numerous language-based approaches proposed to protect JavaScript applications, no work supports application-level and business-level security properties. This means that in order to achieve both application-level and business-level security, developers are forced to rely on multiple different, unintegrated and incompatible tools and mechanisms.**

In this paper, we present BRIGADIER, an interactive security testing framework for NODE.JS applications that enables the specification of both application-level and business-level security policies. BRIGADIER provides developers with a Datalog-based policy specification language that features close interoperability with running JavaScript programs under test. Input JavaScript programs are instrumented to emit relevant application events sent to a Datalog engine resulting from the compilation of the policies. We exhibit BRIGADIER's expressiveness by implementing three case studies from the literature. We also assess BRIGADIER's performance overhead on server-side applications. In our benchmarks, we observed a slowdown factor ranging from ∼1.2x to ∼3x, which is acceptable for a testing scenario.

***Index Terms*—JavaScript, security testing, security policy, instrumentation, NODE.JS, Datalog, dynamic analysis**

I. INTRODUCTION

The NODE.JS back-end JavaScript runtime environment and the Node Package Manager (NPM) are a popular ecosystem for building server-side web applications [1, 2]. JavaScript's flexible and dynamic nature in combination with the vast amount of NPM available libraries is behind such popularity. But, at the same time, the dynamic nature of JavaScript and its flexibility often leads to vulnerable misbehaving applications.

We distinguish between two categories of vulnerabilities: application-level and business-level vulnerabilities. An *application-level* vulnerability is a weakness in an application that enables attackers to perform exploits by abusing insecure language features, vulnerable application components, or flawed design decisions [3]. An example would be the execution of a third-party component with more authority than needed, such as a math utility library with access to the network or the file system. Attackers may exploit the math library and use the unnecessarily allotted privileges to cause severe harm to the application and the systems it runs on. A *business-level vulnerability* (or business logic flaw) is a weakness in the business rules that enables an attacker to cause harm without making an incorrect use of the application [4, 5]. An example would be that the business logic fails to check whether the amount paid by a user corresponds to the accumulated price of all purchased items. This paper uses the term "vulnerability" to refer to either of these two categories.

Program analyses have been proposed to help developers to test and build secure JavaScript applications [6]. In this work, we focus on supporting the development of dynamic program analyses since they are invaluable in any real-world application testing scenario, often complementing other (static) techniques [7, 8, 9]. Since JavaScript was originally designed to be used for scripting client-based applications, most of the research efforts have been devoted to dynamically enforcing application security policies at client-side web applications (e.g., [10, 11, 12, 13, 14] for access control and [15, 16, 17, 18, 19, 20, 21] for information flow control). With the rising popularity of NODE.JS applications, some approaches have been recently proposed that specifically target the server-side of web applications (e.g., [22, 23, 24, 1, 25, 26, 2, 27]).

A. Problem Statement

By surveying the existing dynamic approaches for securing client-side and server-side JavaScript web applications, we observed five important shortcomings that hamper their expressiveness and applicability.

a) Imperative Policy Specification: Some approaches propose domain-specific languages for expressing custom application-specific policies [25, 26]. However, using imperative code in policy specifications increases the security risk because they are difficult to maintain, combine, and reuse.

b) Inflexible and Non-Extensible Policy Specification: Some analysis-based approaches, such as MIR [24] and SYNODE [1], propose ad-hoc security policies mechanisms. Although it is possible to configure these policies using a set of configuration parameters, it is not possible to modify or extend the set of enforced policies itself.

c) Coarse-Grained Policy Specification: NODEMOP [2] and the work of Ancona et al. [27] enable the specification of application-specific policies for NODE.JS programs in a declarative way. However, their policies can only reason about function calls. This reduces the expressiveness of these approaches, which in turn may force developers to rely on

different tools to express other security aspects that cannot be covered by only reasoning at the function level.

d) Intrusive Policy Deployment: Some approaches modify the target program or its runtime environment to deploy the security policy. These modifications can prevent other libraries or systems used by the application from being deployed in the program. For example, approaches such as NODESENTRY and MIR [24] cannot coexist within the same application, as both approaches modify the NODE.JS' module loader.

e) Policy Verification Disconnected From Application: Approaches, such as the one proposed by Ancona et al. [27], instrument the target program to generate events. These events are sent to a policy verification mechanism residing in a different process from the one of the target program. Having the policy verification in a different process poses two difficulties. First, program values must be serialised before being communicated to the analysis process. Unfortunately, not all JavaScript values can be easily serialized (e.g., functions). Second, passing large object graphs between processes is expensive in terms of performance.

B. Our Approach

In this paper, we present BRIGADIER [1], an interactive application security testing (IAST) framework for server-side JavaScript web applications. BRIGADIER was specially designed to cope with the five identified shortcomings.

First, BRIGADIER provides developers with a declarative language for specifying application and business-level policies. More concretely, policies are specified in a Datalog dialect. Such a declarative specification favours the policy's readability and understanding, leaving the implementation of its semantics to the enforcement mechanism. Additionally, declarative policy specifications facilitate the abstraction and composition of policy specifications and their reuse across applications.

In contrast to current declarative approaches [2, 27], security policies in BRIGADIER can observe and reason about more, and fine-grained application events. This enables the expression of a wide range of effective security policies that can control aspects of a JavaScript application's behaviour beyond function calls, e.g. property accesses or conditional statements.

To avoid intrusive policy deployment, BRIGADIER's policy deployment enables the coexistence of the security policy monitoring mechanism with other unrelated program analyses acting upon the same application.

Finally, BRIGADIER offers interoperability between the security policies and the values of the running JavaScript application under test. Because policies can directly inspect and operate on JavaScript values, policy specifications become simpler as no additional domain-specific features must be present for working with the underlying application values.

The contributions of this paper are:

- A declarative security policy specification language that features close interoperability with the values of the NODE.JS program under test.
- A non-intrusive policy verification mechanism that can exert fine-grained control over an application's behaviour.
- Libraries of predefined rules that ease the reasoning about popular NPM packages used within a target application.
- A validation of BRIGADIER's expressiveness by implementing access control and availability policies for application-level and business-level concerns of three case studies appearing in recent related work.

[1]https://gitlab.soft.vub.ac.be/ascullpu/brigadier

II. BRIGADIER

BRIGADIER consists of a policy specification language and a monitoring mechanism for analysing NODE.JS applications (i.e. enabling the runtime verification of the specified policies on a target application). BRIGADIER's policy language is inspired by Datalog and enables developers to specify security policies declaratively. The goal of these security policies is to constrain the application behaviour to avoid and detect application-level and business-level security vulnerabilities. Application behaviour is reified as facts that are used by rules to produce additional facts.

BRIGADIER's monitoring mechanism is designed to be used as part of an Interactive Application Security Testing (IAST) tool during the development and testing phase of the software development lifecycle. BRIGADIER takes a policy specification and a target NODE.JS application, and instruments this application to emit events for sensitive program operations. Such events include property access, function application, choosing a branch in a conditional statement, etc. Those application events are immediately turned into facts that are sent to the BRIGADIER's monitoring mechanism. Once a program operation violates the policy specification, BRIGADIER emits a notification. The goal of using BRIGADIER before deployment is to precisely detect and fix security vulnerabilities before they end up in a released product.

A. Brigadier By Example

We first introduce BRIGADIER by means of a security policy that detects calls to sensitive functions, shown in Listing 1. A BRIGADIER program specifies security policies as a non-empty set of facts and rules. The example policy consists of the definition of a fact (line 1), followed by the definition of a rule (lines 2–4). Line 1 adds a fact to BRIGADIER's sets of facts (part of its monitoring mechanism). The fact can be read as "`eval` is a sensitive function". A fact consists of a predicate name (e.g. `SecSensitiveFn`) and zero or more attribute values (e.g. `$eval`). The `$` preceding `eval` indicates that `eval` here is a JavaScript value, i.e., a pointer to the actual `eval` function used in the application. `$Math` and `$console` are other examples of pointers in BRIGADIER.

```
1 SecSensitiveFn($eval).
2 SecViolation(fn, iid) <-
3    BeforeCall(i, iid, fn, ths, args, res),
4    SecSensitiveFn(fn).
```

Listing 1. Example BRIGADIER policy for detecting sensitive function calls.

```
BeforeCall(i, iid, fn, ths, args)
AfterCall(i, iid, fn, ths, args, res)
BeforeWriteField(i, iid, ths, fld, val)
AfterRead(i, iid, name, val)
BeforeBinary(i, iid, op, lhs, rhs)
AfterConditional(i, iid, val)
```

Fig. 1. Excerpt of BRIGADIER language-level predicates.

The rule defined in Listing 1 can be read as "If a call to a function `fn` is about to happen, and `fn` is a sensitive function, then register this as a security violation". Whenever the body of a rule is satisfied, then the head fact is produced and added to the set of facts. In our example, whenever an application is about to apply a certain function object `fn`, the application emits an event of the type `BeforeCall`. If this `fn` is flagged as sensitive, then a `SecViolation` fact is produced.

`BeforeCall` actually corresponds to a *language-level* predicate for the aforementioned event type. Language-level predicates map relevant program operations to facts. Each language-level predicate has a specific list of *terms* that makes more detailed information of a program operation available in a BRIGADIER program. For example, the terms list `(i, iid, fn, ths, args)` for the `BeforeCall` predicate brings the called function (`fn`), the this (`ths`) object, and the arguments of the call (`args`) into the scope of the `SecViolation` rule. Additionally, all language-level predicate brings into the policy's scope a unique identifier `i` and the operation's location identifier `iid`. Having a unique identifier allows policies to reason about specific instances of a program operation (e.g, two calls to `eval`) and to maintain the order in which the operations were executed. The location identifier `iid` is used by utility functions to query location information at different levels of granularity. For example, line 5 of Listing 6 shows an example usage of the `fullPathOf(iid)` function to query the file path of the program operation at line 3.

Figure 1 contains an excerpt of the language-level predicates that BRIGADIER supports. Thanks to BRIGADIER's declarative nature, it is clear from the predicates declaration which type of application events they represent. Language-level predicates are prefixed with `Before` and `After` to indicate whether the information related to some operation represents the point in time right before or immediately after that operation. For example, predicate `AfterCall` is the `BeforeCall` counterpart and it additionally defines the return value (`res`) of the call. The full list of supported language-level predicates and their description can be found in Section A Table IV.

All predicates that are not language-level, are said to be user-defined predicates. In Listing 1, `SecSensitiveFn` and `SecViolation` are user-defined predicates expressing a security policy that detects calls to sensitive functions. A BRIGADIER program thus boils down to combining language-level facts (representing application events) and user-defined facts (representing policy configuration) into higher-level facts that express security policies.

$$
\begin{aligned}
\mathit{program} &:= [\mathit{require}]\ [\mathit{annotations}]\ \mathit{clause}+ \\
\mathit{require} &:= \texttt{require}\ (\ \text{string}[(\texttt{,}\ \text{string})*]\ ) \\
\mathit{clause} &:= \mathit{fact} \mid \mathit{rule} \\
\mathit{fact} &:= \mathit{predicate}\ (\ [\mathit{constant}\ (\ \texttt{,}\ \mathit{constant}\ )*]\ )\ \texttt{.} \\
\mathit{rule} &:= \mathit{head} \leftarrow \mathit{body}\ \texttt{.} \\
\mathit{head} &:= \mathit{predicate}\ (\ [\mathit{terms}]\ [\texttt{,}\ \mathit{aggregate}]\ ) \\
\mathit{body} &:= \mathit{atom}+ \\
\mathit{atom} &:= [\texttt{not}]\ \mathit{predicate}\ (\mathit{terms}\ ) \mid \mathit{app} \\
\mathit{app} &:= \mathit{app}\ \text{BINARY}\ \mathit{app} \\
&\quad \mid (\mathit{var} \mid \mathit{pointer})\ (\ [\mathit{app}\ [\texttt{,}\ \mathit{app}]*]\ ) \\
&\quad \mid \mathit{term} \mid (\ \mathit{app}\ ) \\
\mathit{annotations} &:= \mathit{annotation}* \\
\mathit{annotation} &:= \texttt{@}\ \text{identifier}\ (\ \mathit{annot\_entries}\ ) \\
\mathit{annot\_entries} &:= [\mathit{annot\_entry}\ [(\texttt{,}\ \mathit{annot\_entry})*]] \\
\mathit{annot\_entry} &:= \text{identifier}\ \texttt{:}\ \mathit{constant} \\
\mathit{terms} &:= \mathit{term}\ [(\ \texttt{,}\ \mathit{term})*] \\
\mathit{term} &:= \mathit{constant} \mid \mathit{var} \\
\mathit{constant} &:= \mathit{pointer} \mid \text{number} \mid \text{boolean} \\
&\quad \mid \text{string} \mid \text{null} \mid \text{undefined} \\
\mathit{pointer} &:= \texttt{\$}\text{identifier} \\
\mathit{aggregate} &:= (\texttt{\#max} \mid \texttt{\#count} \mid \texttt{\#sum}\ \ldots)\ \mathit{var}
\end{aligned}
$$

Fig. 2. BRIGADIER's formal grammar.

B. Syntax and Semantics

We now describe the syntax and semantics of BRIGADIER which are based on Datalog [28]. Figure 2 shows BRIGADIER's complete grammar in an extended BNF style, with names and symbols in green representing tokens. BRIGADIER adds three syntactic extensions to Datalog: the `require` statement, annotations (`@`), and pointers (`$`). The `require` statement extends a program with rules and facts contained in the imported file and is BRIGADIER's main mechanism for modularity and reuse. Annotations allow the specification of metadata that can be used to specialize how the policy is manipulated by BRIGADIER. An example of an annotation is `@Goal(name: 'Predicate')` which instructs the framework to emit facts of the relation specified by the `@Goal`'s `name` field. Finally, as introduced above, pointers (e.g. `$eval`) in BRIGADIER point to global variables in the target application.

C. BRIGADIER's interoperability

Unique to BRIGADIER is the interoperability between security policies and the values of the target JavaScript application. This enables developers to work directly with JavaScript values in their programs. Programmers do not need to turn JavaScript values from the target application into facts to

reason about them in their policies since JavaScript values are first-class in BRIGADIER. Listing 1 showed a basic form of interoperability by using pointers to global variables in the target program. In particular, that policy was able to have access to the actual `eval` JavaScript function (instead of a Datalog fact representing the function).

Rules can also apply plain JavaScript functions from their bodies. For example, consider a policy to report HTTP response cookies that do not have the `HttpOnly` flag set. Sending a cookie without the `HttpOnly` flag implies that the cookie is accessible to the client-side JavaScript code, which is a vulnerability as third-party code can freely access them. Listing 2 shows the specification of a policy to check whether the `HttpOnly` flag was set in all HTTP cookie headers sent by the server.

Assume that `ResponseCookie` facts represent cookie headers that store the cookie string in the first term (line 2). Cookie strings are represented as attributes that are separated by a semicolon. Therefore, to know whether the `HttpOnly` attribute is set in the cookie string, it should be split into its constituent attributes before the attribute's presence can be verified. In BRIGADIER, this can be achieved by applying the `split` and `elementOf` built-in functions. Both `split` and `elementOf` are plain JavaScript functions part of the BRIGADIER's built-in functions API.

Figure 3 provides an overview of the currently supported built-in functions. The list includes *utility* functions (such as `split` and `elementOf`) and, more importantly, reflective functions for introspecting the target application. For example, an important built-in function is `get(obj, prop)` as it provides a generic way for accessing object properties. This, in turn, enables BRIGADIER policies to have access to all methods and properties of objects reachable from the global object. Similarly, the other more specific utility built-in functions are useful shortcuts that can be used in BRIGADIER specifications without performing nested property lookup.

III. IMPLEMENTATION

We now describe the most relevant BRIGADIER's implementation details. Figure 4 shows its core components and how they interact. In the following, we highlight the important aspects of each component.

RULESPACE *Compiler:* BRIGADIER policy specifications are parsed using ANTLR [2], and converted into RULESPACE ASTs adorned with annotations. As shown in Figure 4, policies ASTs are given to the RULESPACE [3] component which compiles them to an incremental Datalog engine featuring aggregation and stratified negation [28]. The resulting engine is a self-contained and embeddable JavaScript module that triggers rule evaluations when facts are added. It can thus store any JavaScript value as a term of a fact enabling the interoperability between the analysis and the target application.

Handler Component: A handler wraps around the Datalog engine and provides the means for hooking on engine-specific events. The handler is notified of language-level facts whenever the application under test performs a sensitive operation. Hooking can be done by implementing `beforeAddFact` and `afterAddFact` methods of the `REHandler` class. Finally, the handler should provide the semantics of the annotations present in the analysis by implementing the `processAnnotations` method.

Listing 3 shows a sample implementation strategy of such a handler implementing the *afterAddFact* hook to notify the programmer whenever a goal fact has been derived. A goal fact represents an instance of a policy violation in the analysis program, for example, a call to `eval` in an application analyzed with Listing 1. The `processAnnotations` method (lines 6–11) first looks for an annotation named `Goal`. Then, at line 9, it extracts the relation name from the annotation *name* entry. At line 10, it gets the relation class corresponding to the relation name and saves it as the `goalClass` property. The `afterAddFact` method notifies the `goalListener` function whenever goal facts are generated.

```
1  class GoalLoggerHandler extends REHandler {
2   constructor(programAST, goalListener) {
3     super(programAST)
4     this.setGoalListener(goalListener);
5   }
6   processAnnotations() {
7     const annot = this.annotations
8           .find(annot => annot.annotName === 'Goal')
9     const q = this.getAnnotEntryValue(annot, "name")
10    this.goalClass = this.getRelationClass(q);
11  }
12  afterAddFact(fact, deltas) {
13    const delta = deltas.get(this.goalClass);
14    if (delta.length > 0) {
15      this.goalListener(delta);
16    }
17  }
18 }
```

Listing 3. Hanlder of the rule engine.

Analysis Composer: The analysis composer takes care of emitting facts corresponding to the relevant application events. To this end, it generates a JALANGI-based [29] analysis used to instrument the target application. As shown in Figure 4, the composer receives the policy AST and a handler as input. For each language-level predicate in the AST, it generates a corresponding JALANGI hook. The body of each hook function generates a language-level fact and adds it to the handler instance.

Considering the AST of the code shown in Listing 1 and an instance of the `GoalLoggerHandler` Listing 3 as input to the component, it generates an analysis object where just `BeforeCall` is materialized in the JALANGI analysis instance. An example of the shape and behaviour of the analysis object is shown in Listing 4. The `invokeFunPre` captures

[2]https://www.antlr.org/
[3]https://github.com/rulespace/rulespace

```
1 NonHttpOnlyCookie(cookie, path) <-
2   ResponseCookie(cookie, path, _, _, _, _),
3   false = elementOf('HttpOnly', split(cookie, ';')).
```

Listing 2. Policy detecting cookies URL paths sending non HttpOnly cookies.

```
apply(fn, ths, ...args)     //Applies the fn function to the given args with this bound to ths.
elementOf(el, col)          //Check whether el is in the iterable col.
arrayFrom(...args)          //Builds an array with the given args.
get(obj, prop)              //Returns the property the value of prop in obj.
prototypeOf(obj)            //Returns the prototype of obj.
constructorOf(obj)          //Returns the constructor of obj.
split(str, ptrn)            //Splits the string str using the pattern ptrn.
indexOf(str, ptrn)          //Returns the index of pattern ptrn in the string str.
lastIndexOf(str, ptrn)      //Returns the last index of pattern ptrn in the string str.
startsWith(str, ptrn)       //Checks whether the string str starts with the pattern ptrn.
endsWith(str, ptrn)         //Checks whether the string str ends with the pattern ptrn.
substring(str, a, b)        //Returns the substring of the string str from the range a to b.
concat(a, b)                //Concatenates the given array arguments.
max(a, b)                   //Returns the maximum of a and b.
requireModule(name)         //Performs the Node.js require and returns the required module's interface. It
                            //should be used with care because some modules are imported for side-effects.
locationOf(iid)             //Returns a meta-object containing location-related information of the given iid.
fullPathOf(iid)             //Returns a string representing full path of the given iid.
isInstanceOf(base, parent) //Checks whether base is an instance of parent.
```

Fig. 3. List of BRIGADIER built-in functions.

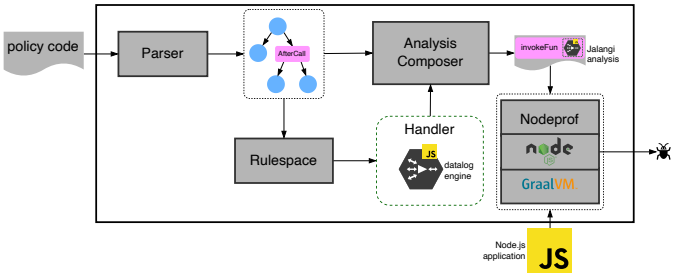

Fig. 4. BRIGADIER's architecture overview.

function call events that are mapped to the `BeforeCall` facts in Listing 1. At lines 3–5 the code creates a `BeforeCall` fact and adds it to the engine by means of the `addFact` method of the handler.

```
1 {
2   invokeFun: function(iid, fn, ths, xs){
3     const f = new BeforeCall(c++, iid, fn, ths, xs)
4     //logHndlr is an instance of GoalLoggerHandler
5     logHndlr.addFact(f)
6   }
7 }
```

Listing 4. Example implementation of a JALANGI analysis that triggers a RULESPACE engine by adding a language-level fact.

NODEPROF*:* BRIGADIER employs NODEPROF [30] to instrument the target application for generating the application events that are sent to the JALANGI-based analysis. The level of deep instrumentation offered by NODEPROF solves three of the problems identified in Section I-A. First, NODEPROF enables the specification and deployment of unintrusive policies concerning the program and its runtime environment. This means that BRIGADIER policy specifications can observe and reason about the target program and its environment without modifying a single line of code. Second, in contrast to other dynamic analysis tools for JavaScript such as JALANGI or ARAN, NODEPROF enables the instrumentation of the target program and its NPM dependencies, including NODE.JS built-in modules. Third, NODEPROF-based analyses can generate fine-grained application events allowing the implementation of dynamic analysis categories beyond what we discuss in this paper (e.g., taint analysis).

IV. VALIDATION

We now validate BRIGADIER to assess the expressiveness and performance of our approach. First, we show BRIGADIER expressiveness by implementing application-level and business-level access control and availability security policies (Section IV-A). Second, we analyzed BRIGADIER's performance by measuring its throughput using policies from the related work (Section IV-B). All experiments and measurements were performed on Ubuntu 22.04 LTS, running on a machine with an 8-core AMD Ryzen 7 3800X processor at 3.9GHz and 32GB of DDR4 RAM. GraalVM Node.js v14.17.6 running on GraalVM CE 21.3.0 (build 17.0.1+12-jvmci-21.3-b05) was used to execute the target applications.

A. Expressiveness

To exhibit BRIGADIER's capabilities for expressing different security policy families, we implemented the vulnerability *detection* phase of 3 recent systems proposing a business-level availability policy (i.e. Zeller et al. [4]) and application-level access control policies (i.e. NODESENTRY [26] and MIR [24]).

1) Policy System for Secure Integration of Third-Party Modules: NODESENTRY [26] provides support for defining *upper-bound* and *lower-bound* application-level access control policies around the public interface of third-party modules.

Lower-Bound policies: A lower-bound policy allows developers to reduce the authority that a host module has over the resources of a particular third-party module. For example, it could allow limiting the authority of `st` [31], a popular NPM module for serving static files. `st` attaches a request/response handler function to an `http` [4] server instance. Whenever a request triggering the handler is made, the `st` module parses the requested file's path, and reads

[4]https://nodejs.org/api/http.html

and sends the file contents to the user. For security reasons, the authority of `st` should be limited to accessing only the *network* API to reply to the user, and, more importantly, to performing only *read* operations on the file system. As such, the `st` module should only import read operations such as `readFile` and `readFileSync` from the `fs` module. However, the current `st` implementation (v3.0.0, July 2022) imports the full `fs` API, in spite of the fact that a manual review of the module's source code reveals that only a small subset of the `fs` module is actually used. This allows `st` module to obtain more authority than needed, making the module vulnerable to an attacker that may escalate the privileges and write to the file system.

A lower-bound policy for client modules of `st` must therefore restrict file system access within `st`'s implementation to *read* operations by allowing only specific `fs` operations to be used. Listing 5 shows the specification of such a lower-bound policy where an `AuthViolation` is raised whenever `st` attempts to call a file system API function other than the ones specified by an `Allow(...)` fact (see lines 3–5).

```
1  require('core')
2  @Goal(name:'AuthViolation')
3  Allow('path/st/', 'fs', 'createReadStream').
4  Allow('path/st/', 'fs', 'stat').
5  Allow('path/st/', 'graceful-fs', 'createReadStream').
6
7  ObjectLiteral(mod) <- Allow(_, modName ,_),
8                        mod := requireModule(modName).
9  ModName(modName) <- Allow(_, modName ,_).
10
11 AuthViolation(hostPath, mod, propName, srcObj) <-
12    ObjectProperty(mod, propName, fn, _, _),
13    ObjectLiteral(mod),
14    ModName(modName),
15    not Allow(hostPath, modName , propName),
16    BeforeCall(i, iid, fn, mod, _, _),
17    hostPath = fullPathOf(iid),
18    srcObj := sourceObjectOf(iid).
```

Listing 5. NODESENTRY's lower-bound policy in BRIGADIER for disallowing modules to write to the file system.

Fact `Allow(client, mod, op)` states that operation `op` is allowed to be used on a module `mod` required by client module *client*. The `require('core')` statement imports BRIGADIER's core utility module, which defines the `ObjectProperty` rule. The `@Goal` annotation specifies that facts with predicate `AuthViolation` are output facts. The `ObjectLiteral` rule uses the built-in `requireModule` function to get a reference to a `fs` module. Then, rule `AuthViolation` (see lines 11–18) flags a call of a function with name `propName` on a module `mod` required by a client module located at `hostPath` as a violation when it is *not* allowed. In this example, operations refer only to function calls, but extending the policy in BRIGADIER to authorizing property access is trivial.

Upper-Bound policies: An upper-bound policy allows developers to add security checks or repair code before or after a target module API is called. To exemplify the use of such an upper-bound policy, consider again the aforementioned `st` module. Prior to version 0.2.5, `st` was vulnerable to *path traversal* attacks because it incorrectly parsed the requested URL [25, 26]. Specifically, an attacker could use encoded dots (`%2e`) to bypass the check that prevented URLs from traversing directories by means of `/../`. Protecting the application from vulnerabilities within the `st` library enabling such a path traversal attack, would require checking and repairing all URLs flowing into the library. This can be checked by an upper-bound policy on the `st` module that validates whether the URL of an incoming request is "malformed" whenever the URL's value is read inside the module. Listing 6 shows the specification of such an upper-bound policy.

```
1 @Goal(name:'MaliciousURLViolation')
2 MaliciousURLViolation(result) <-
3    GetField(_, iid, ths, 'url', result),
4    'IncomingMessage' = constructorOf(ths),
5    '/path/st.js' = fullPathOf(iid),
6    indexOf(result, '%2e') > -1.
```

Listing 6. Example specification of an upper-bound policy for detecting malicious urls flowing into the `st` library.

Rule `MaliciousURLViolation` (lines 2–6) checks whether a `url` property that is being read (using `GetField`) is performed on an `IncomingMessage` object (i.e., an instance of an HTTP request) inside the `st` module. The `@Goal` annotation enables the notification to the user of any `MaliciousURLViolation` derived during the analysis.

2) Detecting violations of RWXI privileges of free variables on NPM *modules.:* Vasilakis et al. [24] introduced a fine-grained permission model around module boundaries called MIR that aims to prevent the exploitation of vulnerabilities in benign modules via its free variables. A policy in MIR specifies what privileges a module has over specific free variables (i.e. Read, Write, Execute, Import). More concretely, free variables refer to global and built-in variables accessible to all modules (e.g., `eval`, `Function`, `console`, `process`, ...) as well as variables locally accessible to modules such as `exports`, `module`, `require`, and modules imported using `require`.

```
serial:
  eval: RX
  module: R
  require("log"): I
```

Listing 7. Example of a MIR policy specification to restrict the authority of the `serial` library (inspired by [24].)

Listing 7 shows an example MIR policy stating that the `serial` module can only read or execute the `eval` function, read from the `module` variable, and import the `log` module. Any attempt to override the `eval` function, import another module, or calling a function exposed by `log`, will result in a policy violation. We now show how MIR's specification and behaviour can be expressed in BRIGADIER.

Listing 8 shows the policy specification in BRIGADIER which is a straightforward translation of MIR's policy shown in Listing 7.

```
1 PermSet('/path/serial.js', 'eval', 'RX').
2 PermSet('/path/serial.js', 'module', 'R').
3 PermSet('/path/serial.js', 'require("log")', 'I').
```

Listing 8. MIR policy configuration of Listing 9 expressed in BRIGADIER.

Recall that a policy violation should be reported whenever a module performs an operation on a free variable or one of its properties without permission. Listing 9 shows the core of the detection phase of the RWXI permission model in BRIGADIER. In particular, it shows the code for monitoring all operations on free variables within any application module.

```
1  GlobalObject('$Reflect', $Reflect).
2  ... //other GlobalObjects
3  WrapSafeFn(fn) <- ...
4  WrapSafeResult(compiledWrapper) <- ...
5
6  ModuleObject(filename, exps, req, mod, dirname) <-
7    AfterCall(_, _, refApply, ths, args, result),
8    refApply = get($Reflect, 'apply'),
9    WrapSafeResult(compiledWrapper),
10   compiledWrapper = get(args, 0),
11   arr := get(args, 2),
12   exps := get(arr, 0),
13   filename := get(arr, 3),
14   dirname := get(arr, 4)
15      ...
16 ModuleScopedObject(mName, 'exports', pointer) <-
17   ModuleObject(mName, pointer, _, _, _).
18 ... //other 'module local': module, require, etc.
19 //Identify module and global variables per module.
20 FVar(inMod, vName, ptr) <-
21   GlobalObject(vName, ptr),
22   Path(inMod, _, _).
23 FVar(inMod, vName, ptr) <-
24   ModuleScopedObject(inMod, vName, ptr),
25   Path(inMod, _, _).
26
27 ObservedProperty(m, ths, name, pVal, path, perms) <-
28   PropInPath(m, ths, name, pos, path),
29   Path(m, path, perms),
30   pos + 1 = get(path, 'length'),
31   pVal := get(ths, name).
32
33 ObservedVar(inMod, ths, name, pth, ps) <-
34       ...//a free var in a policy
35 //calling a free var function is secur. sensitive
36 SensitiveCall(inMod, fnPtr, fName, iid) <-
37   AfterCall(_, iid, fnPtr, ths, args, _),
38   FreeVar(inMod, fName, fnPtr),
39   inMod = fullPathOf(iid).
40 //whitelist free vars that can be called
41 AllowCall(inMod, fnPtr) <-
42   ObservedVar(inMod, fnPtr, fName, _, perms),
43   true = elementOf('X', perms).
44 AllowCall(inMod, fnPtr) <-
45   ObservedProperty(inMod, _, _, fnPtr, _, perms),
46   true = elementOf('X', perms).
47 //defines the policy goal for function calls
48 ACViolation(inMod, '[FUN CALL]', fName, iid) <-
49   SensitiveCall(inMod, fnPtr, fName, iid),
50   FVar(inMod, fName, val),
51   not AllowCall(inMod, val),
52   val = fnPtr.
53
54 Path(modName, spl, perms) <-
55    PermSet(modName, pathStr, perms),
56    false = startsWith(pathStr, 'require('),
57    spl := split(pathStr, '.').
```

Listing 9. Excert of the implementation using BRIGADIER of the detection phase of the RWXI permission model of Vasilakis et al. [24].

In BRIGADIER, identifying global variables from within the policy specification is straightforward as they are globally accessible to all modules within the application (line 1). However, free variables local to a module, such as `require`, `module`, `exports`, etc., are added to the module's scope at load time by NODE.JS' module loader. This requires hooking into the module loader to access module locals variables from outside a module's source code (lines 3–17). Rules `WrapSafe` and `WrapSafeResult` at lines 3 and 4, allow us to obtain all wrapper functions created by the loader around the applications' modules. As shown in Listing 10, such a wrapper function receives as arguments the module's local free variables.

```
1 function(exports, require, module, __filename ...) {
2 // module source code is copied here.
3 }
```

Listing 10. Excert of the wrapper function used by NODE.JS while loading modules.

The implementation then hooks into wrapper functions calls to collect calls arguments as shown in the specification of the `ModuleObject` rule (lines 6–14). A module local free variable is represented by a `ModuleScopedObject` fact. Lines 20–25 define the `FVars` which provide a uniform representation of the two types of free variables (i.e., global such as `eval` and module-local variables such as `module`) in the system).

Once the analysis has access to free variables, it is possible to determine what sensitive operations are happening during the program execution by matching the module's free variables (`FVar`) with program operation facts such as `AfterCall`, `AfterRead`, etc. The `SensitiveCall` rule shown in lines 36–39 is triggered whenever a free var is called within the module (`inMod`).

To know whether a sensitive operation violated any stated permission for a module, we define the `ObserverVar` and `ObservedProperty` rules. These rules match the permissions stated in an access path (`Path(modulePath, accessPath, permissions)`) with the corresponding free variables (`FVar`). An RWXI violation is derived by matching a sensitive operation (e.g., `SensitiveCall`) with the absence of the permission (e.g., `not AllowCall(...)`) for the free variable used by the operation.

3) Detecting logic Denial of Service (DoS) attacks against a hotel's reservation system: Zeller et al. [4] proposed a self-protection system for *detecting the misuse of business rules* of a hotel chain booking system. In particular, the business rules established by the hotel chain are:

1. The users must have a valid account with valid names and credit card information to avoid fake bookings.
2. The number of rooms and nights that can be booked is limited.
3. The hotel cannot be overbooked.
4. Hotel users have a free cancellation policy until one day before check-in.

In this system, the fourth rule is actually vulnerable to a logic attack. While the hotel management assumes that most of the guests will show up on the check-in day, the free cancellation policy enables a logical DoS attack. An attacker could perform bookings without the intention of checking in and cancelling the bookings at the last possible moment. In

this case, the *availability* of hotel rooms is depleted which may affect the hotel's revenue. Note also that knowing the user's intent is hard, therefore it is impossible to distinguish between benign and malicious users.

Zeller et al. [4] proposed the analysis of traces of late cancellations to detect such a DoS attack. All users cancelling the last day within the free cancellation policy period are subject to a policy that can lower their trust level. Whenever the number of daily late cancellation exceeds a predefined threshold the system lower the trust level of all users that perform a late cancellation on this specific day. The trust level is used to adapt the booking process for each user. Users with low trust levels may be required to perform additional verification steps to book a room.

We implemented the policy to detect the possible logic DoS attacks in BRIGADIER. To this end, we prototyped a JavaScript application using EXPRESS.JS simulating the booking process equivalent to the original paper's application (which was written in Java). The application comprises a `Hotel` class that implements the business logic for reserving and cancelling rooms. A `Hotel` instance has, for example, the `pay`, `cancelBooking`, and `book` methods. Reservations are stored as `Booking` instances in an in-memory database. Both the `Hotel` and `Booking` classes are public members of the emphapp/models.js module.

Listing 11 shows the implementation of the aforementioned DoS policy using BRIGADIER. Lines 2–8 specify how the application's module and the `Hotel` and `Booking` classes are brought to the analysis scope. Lines 10–14 register all accepted bookings by intercepting the instantiation (see `AfterConstructor` at line 11) of the `Booking` class. The `CancelledBooking` rule (lines 16–23) monitors all *late* cancellations of the system. The `CancellationsToday` rule (lines 26–27) counts daily late cancellations. While the `DecreasedTrustLevel` rule (lines 29–33) identifies the users whose trust level should be decreased, the `IncreaseTrustLevel` rule (lines 35–39) takes care of increasing the trust level when a user pays. Finally, the `UserTrust` (lines 50–53) computes the user's trust level using the difference between the `IncreasedTrustLevel` minus `DecreasedTrustLevel`.

```
1  ...
2  HotelMod(hotelMod) <-
3    hotelMod := requireModule('app/models.js').
4  BookingCtrFn(bookinCls) <-
5    HotelMod(mod), bookinCls := get(mod,'Booking').
6  Clazz(hotelCls) <-
7    HotelMod(mod),
8    hotelCls := get(mod, 'Hotel').
9
10 AcceptedBooking(user, inDay, i) <-
11   AfterConstructor(i, _, klass, _, args, res),
12   BookingCtrFn(klass),
13   user := get(args, 1),
14   inDay := get(args, 2).
15
16 CancelledBooking(currentDay, user, inDay, i) <-
17   ObjectProperty(_, 'cancelBooking', fn, _, _),
18   AfterCall(i, _, fn, _, args, result),
19   true = result,
20   currentDay := get(args, 3),
21   user := get(args, 0),
22   inDay := get(args, 1),
23   currentDay + 1 = inDay.
24 //store late cancellations for each day
25 CancellationsToday(today, #count _) <-
26   CancelledBooking(today, user, inDay, _).
27 //decrease trust of late users on day
28 //with more than 5 late canc.
29 DecreaseTrustLevel(user, today) <-
30   CancelledBooking(today, user, inDay, _),
31   CancellationsToday(today, amount),
32   amount >= 5,
33   today + 1 = inDay.
34 //incr. trust of all users paying their bookings
35 IncreaseTrustLevel(user, inDay) <-
36   ObjectProperty(_, 'pay', pay, _, _),
37   AfterCall(i, _, pay, _, args, result),
38   user := get(args, 0),
39   inDay := get(args, 1).
40
41 IncreasedTrustLevel(user, 0) <-
42   AcceptedBooking(user, _, _).
43 DecreasedTrustLevel(user, 0) <-
44   AcceptedBooking(user, _, _).
45
46 IncreasedTrustLevel(user, #count _) <-
47   IncreaseTrustLevel(user, today).
48 DecreasedTrustLevel(user, #count _) <-
49   DecreaseTrustLevel(user, today).
50
51 UserTrust(user, trustLevel) <-
52   DecreasedTrustLevel(user, ldec),
53   IncreasedTrustLevel(user, linc),
54   trustLevel := 3 + linc - ldec.
```

Listing 11. Policy specification for detecting a logic DDoS attack against an online booking system of a hotel.

TABLE I
COMPARISON OF APPROACHES IN TERMS OF LINES OF CODE (LOC).

Policy	Approach	LoC
Lower-bound	BRIGADIER	16
	NODESENTRY	12
Upper-bound	BRIGADIER	6
	NODESENTRY	13
RWXI	BRIGADIER	∼200
	MIR	∼2.8k

Discussion: We now compare BRIGADIER in terms of the three case studies conducted in this section. Table I summarizes the lines of code (LoC) of the original implementation with respect to the BRIGADIER one. Overall, BRIGADIER requires fewer lines of code. This is mainly due to its declarative nature combined with the fact it features interoperability with NODE.JS values. More importantly, BRIGADIER is able to implement all three kinds of policies without changes to the target application or its execution environment.

Table II compares BRIGADIER to the approaches targeting JavaScript used in this section (i.e. NODESENTRY and MIR) in terms of the shortcomings identified in related work in Section I-A. We do not compare to Zeller et al. [4] as it targets Java applications.

We consider the policy specification paradigm of both NODESENTRY and MIR as *hybrid* because parts of the policy

TABLE II
COMPARISON OF APPROACHES W.R.T THE IDENTIFIED SHORTCOMINGS ENUMERATED IN SECTION I-A: POLICY SPECIFICATION PARADIGM (PAR.), EXTENSIBLE/REUSABLE POLICY SPECIFICATIONS (EXT./REUS.), GRANULARITY (GRAN.), AND INTRUSIVE POLICY DEPLOYMENT (INTR.)

Approach	Par.	Ext./Reus.	Gran.	Intr.
BRIGADIER	declarative	yes	fine-grained	no
NODESENTRY	hybrid	no	fine-grained	yes
MIR	hybrid	no	fixed	yes

specifications are implemented imperatively. For example, NODESENTRY follows an aspect-oriented policy specification paradigm where points of interest are declaratively specified but the policy semantics is imperatively implemented. In MIR, programmers can declaratively configure the policy but, the policy itself is implemented imperatively. In contrast, policy specifications in BRIGADIER are fully declarative which improve their composition and reuse.

Regarding extensibility and reusability, neither MIR nor NODESENTRY are sufficiently general to enable the specification of the policies shown in this section without making changes to their implementation. For example, implementing upper-bound and lower-bound access control policies using MIR is just not possible without a major re-implementation of their system. On the other hand, detecting violations of RWXI privileges with NODESENTRY requires changes in the NODE.JS' module loader (i.e., the implementation of `require`) to wrap module local free variables. In contrast, BRIGADIER enables the specification of upper-bound, lower-bound and RWXI policies without changing the target application or its execution environment.

In terms of granularity, NODESENTRY and BRIGADIER offer fine-grained granularity for monitoring application events. We consider MIR to have a fixed granularity as programmers cannot reason nor combine specific application events.

Both NODESENTRY and MIR intrusive as they require modifications of the NODE.JS' module loader or the target application. As a result, they cannot be combined to protect a target application, i.e. they are incompatible. In contrast, BRIGADIER policies can coexist with either NODESENTRY or MIR policies without any additional requirement because our approach does not change the target application or its execution environment to perform the analysis.

Finally, while BRIGADIER, MIR and NODESENTRY have the policy verification in the same process as the monitoring, only BRIGADIER features close interoperability without modifying the target application.

B. Performance

To evaluate BRIGADIER in terms of performance, we measure the throughput of server-side applications. Our goal is to see how many `HTTP` requests an application under test can serve. This throughput gives the intuition of how many test cases can be processed in a unit of time. All experiments in this section were executed with `wrk` [5] using 15 seconds as the time limit and one thread as configuration parameters.

Application-level AC: We used the NODE.JS test program used in NODESENTRY [26], OWASP Juice Shop [6] and OWASP NodeGoat [7] as target applications. We used 3 configurations to execute each application: plain NODE.JS as baseline, NODE.JS with BRIGADIER without policies and NODE.JS with BRIGADIER and policies. Table III describes the lower-bound and upper-bound policies combination used to evaluate each test program. Each configuration was executed 5 times using a fresh VM. We do not consider the warmup time or peak performance of the VM, because, in a testing scenario, the VM is restarted very often preventing it from reaching peak performance.

Figure 5 shows the throughput achieved by a target application with policies deployed. An application executed using BRIGADIER (blue bars) without any policy does not suffer from any performance overhead as the code is not instrumented. However, once the application is subject to a policy (yellow and red bars) the throughput decreases from $\sim$1.2x to $\sim$3x.

Business-level availability: In a second experiment, we measured the throughput of the hotel chain booking system subject to the availability policy implemented in Listing 11. In this experiment, the baseline execution featured a throughput of $\sim$3588 req/sec, while execution subject to the policy had a throughput of $\sim$1472 req/sec. On average, running the application with the policy is $\sim$2.4x slower than the baseline.

Discussion: We believe that the main reason for our performance overhead is the Datalog engine. Our engine compiler materializes policies into a program that uses a bottom-up evaluation strategy. However, typical optimizations of database systems, such as indexes, rule re-writing, and efficient data structures are still missing. Although this overhead is high, the throughput achieved is still acceptable for a testing scenario. Moreover, this performance penalty is outweighed by the policy readability, expressiveness and non-intrusiveness achieved with BRIGADIER.

V. RELATED WORK

In this section, we discuss both, language-based approaches for securing client-side and server-side JavaScript applications and security approaches based on Datalog.

Approaches for securing client-side JavaScript applications: As mentioned in the introduction, a bulk of research has focused on securing client-side web applications. [17, 19, 18, 16, 21, 20, 15] propose systems for enforcing information flow control policies, and [32, 12, 11, 13, 14, 33, 10] propose systems for enforcing access control policies. However, none of those approaches is expressive enough to support the implementation of both application-level and business-level policies without resorting to the host language, usually JavaScript. For example, policy specifications of such approaches such as

[5] https://github.com/wg/wrk
[6] https://owasp.org/www-project-juice-shop/
[7] https://owasp.org/www-project-node.js-goat/

TABLE III
DESCRIPTION OF THE UPPER-BOUND AND LOWER-BOUND POLICIES USED FOR THE PERFORMANCE BENCHMARK EXPERIMENT.

Application	Lower-bound policy	Upper-bound policy
st server	Detect unwanted file system accesses within the st module.	Detect malicious urls with path traversal characters when entering the st module.
JuiceShop	Detect unwanted file system accesses within the send module.	Detect malicious url with null byte payloads when entering the fileServer component of the application.
NodeGoat	Detect calls to eval within the contributions component of the application.	Detect whether a given Express.js route handler can be executed without calling an authorization middleware previously.

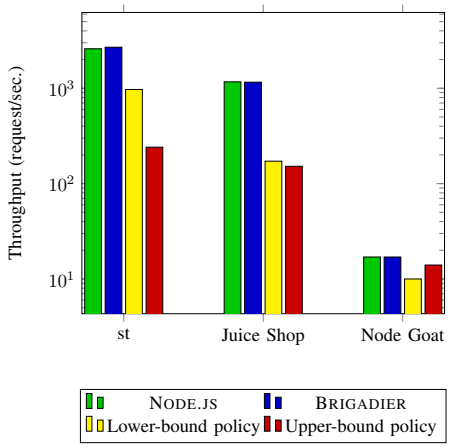

Fig. 5. Throughput (requests per second) of a server-based application using the *st* library to serve static files. Each bar represents the average *request/sec.* of executing 5 times the test program on each configuration.

Phung et al. [12] and CONSCRIPT [14], where the specification follows an aspect-oriented programming style, require the implementation of the policy semantics using imperative code. In BRIGADIER, both, the point-cut and the policy semantics are declaratively specified.

Approaches for securing NODE.JS *applications:* From the existing dynamic approaches for NODE.JS applications [22, 23, 24, 1, 25, 26, 2, 27], only NODEMOP and the work of Ancona et al. [27] offer declarative policy specification like BRIGADIER. NODEMOP [2] propose a *runtime verification* framework for NODE.JS applications. Users of NODEMOP can specify correctness properties (or security properties) by defining events and monitors. Monitors can detect and recover from incorrect code patterns at run time. A monitor requires the specification of *events* to map from function calls to runtime events that are verified using a regular expression-based language. In contrast, BRIGADIER can monitor other operations than function calls, and can reason about all the contextual information of the operation being monitored (e.g., the `this` value, and property name and value).

Ancona et al. [27] propose a runtime verification framework for NODE.JS and NODE-RED [8] applications. Applications are instrumented using JALANGI to generate events that are communicated to a server hosting a PROLOG interpreter via HTTP. To avoid the issues when the policy verification is disconnected from the application explained in the introduction, in BRIGADIER the DATALOG interpreter lives in the same process as the target application. This results in better interoperability between policies and applications as well as easing the policy specification.

Datalog as Security Policy Language: Datalog has been used previously as a language to express application-level security policies in static program analysis approaches. In this context, it has been used for expressing policies in static analyses such as access control [9] and taint analysis [34, 9, 35]. Those Datalog dialects do not support interoperability with the target program which is essential to enable precise introspection and manipulation of the values of the program under analysis.

In dynamic analysis, Datalog has been used as policy language for other kinds of security policies than those targeted by BRIGADIER. For example, many approaches [36, 37, 38, 39, 40] of *trust management* in distributed systems adopted Datalog or extensions of it as the policy language. At application-level,FAF [41] proposed a Datalog-based policy language for specifying authorization policies that describe which actions the users of the system can perform on data items. These authorization policies are then enforced whenever a user request permission to perform an action on a data item.

VI. CONCLUSION

This paper presented BRIGADIER, an interactive security testing framework for NODE.JS applications. Developers specify security policies using a Datalog-based language to detect application-level and business-level vulnerabilities. To the best of our knowledge, BRIGADIER is the first IAST framework for application-level and business-level policies leveraged on top of a Datalog engine. In contrast to prior dynamic or static datalog-based approaches, BRIGADIER features good interoperability with the values of the program under analysis.

We demonstrated BRIGADIER's expressiveness by means of three case studies from recent related work proposing application-level and business-level policies. We also conducted benchmarks to assess the performance overhead of BRIGADIER. Even though we observed a slowdown factor ranging from $\sim$1.2x to $\sim$3x, we believe it is still acceptable for a security testing scenario. The next steps include improving BRIGADIER's performance by adding optimizations on the Datalog engine and providing support to enable policy developers to delete unused/outdated facts from the database.

[8]https://nodered.org/

References

[1] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS," in *Proceedings 2018 Network and Distributed System Security Symposium*, no. February. Reston, VA: Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07A-2_Staicu_paper.pdf

[2] F. Schiavio, H. Sun, D. Bonetta, A. Rosà, and W. Binder, "NodeMop: Runtime verification for Node.js applications," in *Proceedings of the ACM Symposium on Applied Computing*, vol. Part F1477. Association for Computing Machinery, 2019, pp. 1794–1801.

[3] R. Ross, "Guide for conducting risk assessments," 2012-09-17 2012.

[4] S. Zeller, N. Khakpour, D. Weyns, and D. Deogun, "Self-protection against business logic vulnerabilities," *Proceedings - 2020 IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2020*, pp. 174–180, 2020.

[5] G. Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais, "DetLogic: A black-box approach for detecting logic vulnerabilities in web applications," *Journal of Network and Computer Applications*, vol. 109, no. September 2017, pp. 89–109, 2018. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2018.01.008

[6] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Comput. Surv.*, vol. 50, no. 5, sep 2017. [Online]. Available: https://doi.org/10.1145/3106739

[7] A. L. S. Pupo, J. Nicolay, and E. G. Boix, "Deriving static security testing from runtime security protection for web applications," *Art Sci. Eng. Program.*, vol. 6, no. 1, p. 1, 2022. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2022/6/1

[8] J. Nicolay, V. Spruyt, and C. D. Roover, "Static detection of user-specified security vulnerabilities in client-side javascript," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, T. C. Murray and D. Stefan, Eds. ACM, 2016, pp. 3–13. [Online]. Available: https://doi.org/10.1145/2993600.2993612

[9] S. Guarnieri and B. Livshits, "Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code," in *Proceedings of the 18th USENIX Security Symposium*, 2009, pp. 151–168.

[10] A. L. Scull Pupo, J. Nicolay, and E. Gonzalez Boix, *GUARDIA: specification and enforcement of JavaScript security policies without VM modifications*, ser. Proceedings of the 15th International Conference on Managed Languages & Runtimes - ManLang '18. ACM, 09 2018.

[11] J. Magazinius, P. H. Phung, and D. Sands, "Safe Wrappers and Sane Policies for Self Protecting JavaScript," ser. Informatics, 2012, vol. 7127, pp. 239 – 255.

[12] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting JavaScript," ser. the 4th International Symposium, 2009, pp. 47 – 60.

[13] L. A. Meyerovich, A. P. Felt, and M. S. Miller, "Object views: Fine-Grained Sharing in Browsers," ser. the 19th international conference, 2010, pp. 721 – 730.

[14] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser," ser. 2010 IEEE Symposium on Security and Privacy, 2010, pp. 481 – 496.

[15] A. L. Scull Pupo, L. Christophe, J. Nicolay, C. De Roover, and E. Gonzalez Boix, "Practical Information Flow Control for Web Applications." *RV*, vol. 11237, no. 5, pp. 372 – 388, 2018.

[16] A. Bichhawat, V. Rajani, J. Jain, D. Garg, and C. Hammer, "WebPol: Fine-Grained Information Flow Policies for Web Browsers," ser. Computer Security – ESORICS 2017. Springer, Cham, 09 2017, vol. 10492, pp. 242 – 259.

[17] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow - tracking information flow in JavaScript and its APIs." *SAC*, pp. 1663 – 1671, 2014.

[18] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information Flow Control in WebKit's JavaScript Bytecode," ser. Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues. Springer Berlin Heidelberg, 2014, vol. 8414, pp. 159 – 178.

[19] A. Chudnov and D. A. Naumann, "Inlined Information Flow Monitoring for JavaScript," ser. the 22nd ACM SIGSAC Conference, 2015, pp. 629 – 643.

[20] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow." *POPL*, p. 165, 2012.

[21] D. Devriese and F. Piessens, "Noninterference through Secure Multi-execution," ser. 2010 IEEE Symposium on Security and Privacy (SP), 2010, pp. 109 – 124.

[22] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis," in *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Inc, aug 2021, pp. 268–279. [Online]. Available: https://doi.org/10.1145/3468264.3468542

[23] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: Feedback-driven static analysis of Node.js applications," in *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, vol. 19, 2019, pp. 455–465. [Online]. Available: https://doi.org/10.1145/3338906.3338933

[24] N. Vasilakis, C. A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. Dehon, and M. Pradel, "Preventing dynamic library compromise on node.js via rwx-based privilege reduction." Association for Computing

Machinery, 11 2021, pp. 1821–1838. [Online]. Available: https://doi.org/10.1145/3460120.3484535

[25] 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*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 446–455. [Online]. Available: https://doi.org/10.1145/2664243.2664276

[26] N. Van Ginkel, W. De Groef, F. Massacci, and F. Piessens, "A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries," *Security and Communication Networks*, vol. 2019, 2019.

[27] D. Ancona, L. Franceschini, G. Delzanno, M. Leotta, M. Ribaudo, and F. Ricca, "Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things," *Electronic Proceedings in Theoretical Computer Science, EPTCS*, vol. 264, pp. 27–42, feb 2018. [Online]. Available: http://arxiv.org/abs/1802.01790http://dx.doi.org/10.4204/EPTCS.264.4

[28] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou, "Datalog and recursive query processing," *Foundations and Trends in Databases*, vol. 5, no. 2, pp. 105–195, 2012.

[29] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 488–498. [Online]. Available: https://doi.org/10.1145/2491411.2491447

[30] H. Sun, C. Humer, D. Bonetta, and W. Binder, "Efficient dynamic analysis for node.Js," in *CC 2018 - Proceedings of the 27th International Conference on Compiler Construction, Co-located with CGO 2018*, vol. 2018-Febru, 2018.

[31] I. npm, "st," https://www.npmjs.com/package/st, July 2022, (Accessed on 07/27/2022).

[32] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications." *ACSAC*, pp. 1 – 10, 2012.

[33] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. R. Pietzuch, and R. Kapitza, "TrustJS - Trusted Client-side Execution of JavaScript." *EUROSEC*, pp. 1 – 6, 2017.

[34] B. Livshits, "Improving software security with precise static and runtime analysis," Ph.D. dissertation, Stanford, CA, USA, 2006, aAI3242585.

[35] N. Grech and Y. Smaragdakis, "P/Taint: Unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 2017. [Online]. Available: https://doi.org/10.1145/3133926

[36] N. Li, B. N. Grosof, and J. Feigenbaum, "A Logic-based Knowledge Representation for Authorization with Delegation," 1999.

[37] W. Winsborough, K. Seamons, and V. Jones, "Automated trust negotiation," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 1, 2000, pp. 88–102 vol.1.

[38] T. Jim, "SD3: A trust management system with certified evaluation," in *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, 2001, pp. 106–115. [Online]. Available: www.lcs.mit.edu.'.

[39] J. DeTreville, "Binder, a logic-based security language," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2002-January, pp. 105–113, 2002.

[40] N. Li and J. C. Mitchell, "DATALOG with constraints: A foundation for trust management languages," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2562, pp. 58–73, 2003.

[41] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian, "Flexible Support for Multiple Access Control Policies," *ACM Transactions on Database Systems*, vol. 26, no. 2, pp. 214–260, 2001.

APPENDIX A
LANGUAGE-LEVEL PREDICATES

Table IV describes the language-level predicates currently supported by BRIGADIER.

TABLE IV
LIST OF BRIGADIER LANGUAGE-LEVEL PREDICATES.

Predicate	**Description**
`BeforeCall(i, iid, fn, ths, args)`	Captures the function `fn`, the this `ths` and arguments `args` before a function call.
`AfterCall(i, iid, fn, ths, args, res)`	Captures the function `fn`, the this `ths`, arguments `args` and result `res` after a function call.
`BeforeConstructor(i, iid, fn, args)`	Captures the function `fn` and arguments `args`used before a constructor call.
`AfterConstructor(i, iid, fn, args, res)`	Captures the function `fn`, arguments `args` and result `res` after a constructor call.
`BeforeGetField(i, iid, ths, fld)`	Captures the information before getting an object `ths` property's (`fld`) value
`AfterGetField(i, iid, ths, fld, res)`	Captures the information after getting an object `ths` property's (`fld`) value `res`.
`BeforeWriteField(i, iid, ths, fld, val)`	Captures the information before setting an object `ths` property's (`fld`) value `val`.
`AfterWriteField(i, iid, ths, fld, val)`	Captures the information after setting an object `ths` property' (`fld`) value `val`.
`AfterWrite(i, iid, name, val)`	Captures the information after writing a variable `name` with the value `val`.
`AfterRead(i, iid, name, val)`	Captures the information after reading a variable `name` holding the value `val`.
`BeforeBinary(i, iid, op, lhs, rhs)`	Captures the information before evaluating a binary expression `op` with left `lhs` and right `rhs`.
`AfterBinary(i, iid, op, lhs, rhs, res)`	Captures the information after evaluating a binary expression `op` with left `lhs` and right `rhs` and value `res`.
`AfterLiteral(i, iid, val)`	Captures the information after evaluating a literal `val`.
`AfterConditional(i, iid, val)`	Captures the information after evaluating a conditional expression with result `val`.