Understanding and Automatically Preventing Injection Attacks on Node.js

Cristian-Alexandru Staicu, Michael Pradel, Ben Livshits

Technical Report TUD-CS-2016-14663

TU Darmstadt, Department of Computer Science

November, 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

SOLA
SoftwareLab

Understanding and Automatically Preventing Injection Attacks on NODE.JS

Cristian-Alexandru Staicu*, Michael Pradel*, and Benjamin Livshits†

TU Darmstadt* Microsoft Research†

***Abstract*—The NODE.JS ecosystem has lead to the creation of many modern applications, such as server-side web applications and desktop applications. Unlike client-side JavaScript code, NODE.JS applications can interact freely with the operating system without the benefits of a security sandbox. The complex interplay between NODE.JS modules leads to subtle injection vulnerabilities being introduced across module boundaries. This paper presents a large-scale study across 235,850 NODE.JS modules to explore such vulnerabilities. We show that injection vulnerabilities are prevalent in practice, both due to `eval`, which was previously studied for browser code, and due to the powerful `exec` API introduced in NODE.JS. Our study shows that thousands of modules may be vulnerable to command injection attacks and that even for popular projects it takes long time to fix the problem. Motivated by these findings, we present SYNODE, an automatic mitigation technique that combines static analysis and runtime enforcement of security policies for allowing vulnerable modules to be used in a safe way. The key idea is to statically compute a template of values passed to APIs that are prone to injections, and to synthesize a grammar-based runtime policy from these templates. Our mechanism does not require the modification of the NODE.JS platform, is fast (sub-millisecond runtime overhead), and protects against attacks of vulnerable modules while inducing very few false positives (less than 10%).**

I. INTRODUCTION

JavaScript is the most widely-used programming language for the client-side of web applications, powering over 90% of today's web sites[1]. Recently, JavaScript has become increasingly popular for platforms beyond the browser: server-side and desktop applications (NODE.JS), mobile programming (Apache Cordova/PhoneGap); it is even used for writing operating systems (Firefox OS). One of the forces behind using JavaScript in other domains is to enable client-side programmers to reuse their skills in other environments.

Unfortunately, this skill transfer also spreads the risk of misusing JavaScript in a way that threatens software security. On the one hand, some of the bad habits of client-side JavaScript, such as the widespread use of the `eval` construct [24], spread to additional platforms. On the other hand, new vulnerabilities and kinds of attacks become possible, which do not directly map to problems known from the client-side. For example, recent work shows that mobile applications written in JavaScript contain injection vulnerabilities [12] and that the impact of attacks in mobile applications is potentially more serious than that of client-side cross-site scripting (XSS).

[1]JavaScript use statistics: http://w3techs.com/technologies/details/cp-javascript/all/all.

This paper is the first to thoroughly investigate a security issue specific to JavaScript executed on the NODE.JS platform. Specifically, we focus on injection vulnerabilities, i.e., programming errors that enable an attacker to inject and execute malicious code in an unintended way. Injection vulnerabilities on the NODE.JS platform differ from those on other JavaScript platforms in three ways.

1) Injection APIs and impact of attacks: NODE.JS provides two families of APIs that may accidentally enable injections. The `eval` API and its variants take a string argument and interpret it as JavaScript code, allowing an attacker to execute arbitrary code in the context of the current application. The `exec` API and its variants take a string argument and interpret it as a shell command, allowing an attacker to execute arbitrary system-level commands, beyond the context of the current application. Moreover, attackers may combine both APIs by injecting JavaScript code via `eval`, which then uses `exec` to execute shell commands. Because of these two APIs and because NODE.JS lacks the security sandbox known from browsers, injection vulnerabilities can cause significantly more harm than in browsers, e.g., by modifying the local file system or even taking over the entire machine.

2) Developer stance: While it is tempting for researchers to propose an analysis that identifies vulnerabilities as a solution, to have longer-range impact, it helps to understand NODE.JS security more holistically. By analyzing security issues reported in the past and through developer interactions, we observed that, while injection vulnerabilities are indeed an important problem, developers who both use and maintain JavaScript libraries are reluctant to use analysis tools and are not always willing to fix their code.

To understand the attitude of NODE.JS module developers toward potential injection flaws, we submitted a sample of 20 bug reports to developers on GitHub. Somewhat to our surprise, only about half were responded to and only a small fraction was fixed (the results of this experiment are detailed in Figure 6). To understand the situation further, we reviewed many cases of the use of `eval` and `exec`, to discover that most (80%) could be easily refactored by hand, eliminating the risk of injections [19]. These observations suggest that it is unlikely that, given the right analysis tool, developers will proceed to voluntarily fix potential vulnerabilities.

3) Blame game: A dynamic we have seen develop is a blame game between NODE.JS module maintainers and developers

who use these modules, where each party tries to claim that the other is responsible for checking untrusted input. Furthermore, while a developer can find it tempting to deploy a local fix to a vulnerable module, this patch is likely to be made obsolete or will simply be overwritten by the next module update. These observations motivated us to develop an approach that provides a high level of security with a very small amount of developer involvement.

Given the above observations, it is important to offer a solution that provides complete automation. This paper presents SYNODE, an automatic approach to identify potential injection vulnerabilities and to prevent injection attacks. The basic idea is to check all third-party modules as part of their installation and to rewrite them to enable a *safe mode*. A mix of two strategies is applied as part of rewriting. First, we propose to *statically* analyze the values that may be passed to APIs prone to injections. The static analysis extracts a template that describes values passed to the APIs. Second, for code locations where the static analysis cannot ensure the absence of injections, we present a *dynamic* enforcement mechanism that stops malicious inputs before passing them to the APIs. A combination of these techniques is applied to a module at the time of installation via the use of NODE.JS installation hooks, effectively enforcing a safe mode for third-party modules. In principle, our runtime enforcement may be overly conservative, but our evaluation shows that such cases are rare.

A. Contributions

- **Study**: We present a study of injection vulnerabilities in NODE.JS modules, focusing on why and how developers use potentially dangerous APIs and whether developers appear open to using tools to avoid these APIs. (Section III)
- **Static analysis**: We present a static analysis that attempts to infer templates for the user input to be used at potentially dangerous sinks. (Section IV)
- **Runtime enforcement**: For cases that cannot be shown safe via static analysis, we present a runtime enforcement achieved through code rewriting. The runtime approach uses partially instantiated abstract syntax trees (ASTs) and ensures that the runtime values do not introduce any unwanted code beyond what is expected. (Section IV)
- **Evaluation**: We apply our static technique to a set of 16,795 NODE.JS modules. We discover that 36.66% of them are statically guaranteed to be safe. For a subset of the statically unsafe modules, we create both malicious inputs that exploit the injection vulnerabilities and benign inputs that exercise the advertised functionality of the module. Our runtime mechanism effectively prevents 100% of the attacks, while being overly conservative for only 8.92% of the benign inputs.

II. BACKGROUND AND EXAMPLE

NODE.JS and injection APIs: The NODE.JS platform is the de-facto standard for executing JavaScript outside of browsers.

```
1 function backupFile(name, ext) {
2   var cmd = [];
3   cmd.push("cp");
4   cmd.push(name + "." + ext);
5   cmd.push("~/.localBackup/");
6
7   exec(cmd.join(" "));
8
9   var kind = (ext === "jpg") ? "pics" : "other";
10  console.log(eval("messages.backup_" + kind));
11 }
```

Fig. 1: Motivating example

The platform provides two families of APIs that may allow an attacker to inject unexpected code, which we call *injection APIs*. First, `exec` enables command injections if an attacker can influence the string given to `exec`, because this string is interpreted as a shell command. The `exec` API has been introduced for NODE.JS and is not available in browsers. Second, calling `eval` enables code injections if an attacker can influence the string passed to `eval`, because this string is interpreted as JavaScript code.[2] Since code injected via `eval` may contain calls to `exec`, any code injection vulnerability is also a command injection vulnerability. The latter distinguishes server-side JavaScript from the widely studied [24] client-side problems of `eval` and introduces an additional security threat.

In contrast to the browser platform, NODE.JS does not provide a security sandbox that controls how JavaScript code interacts with the underlying operating system. Instead, NODE.JS code has direct access to the file system, network resources, and any other operating system-level resources provided to the processes. As a result, injections are among the most serious security threats on NODE.JS, as evidenced by the Node Security Platform[3], where, at the time of writing, 20 out of 66 published security advisories address injection vulnerabilities.

Module system: Code for NODE.JS is distributed and managed via the *npm* module system. A module typically relies on various other modules, which are automatically installed when installing the module. There is no mechanism built into npm to specify or check security properties of third-party modules before installation.

Motivating example: Figure 1 shows a motivating example that we use throughout the paper to illustrate our approach. The function receives two parameters from an unknown source and uses them to copy a file on the local file system. The parameters are intended to represent a file name and a file extension, respectively. To copy the file, lines 2 to 5 construct a string that is passed as a command to `exec` (line 7), which will execute a shell command. The code also logs a message to the console. Line 10 retrieves the content of the message by looking up a property of the `messages` object. The property,

[2]We focus on `exec` and `eval` in this paper, as these are the most prominent members of two families of APIs. Extending our work to more APIs, e.g., `new Function()`, is straightforward.

[3]https://nodesecurity.io/advisories/

and therefore also the message, depends on the extension of the backed up file. Implementing a lookup of a dynamically computed property with `eval` is a well-known misuse of `eval` that frequently occurs in practice [24]. For example, suppose the function is called with `backupFile("f", "txt")`. In this case, the command will be `cp f.txt ~/.localBackup` and the logged message will be the message stored in `messages.backup_other`.

The example contains two calls to APIs that may allow for injecting code (lines 7 and 10). As an example for an injection attack, let us consider the following call:

```
backupFile("--help && rm -rf * && echo ", "")
```

The dynamically constructed command will be:

```
cp --help && rm -rf * && echo . ~/.localBackup/
```

Unfortunately, this command does not backup any files but instead it creates space for future backups by deleting all files in the current directory. Such severe consequences distinguish the problem of injections on NODE.JS from injections known from client-side JavaScript, such as XSS: because NODE.JS code runs without any sandbox that could prevent malicious code from accessing the underlying system, an attacker is able to inject arbitrary system-level commands.

III. A STUDY OF INJECTION VULNERABILITIES

To better understand how developers of JavaScript for NODE.JS handle the risk of injections, we conduct a comprehensive empirical study involving 235,850 npm modules. We investigate four research questions (RQs):

- **RQ1: Prevalence**. At first, we study whether APIs that are prone to injection vulnerabilities are widely used in practice. We find that injection APIs are used frequently and that many modules depend on them either directly or indirectly.
- **RQ2: Usage**. To understand why developers use injection APIs, we identify recurring usage patterns and check whether the usages could be replaced with less vulnerable alternatives. We find that many uses of injection APIs are unlikely to be replaced by less vulnerable alternatives.
- **RQ3: Existing mitigation**. To understand how developers deal with the risk of injections, we study to what extent data gets checked before being passed into injection APIs. We find that most call sites do not at all check the data passed into injection APIs.
- **RQ4: Maintenance**. To understand whether module developers are willing to prevent vulnerabilities, we report a sample of vulnerabilities to developers and analyze how the developers react. We find that, even for widely used modules, most vulnerabilities remain in the code several months after making the developers aware of the issue.

The remainder of this section presents the results of our study in detail. Sections III-A to III-D addresses the research questions by studying all or a representative sample of all npm modules. Section III-E reports a detailed analysis of one particular module. The study is based on data collected from npm in February 2016.

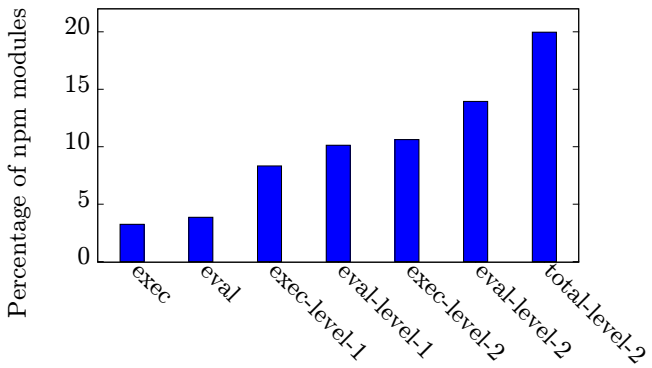


Fig. 2: Prevalence of usages of injection APIs in npm modules.

A. RQ1: Prevalence

To assess how wide-spread the use of injection APIs is in real-world JavaScript code written for NODE.JS, we analyze call sites of these APIs across all 235,850 npm modules. For each module, we perform a regular expression-based search that detects call sites of injection APIs based on the name of the called function. We call modules that rely on these injection APIs *injection modules*. Furthermore, we analyze dependences between modules, as specified in their `package.json` file, to assess whether a module uses another module that calls an injection API. Given a module m_{inj} that calls an injection API, we say that another module m_1 has a "level-1" ("level-2") dependence if it depends on m_{inj} (via another module).

Figure 2 shows how many npm modules use injection APIs, either directly or via another module. 7,686 modules and 9,111 modules use `exec` and `eval`, respectively, which corresponds to 3% and 4% of all modules. Furthermore, more than 10% of all modules have a level-2 dependence on a module calling `exec` and almost 15% on a module calling `eval`. We conclude that the risk of injections is prevalent and that it propagates to a significant part of the NODE.JS code base.

The results from Figure 2 raise the question as to how to effectively protect npm modules against injections. For example, if many modules with level-1 dependences on injection APIs would depend on only a single injection module, then protecting this module against injections would protect all its dependent modules. To estimate how many modules would be directly and indirectly protected by protecting a subset of all modules, we analyze the module dependence graph. Specifically, we initially mark all injection modules and all modules with level-1 dependences as unprotected and then, assuming that some set of injection modules is protected, propagate this information along dependence edges. Based on this analysis, we compute a minimal set of injection modules to protect, so that a particular percentage of modules is protected. The strategy we consider is to fix vulnerabilities in those modules with the highest number of level-1 dependences.

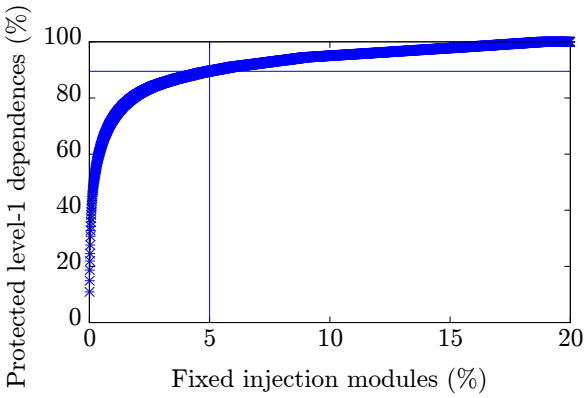


Fig. 3: Percentage of modules that would be protected by protecting a particular percentage of modules that use injection APIs.

Figure 3 summarizes the results of this experiment. We find that fixing calls to the injection APIs in the most popular 5% of all injection modules will protect almost 90% of the level-1 dependences. While this result is encouraging, it is important to note that 5% of all modules still corresponds to around 780 modules, i.e., many more than could be reasonably fixed manually.

B. RQ2: Usages

The results from RQ1 show that the risk of injections is prevalent. In the following, we try to understand why developers take this risk and whether there are any alternatives. We manually inspect a random sample of 50 uses of `exec` and 100 of `eval` to identify recurring usage patterns. Our classification yields four and nine recurring patterns for `exec` and `eval`, respectively, illustrated in detail in the appendix.

Patterns of `exec` usage: The majority of calls (57%) trigger a single operating system command and pass a sequence of arguments to it. For these calls, the developers could easily switch to `spawn`, which is a safer API to use, equivalent to the well-known `execv` functions in C. The second-most common usage pattern (20%) involves multiple operating system commands combined using Unix-style pipes. For this pattern, we are not aware of a simple way to avoid the vulnerable `exec` call. The above-mentioned `spawn` accepts only one command, i.e., a developer would have to call it multiple times and emulate the shell's piping mechanism. Another interesting pattern is the execution of scripts using relative paths, which accounts for 10% of the analyzed cases. This pattern is frequently used as an ad-hoc parallelization mechanisms, by starting another instance of NODE.JS, and to interoperate with code written in another programming language.

Patterns of `eval` usage: Our results of the usage of `eval` mostly match those reported in a study of client-side JavaScript code [24], showing that their findings extend to NODE.JS JavaScript code. One usage pattern that was not previously reported is to dynamically create complex functions. This pattern, which we call "higher-order functions", is widely used

```
1 function escape(s) {
2   return s.replace(/"/g, '\\"');
3 }
4 exports.open = function open(target, callback) {
5   exec(opener + ' "' + escape(target) + '"');
6 }
7 // possible attack:
8 open("`rm -rf *`");
```

Fig. 4: Regex-based sanitization and input that bypasses it.

in server-side JavaScript for creating functions from both static strings and user-provided data. We are not aware of an existing technique to easily refactor this pattern into code that does not use `eval`.

Overall, we find that over 20% of all uses of injection APIs cannot be easily removed. Furthermore, we believe that many of the remaining uses are unlikely to be refactored by the developers. The reason is that even though several techniques for removing some usages of `eval` have been proposed years ago [19], [11] and the risks of this function are widely documented [4], [9], using `eval` unnecessarily is still widespread.

C. RQ3: Mitigation

The high prevalence of using APIs prone to injections raises the question how developers protected their code against such vulnerabilities. To address this question, we manually analyze the protection mechanisms used by the sample of API usages from RQ2. Specifically, we have analyzed (i) whether a call site of an injection API may be reached by attacker-controlled data, i.e., whether any mitigation is required, and (ii) if the call site requires mitigation, which technique the developers use.

We find that for 58% of the inspected call sites, it may be possible that attacker-controlled data reaches the injection API. Among these call sites, the following mitigation techniques are used:

- **None.** A staggering 90% of the call sites do not use any mitigation technique at all. For example, the call to `exec` in the motivating example in Figure 1 falls into this category.
- **Regular expressions.** For 9% of the call sites, the developers harden their module against injections using regular expression-based checks of input data. An example in our data set is shown in Figure 4. Unfortunately, most regular expressions we inspected are not correctly implemented and cannot protect against all possible injection attacks. For example, the escape method in the figure does not remove the back ticks characters allowing an attacker to deliver a malicious payload using the command substitution syntax, as illustrated in the last line of Figure 4. In general, regular expressions are fraught with danger when used for sanitization [10].
- **Sanitization modules.** To our surprise, none of the modules uses a third-party sanitization module to prevent injections. To validate whether any such modules exists,

we searched the npm repository and found six modules intended to protect calls to `exec` against command injections: shell-escape, escapeshellarg, command-join, shell-quote, bash, and any-shell-escape. In total, 198 other modules depend on one of these sanitization modules, i.e., only a small fraction compared to the 19,669 modules that directly or indirectly use `exec`. For `eval`, there is no standard solution for sanitization and the unanimous experts advice is to either not use it at all in combination with untrustworthy input, or to rely on well tested filters that allow only a restricted class of inputs, such as string literals or JSON data.

For the remaining 42% of call sites, it is impossible or very unlikely for an attacker to control the data passed to the injection APIs. For example, this data includes constant strings, IDs of operating system processes, and the path of the current working directory.

We conclude from these results that most modules are vulnerable to injections and that standard sanitization techniques are rarely used, despite some of the specialized modules available for this purpose. Developers are either unaware of the problem in the first place, unwilling to address it, or unable to properly apply existing solutions.

D. RQ4: Interactions with Module Maintainers

The mitigation techniques discussed for RQ3 depend on developers investing time and effort into avoiding vulnerabilities. To better understand to what extent developers are willing to invest that kind of effort, we reported 20 previously unknown command injection vulnerabilities to the developers of the respective modules that call the injection APIs. We have manually identified these vulnerabilities during the study described in this section. For each vulnerability, we describe the problem and provide an example attack to the project developers. Figure 5 summarizes our interactions with the developers. In total, we received eight responses. Most of the developers acknowledge the problem, and they want to fix it. However, in the course of several months, only three of the 20 vulnerabilities have been completely fixed. The majority of issues are still pending, showing the lack of maintenance for many of the modules involved.

One may hypothesize that these vulnerabilities are characteristic to unpopular modules that are not expected to be well maintained. We checked this hypothesis by measuring the number of downloads between January 1 and February 17, 2016 for three sets of modules: (i) modules with vulnerabilities reported either by us or by others via the Node Security Platform, (ii) all modules that call an injection API, (iii) all modules in the npm repository.

Figure 6 summarizes our results on a logarithmic scale. The boxes are drawn between the lower quartile (25%) and the upper one (75%) and the horizontal line marks the median. The results invalidate the hypothesis that vulnerable modules are unpopular. On the contrary, we observe that various vulnerable modules and injection modules are highly popular, exposing millions of users to the risk of injections.

E. Case Study: The growl Module

To better understand whether developers are aware of possible injection vulnerabilities in modules that they use, we manually analyzed 100 modules that depend on growl. The growl module displays notifications to users by invoking a particular command via `exec`, which is one of the vulnerabilities we reported as part of RQ4 (see last entry in Figure 5). We found that modules depending on growl pass various kinds of data to growl, including error messages and data extracted from web pages. As anticipated in RQ1, vulnerabilities propagate along module dependences. For example, the loggy module exposes the command injection vulnerability in growl to 15 other modules that depend on loggy by sending inputs directly to growl without any check or sanitization.

We found only four modules that sanitize the data before sending it to the vulnerable module: mqtt-growl, chook-growl-reporter, bungle, and autolint. We report these sanitizers in Figure 7. Sadly, we find that all these methods are insufficient: one can easily bypass them, as illustrated by the example input at the end of Figure 7. The input again exploits the command substitution syntax, which is not considered by any of the sanitizers.

IV. Overview

The previous section shows that the risk of injection vulnerabilities is widespread, and that a practical technique to mitigate them must support module maintainers who are not particularly responsive. Motivated by these findings, this section presents SYNODE, a novel mitigation approach that combines static analysis and runtime enforcement into a fully automatic approach to prevent injection attacks. To the best of our knowledge, our approach is the first to address the problem of injection vulnerabilities in NODE.JS JavaScript code.

The overall idea of the mitigation technique is to prevent injections at the call sites of injection APIs. Figure 8 shows an overview of the approach. Given a potentially vulnerable JavaScript module, a static analysis summarizes the values that may flow into injection APIs in the form of string templates, or short *templates*. A template is a sequence consisting of string constants and holes to be filled with untrusted runtime data. For call sites where the analysis can statically show that no untrusted data may flow into the injection API, no further action is required to ensure safe runtime behavior.

For the remaining call sites, the approach synthesizes a runtime check and statically rewrites the source code to perform this check before untrusted data reaches the injection API. When executing the module, the rewritten code enforces a security policy that checks the runtime values to be filled into the holes of the template against the statically extracted template. If this check fails, the program is terminated to prevent an injection attack.

V. Static Analysis

We present a static analysis of values passed to injection APIs. For each call site of such an API, the analysis summarizes all values that may be passed to the called function into a

API	Reported	Affected module	Confirmed	Fixed	Reference
`eval`	Apr 7	mixin-pro	yes	Apr 7	github.com/floatinghotpot/mixin-pro/issues/1
`eval`	Apr 7	modulify	no	–	github.com/matthewkastor/modulify/issues/2
`eval`	Apr 7	proto	yes	Sep 9 *	github.com/milojs/proto/issues/1
`eval`	Apr 8	mongoosify	yes	Jun 20	github.com/nanachimi/mongoosify/issues/1
`eval`	Apr 8	summit	yes	–	github.com/notduncansmith/summit/issues/23
`eval`	Apr 8	microservicebus.node	yes	–	github.com/microServiceBus/microservicebus.node/issues/9
`eval`	Apr 8	mobile-icon-resizer	yes	Apr 9	github.com/muzzley/mobile-icon-resizer/issues/8
`eval`	Apr 4	m-log	–	–	github.com/m-prj/m-log/pull/1
`eval`	Apr 8	mongo-edit	–	–	github.com/louischatriot/mongo-edit/issues/18
`eval`	Apr 8	mongo-parse	yes	–	github.com/fresheneesz/mongo-parse/issues/7
`eval`	Apr 8	mock2easy	–	–	github.com/appLhui/mock2easy/issues/2
`eval`	Apr 8	mongui	–	–	github.com/jjtortosa/mongui/issues/1
`eval`	Apr 8	m2m-supervisor	–	–	github.com/numerex/m2m-supervisor/issues/1
`eval`	Apr 8	nd-validator	–	–	github.com/ndfront/nd-validator/issues/4
`eval`	Apr 8	nameless-cli	–	–	github.com/StarmanMartin/nameless-cli/issues/2
`eval`	Apr 8	node-mypeople	–	–	github.com/hackrslab/mypeople/issues/2
`eval`	Apr 8	mongoosemask	–	–	github.com/mccormicka/MongooseMask/issues/1
`eval`	Apr 7	kmc	–	–	github.com/daxingplay/kmc/issues/54
`eval`	Apr 7	mod	–	–	github.com/modjs/mod/issues/82
`exec`	Jul 21	growl	yes	–	github.com/tj/node-growl/issues/60

Fig. 5: Injection vulnerabilities that we reported to the developers. A dash (–) indicates that the developers have not reacted to our report. All dates are in 2016. The fix marked with * is incomplete.

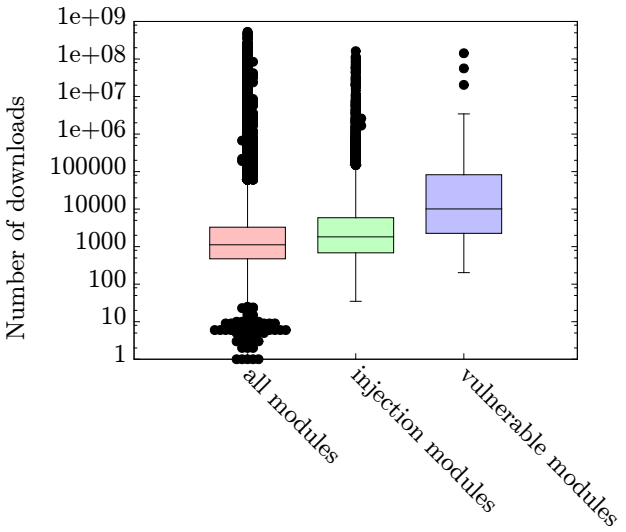


Fig. 6: Comparison of the popularity of all the modules, modules with calls to injection APIs, and modules with reported vulnerabilities. The boxes indicate the lower quartile (25%) and the upper quartile (75%); the horizontal line marks the median.

```
// sanitization autolint
function escape(text) {
  return text.replace('$', '\\$');
}

// sanitization mqtt-growl
message = message.replace(/"/g, "\\\"");

// sanitization bungle
const ansiRx =
  /[\u001b\u009b][[()#;?]*
  (?:[0-9]{1,4}(?:;[0-9]{0,4})*)?
  [0-9A-ORZcf-nqry=><]/g;
Growl(message.replace(ansiRx, ''));

// sanitization chook-growl-reporter
function escapeForGrowl(text) {
  var escaped = text.replace(/\(/g, '\\(');
  escaped = escaped.replace(/\)/g, '\\)');
  escaped = escaped.replace(/\""/g, '\\""');
  return escaped;
}

// input that bypasses all the sanitizations
input = "tst`rm -rf *`";
```

Fig. 7: Broken sanitization in growl's clients.

tree representation (Section V-A). Then, the analysis statically evaluates the tree to obtain a set of templates, which represent the statically known and unknown parts of the possible string values passed to the function (Section V-B). Finally, based on the templates, the analysis decides for each call site of an injection API whether it is statically safe or whether to insert a runtime check that prevents malicious strings from reaching the API (Section V-C).

A. Extracting Template Trees

The analysis is a flow-sensitive, path-insensitive, intra-procedural, backward data flow analysis. Starting from a call site of an injection API, the analysis propagates information about the possible values of the string argument passed to the API call along inverse control flow edges. The propagated information is a tree that represents the current knowledge of the analysis about the value:

Definition 1 (Template tree)**.** *A template tree is an acyclic,*

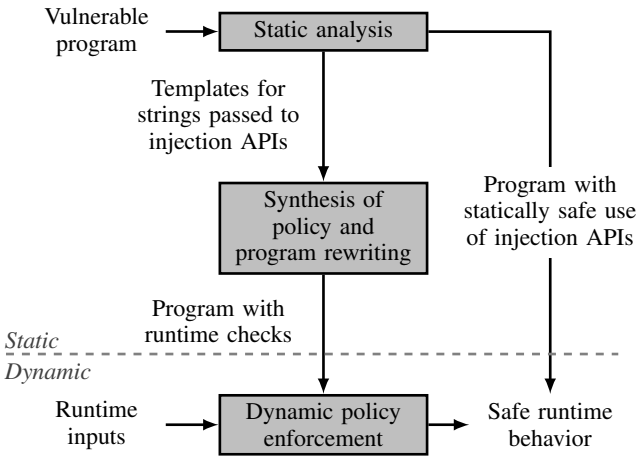
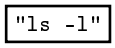
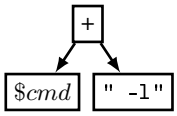


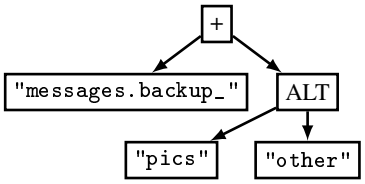
Fig. 8: Architectural diagram for mitigating injection attacks.



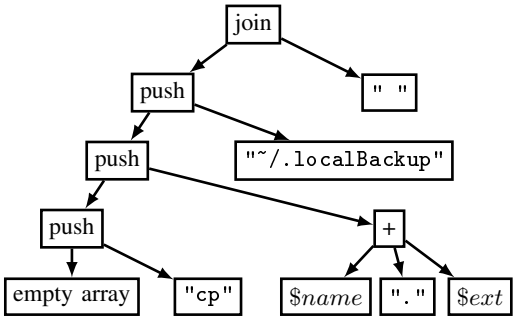
(a) Constant string.



(b) Concatenation of variable and constant.



(c) Argument of `eval` in Figure 1.



(d) Argument of `exec` in Figure 1.

Fig. 9: Examples of template trees.

connected, directed graph $(\mathcal{N}, \mathcal{E})$ where

- *a node $n \in \mathcal{N}$ represents a string constant, a symbolic variable, an operator, or an alternative, and*
- *an edge $e \in \mathcal{E}$ represents a nesting relationship between two nodes.*

Figure 9 shows several examples of template trees:

- Example (a) represents a value known to be a string constant `"ls -l"`. The template tree consist of a single node labeled with this string.
- In example (b), the analysis knows that the value is the result of concatenating the value of a symbolic variable $\$cmd$ and the string constant `" -l"`. The root node of the template tree is a concatenation operator, which has two child nodes: a symbolic variable node and a string constant node.
- Example (c) shows the tree that the analysis extracts for the values that may be passed to `eval` at line 10 of Figure 1. Because the value depends on the condition checked at line 9, the tree has an alternative node with children that represent the two possible string values.
- Finally, example (d) is the tree extracted for the value passed to `exec` at line 7 of Figure 1. This tree contains several operation nodes that represent the push operations and the string concatenation that are used to construct the string value, as well as several symbolic variable nodes and string constant nodes.

Example	Templates
(a)	$t_{a1} = [$`"ls -l"`$]$
(b)	$t_{b1} = [\$cmd,$ `" -l"`$]$
(c)	$t_{c1} = [$`"messages.backup_pics"`$]$
	$t_{c2} = [$`"messages.backup_other"`$]$
(d)	$t_{d1} = [$`"cp "`$, \$name,$ `"."`$, \$ext, ,$ `" ~/.localBackup"`$]$

Fig. 10: Evaluated templates for the examples in Figure 9.

To extract such templates trees automatically, we use a data flow analysis [21], [1], which propagates template trees through the program. Starting at a call site of an injection API with an empty tree, the analysis applies the following transfer functions:

- *Constants.* Reading a string constant yields a node that represents the value of the constant.
- *Variables.* A read of a local variable or a function parameter yields a node that represents a symbolic variable.
- *Operations.* Applying an operation, such as concatenating two strings with `+`, yields a tree where the root node represents the operator and its children represent the operands.
- *Calls.* A call of a function yields a tree where the root node represents the called function and its children represent the base object and arguments of the call.
- *Assignments.* An assignment of the form $lhs = rhs$ transforms the current tree by replacing any occurrence of the symbolic variable that corresponds to lhs by the tree that results from applying the transition function to rhs.

Whenever the backward control flow merges, the analysis merges the two template trees of the merged branches. The merge operation inserts an alternative node that has the two merged trees as its children. To avoid duplicating subtrees with identical information, the analysis traverses the two given

trees t_1 and t_2 to find the smallest pair of subtrees t'_1 and t'_2 that contain all differences between t_1 and t_2, and then inserts the alternative node as the parent of t'_1 and t'_2.

Example 1 For example, consider the call site of `eval` at line 10 of Figure 1. Starting from an empty tree, the analysis replaces the empty tree with a tree that represents the string concatenation at line 10. One child of this tree is a variable node that represents the variable `kind`, which has an unknown value at this point. Then, the analysis reasons backwards and follows the two control flow paths that assign `"pics"` and `"other"` to the variable `kind`, respectively. For each path, the analysis updates the respective tree by replacing the variable node for `kind` with the now known string constant. Finally, the variable reaches the merge point of the backward control flow and merges the two trees by inserting an alternative node, which yields the tree in Figure 9c. □

B. Evaluating Template Trees

Based on the template trees extracted by the backward data flow analysis, the second step of the static analysis is to evaluate the tree for each call site of an injection API. The result of this evaluation process is a set of templates:

Definition 2 (Template)**.** *A template is a sequence* $t = [c_1, \ldots, c_k]$ *where each* c_i *represents either a constant string or an unknown value (hole).*

For example, the template trees in Figure 9 are evaluated to the templates in Figure 10. To obtain the templates for a given tree, the analysis iteratively evaluates subtrees in a bottom-up way until reaching the root node. The evaluation replaces operation nodes that have a known semantics with the result of the operation. Our implementation currently models the semantics of string concatenation, `Array.push`, `Array.join`, and `String.replace` where the arguments are constant strings. These operations cover most templates trees that the analysis extracts from real-world JavaScript code (Section VIII-A). For alternative nodes, the evaluation considers both cases separately, duplicating the number of template trees that result from the evaluation.

Finally, the analysis transforms each evaluated tree into a template by joining continuous sequences of characters into constant strings and by representing all symbolic values and unknown operations between these constants as unknown values.

C. Identifying Statically Safe Calls

After evaluating template trees, the analysis knows for each call site of an injection API the set of templates that represent the string values passed to the call. If all templates for a particular call site are constant strings, i.e., there are no unknown parts in the template, then the analysis concludes that the call site is statically safe. For such statically safe call sites, no runtime checking is required. In contrast, the analysis cannot statically ensure the absence of injections if the templates for the call site contain unknown values. In this case, checking is deferred to runtime, as explained in Section VI.

For our running example, the analysis determines that the `eval` call site at line 10 of Figure 1 is statically safe because both possible values passed to the function are known. In contrast, parts of the strings that may be passed to `exec` at line 7 are unknown and therefore the check whether an injection happens is deferred to runtime.

VI. Dynamic Enforcement

For call sites where the values passed to the injection API cannot be statically determined, we provide a dynamic enforcement mechanism. The goal of this mechanism is to reject values found to be dangerous according to a policy. Intuitively, we want to prevent values that expand the template computed for the call site in a way that is likely to be unforeseen by the developer. Our approach achieves this goal in two steps:

1) Before executing the module, the approach transforms the statically extracted set of templates for a call site into a set of partial abstract syntax trees (PAST) that represents the expected structure of benign values. The trees are partial because the unknown parts of the template are represented as unknown subtrees.
2) While executing the module, the approach parses the runtime value passed to an injection API into an AST and compares the PASTs from step 1 against the AST. The runtime mechanism enforces a policy that ensures that the runtime AST is (i) derivable from at least one of the PASTs by expanding the unknown subtrees and (ii) these expansions remain within an allowed subset of all possible AST nodes.

The following two subsections present the two steps of the dynamic enforcement mechanism in detail.

A. Synthesizing a Tree-based Policy

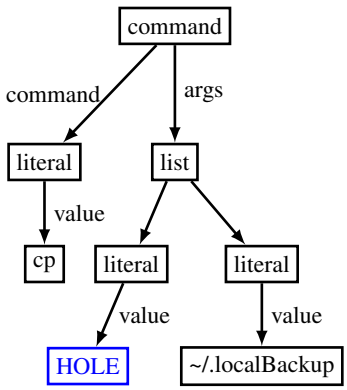
The goal of the first step is to synthesize for each call site a set of trees that represents the benign values that may be passed to the injection API. Formally, we define this trees as follows:

Definition 3 (Partial AST)**.** *The partial AST (PAST) for a template of an injection API call site is an acyclic, connected, directed graph* $(\mathcal{N}, \mathcal{E})$ *where*

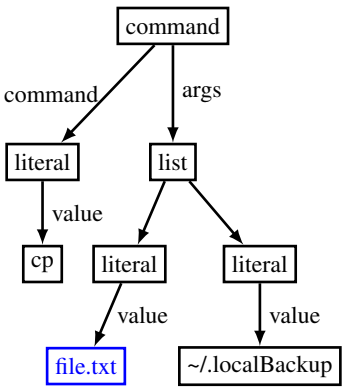
- $\mathcal{N}_{sub} \subseteq \mathcal{N}$ *is a set of nodes that each represent a subtree of which only the root node* $n_{sub} \in \mathcal{N}_{sub}$ *is known, and*
- $(\mathcal{N}, \mathcal{E})$ *is a tree that can be expanded into a valid AST of the language accepted by the API.*

For example, Figure 11a shows the PAST for the template t_{d1} from Figure 10. For this partial tree, $\mathcal{N}_{sub} = \{\text{HOLE}\}$, i.e., the hole node can be further expanded, but all the other nodes are fixed.

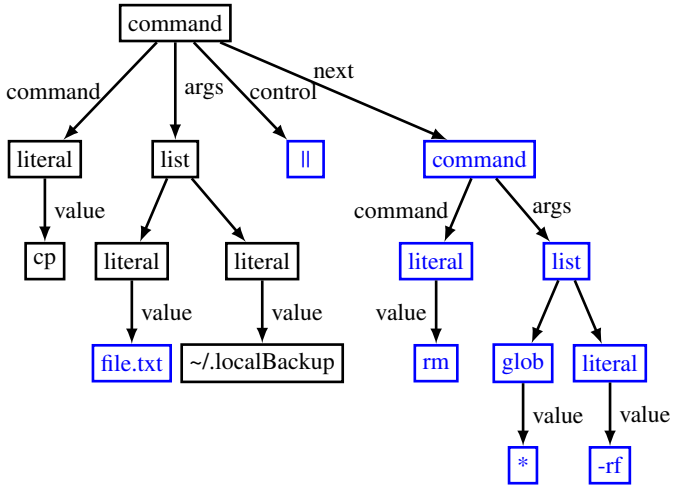
To synthesize the PAST for a template, the approach performs the following steps. At first, it instantiates the template by filling its unknown parts with simple string values known to be benign. The set of *known benign values* must be defined only once for each injection API. Figure 12 shows the set of values we use for `exec` and `eval`, respectively. The approach



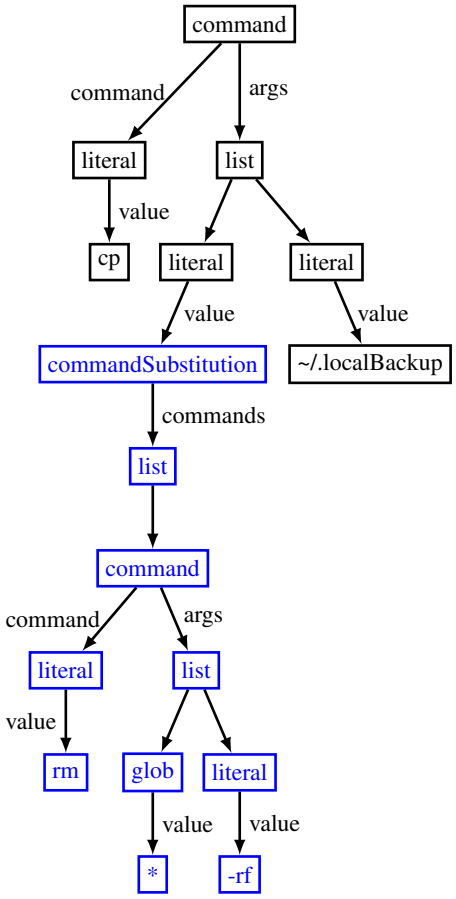
(a) Partial AST for the template in Figure 10.



(b) AST accepted by the policy derived from the partial AST.



(c) AST rejected by the policy derived from the partial AST.



(d) AST rejected by the policy derived from the partial AST.

Fig. 11: A partial AST and three ASTs compared against it. The blue nodes are holes and runtime values filled into the holes at runtime.

API	Language	Known benign values
`exec`	Bash	`"./file.txt"`, `"ls"`
`eval`	JavaScript	`x`, `y`, `"x"`, `x.p`, `{x:23}`, `23`

Fig. 12: Known benign values used to synthesize PASTs.

exhaustively tries all possible assignments of these values to the unknown parts of a template. Then, each of the resulting strings is given to a parser of the language, e.g., a JavaScript or Bash parser. If and only if the string is accepted as a legal member of the language, then the approach stores the resulting AST into a set of *legal example ASTs*.

Given the legal example ASTs for a template, the next step is to merge all of them into a single PAST. To this end, the approach identifies the least common nodes of all ASTs, i.e., nodes that are shared by all ASTs but that have a subtree that differs across the ASTs. At first, the given ASTs are aligned by their root nodes, which must match because all ASTs belong to the same language. Then, the approach simultaneously traverses all ASTs in a depth-first manner and searches for nodes n_{lc} with children that differ across at least two of the ASTs. Each such node n_{lc} is a least common node. Finally, the approach creates a single PAST that contains the common parts of all ASTs and where the least common nodes remain unexpanded and form the set $\mathcal{N}_{sub}$ (Definition 3). Note that $\mathcal{N}_{sub}$ is effectively an under-approximation of the possible valid inputs, given that we construct it using a small number of known benign inputs. However, in practice we do not observe any downsides to this approach, as discussed in Section VIII.

Example 2 For example, for the template t_{d1} and the known benign inputs for Bash in Figure 12, the first argument passed to `cp` will be expanded to the values `./file.txt.ls`, `ls.ls`, `./file.txt../file.txt` and `ls../file.txt`. All these values are valid literals according to the Bash grammar, i.e., we obtain four legal example ASTs. By merging these ASTs, the

approach obtains the PAST in Figure 11a because the only variations observed across the four ASTs are in the value of the literal. □

B. Checking Runtime Values Against the Policy

The set of PASTs synthesized for a call site is the basis of a policy that our mechanism enforces for each string passed at the call site. We implement this enforcement by rewriting the underlying JavaScript code at the call site. When a runtime value reaches the rewritten call site, then the runtime mechanism parses it into an AST and compares it with the PASTs of the call site. During this comparison, the policy enforces two properties:

- **P1**: The runtime value must be a syntactically valid expansion of any of the available PASTs. Such an expansion assigns to each node $n_{sub} \in \mathcal{N}_{sub}$ a subtree so that the resulting tree (i) is legal according to the language and (ii) structurally matches the runtime value's AST.
- **P2**: The expansion of a node n_{sub} of the PAST is restricted to contain only AST nodes from a pre-defined set of *safe node types*. The set of safe node types is defined once per language, i.e., it is independent of the specific call site and its PASTs. For shell commands passed to `exec`, the approach considers only nodes that represent literals as safe. For JavaScript code passed to `eval`, the approach allows all AST node types that occur in JSON code, i.e., literals, identifiers, properties, array expressions, object expressions, member expressions, and expression statements. Importantly, none of these nodes types enables an attacker to inject code that has side effects.

Example 3 To illustrate these properties, suppose that the three example inputs in Figure 13 are given to the `backupFile` function in Figure 1. Input 1 uses the function as expected by the developer. In contrast, inputs 2 and 3 exploit the vulnerability in the call to `exec` by passing data that will cause an additional command to be executed.

Figure 11 shows the PAST derived (only one because there is only one template available for this call site) for the vulnerable call site and the ASTs of the three example inputs. Input 1 fulfills both P1 and P2 and the value is accepted. In contrast, the policy rejects input 2 because it does not fulfill P1. The reason is that the AST of the input (Figure 11c) does not structurally match the PAST. Likewise, the policy rejects input 3 because it fulfills P1 but not P2. The reason for not fulfilling P2 is that the expanded subtree (i.e., the highlighted nodes in Figure 11d) contain nodes that are not in the set of safe node types. □

To summarize, the enforced policy can be formalized as follows:

Definition 4 (Security Policy)**.** *Given a runtime value v, a set of PASTs $\mathcal{T}$, and a set $\mathcal{N}_{safe}$ of safe node types, v is* rejected *unless there exists an expansion t' of some $t \in \mathcal{T}$, where*

- *t' is isomorphic to the AST of v, and*
- *let $\mathcal{N}_{input}$ be the set of nodes that belong to a subtree in the AST of v that matches a node in $\mathcal{N}_{sub}$, then the node type of all $n \in \mathcal{N}_{input}$ is in $\mathcal{N}_{safe}$.*

ID	name	ext	Property	
			P1	P2
1	`file`	`txt`	✓	✓
2	`file`	`txt || rm * -rf`	✗	–
3	`file`	`$(rm * -rf)`	✓	✗

Fig. 13: Inputs compared against the partial AST in Figure 11a.

Our runtime enforcement approach can be applied to any kind of injection API that expects string values specified by a context-free grammar. The effectiveness of the enforcement depends on two language-specific ingredients: the set of benign example inputs and the set of safe AST node types. Given that we are primarily interested in `eval` and `exec` sinks, we have created these ingredients for JavaScript and Bash, and Section VIII-B shows both to be effective for real-world NODE.JS code.

VII. IMPLEMENTATION

Static analysis: We implement the static analysis in Java, building upon the Google Closure Compiler[4]. To handle loops and recursion, the static data flow analysis limits the number of times a statement is revisited while computing a particular data flow fact to ten. When applying the static analysis to a module, we impose a one minute timeout per module. After finishing the analysis of a module, the implementation writes the set of templates for each call sites into a text file to be used by the dynamic analysis.

Runtime analysis: We implement the dynamic analysis in JavaScript. Before executing the module, the analysis precomputes the PASTs for each call site based on the templates gathered by the static analysis. While executing a module, the analysis intercepts all calls to `exec` and `eval` and extracts the strings passed to these function to be checked against our policy. To parse strings given to `exec` and `eval`, we build upon the esprima[5] and shell-parse[6] modules.

Automatic deployment: As shown by our study (Section III), the practical benefits of a technique to prevent injection attacks depend on how seamlessly the technique can be deployed. A particular challenge is how to apply a mitigation technique to code written by third parties that may not be willing to modify their code. To make the deployment of SYNODE as easy as possible without relying on the cooperation of third-party code providers, we advocate an approach in which a module developer or a system administrator adds a post-installation script[7] to the application packaged as an npm module.

[4] https://developers.google.com/closure/
[5] http://esprima.org/
[6] https://www.npmjs.com/package/shell-parse
[7] https://docs.npmjs.com/misc/scripts

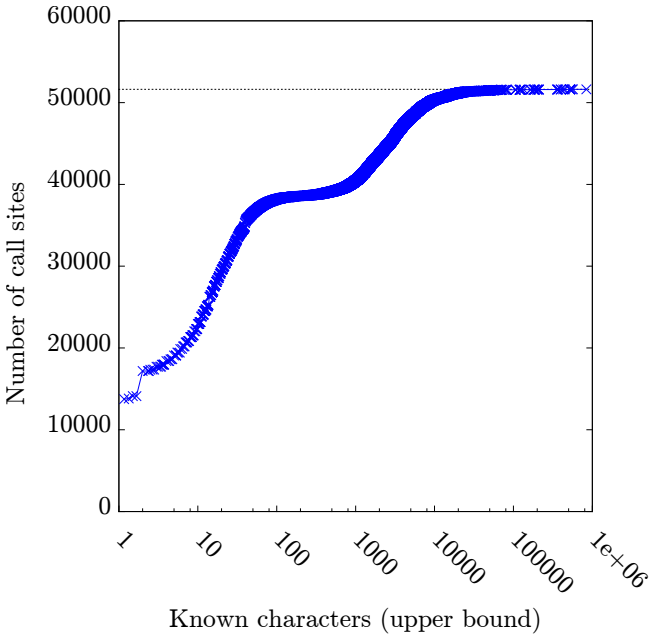


Fig. 14: CDF for the average number of constant characters per call site. Note the logarithmic horizontal axis.

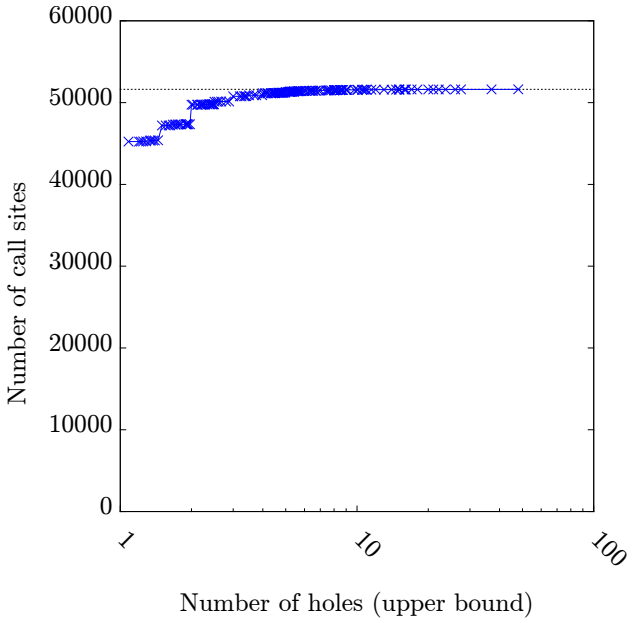


Fig. 15: CDF for the average number of holes per call site. Note the logarithmic horizontal axis.

The script runs on each explicitly declared third-party dependent module and, if necessary, performs the code rewriting step that adds dynamic enforcement at each statically unsafe call site of an injection API. As a result, our technique to prevent injection attacks can be deployed with very little effort and without requiring any knowledge about third-party code.

VIII. Evaluation

We evaluate our mitigation technique by applying it to all 235,850 Node.js modules. To avoid analyzing modules without any injection call sites, we filter modules by searching for call sites of these methods and include all 16,795 modules with at least one such call site in our evaluation. Since evaluating the runtime mechanism requires inputs that exercise the modules, we consider a subset of the modules, with known vulnerabilities, found by others or by us during the study (Section III).

A. Static Analysis

Statically safe call sites: The static analysis finds 18,924 of all 51,627 call sites (36.66%) of injection APIs to be statically safe. That is, the values that are possibly passed to each of these call sites are statically known, and an attacker cannot modify them. To further illustrate this point, Figure 16 shows to what extent the analysis can evaluate trees into templates. For 31.05% and 39.29% of all call sites of `exec` and `eval`, respectively, the template tree contains only constant nodes, operators supported by the analysis, and alternative nodes, which yield constant strings after evaluating the tree. The remaining template trees also contain symbolic variable nodes. Most of these trees (49.02% and 34.52%) are fully evaluated by the analysis, i.e., they contain no unsupported operators. It is important to note that the static analysis may provide a useful template even if the template tree contains an unsupported operation. The reason is that the other nodes in the tree often provide enough context around the unknown part created by the unsupported operation.

Kind of template tree	Number of call sites	
	`exec`	`eval`
Evaluates to constant string without holes	31.05%	39.29%
Holes due to symbolic variables only	49.02%	34.52%
Holes due to unsupported operations	19.93%	26.19%

Fig. 16: Kinds of template trees extracted by the static analysis.

Context encoded in templates: To better understand the templates extracted by the static analysis, we measure how much context about the passed string the static analysis extracts. First, we measure for each call site how many known characters are present per template, on average. Figure 14 shows that the templates for the majority of call sites contain at least 10 known characters. For some call sites, several thousands of characters are known, e.g., for calls of `eval` that evaluate a large piece of JavaScript code stored in a variable. For 10,967 call sites (21.24%), there is no known character, i.e., our approach relies entirely on dynamic information.

Second, we measure how many unknown parts the extracted templates contain. As shown in Figure 15, the templates for the vast majority of call sites has at most one hole, and very few templates contain more than five holes. The main reason for templates with a relatively large number of holes is that

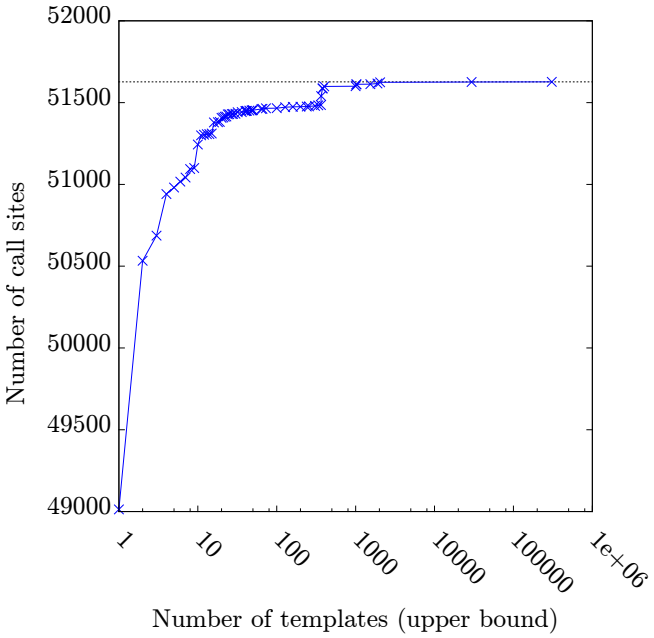


Fig. 17: CDF for the number of inferred templates per call site. Note the logarithmic horizontal axis.

the string passed to injection API is constructed in a loop that appends unknown values to the final string. The static analysis unrolls such loops a finite number of times, creating a relatively large number of unknown parts.

Third, we measure how many templates the analysis extracts per call site. Because different executed paths may cause different string values to be passed at a particular call site of an injection API, the analysis may yield multiple templates for a single call site. Figure 17 shows that for most call sites, a single template is extracted.

Reasons for imprecision: To better understand the reasons for imprecision of the static analysis, we measure how frequent particular kinds of nodes in template trees are. We find that 17.48% of all call sites have a template tree with at least one node that represent a function parameter. This result suggests that an inter-procedural static analysis might collect even more context than our current analysis. To check whether the static analysis may miss some sanitization-related operations, we measure how many of the nodes correspond to string operations that are not modelled by the analysis and to calls of functions whose name contains "escape", "quote", or "sanitize". We find that these nodes appear at only 3.03% of all call sites. The low prevalence of such nodes, reiterates the observation we made during our study: An npm module that uses sanitization when calling an injection API is the exception, rather than the rule.

Analysis running time: Our analysis successfully completes for 96.27% of the 16,795 modules without hitting the one-minute timeout after which we stop the analysis of a module. The average analysis time for these modules is 4.38 seconds, showing the ability of our approach to analyze real-world code at a very low cost.

We conclude from these results that the static analysis is effective for the large majority of call sites of injection APIs. Either the analysis successfully shows a call site to receive only statically known values, or it finds enough context to yield a meaningful security policy to be checked at runtime. This finding confirms our design decision to use a scalable, intra-procedural analysis. The main reason why this approach works well is because most strings passed to injection APIs are constructed locally and without any input-dependent path decisions.

B. Runtime Mechanism

For evaluating our runtime mechanism we consider a set of vulnerable modules listed in Figure 18. The set includes modules reported as vulnerable on the Node Security Platform, modules with vulnerabilities found during our study (Figure 5), and clients of known vulnerable modules.

Our static analysis identifies a total of 1,560 templates for the injection APIs in the considered modules. For each of them, we construct a PAST with a median computation time of 2 milliseconds per module. We note that for some modules this number is significantly higher due to our simple PAST construction algorithm and due to the high number of templates per module.

For each call site of an injection API in the benchmarks modules, we find module-level APIs that propagate data to the call sites of injection APIs. Figure 18 lists the modules and the type of injection vector we use. "Interface" means that we call the module via one of its exported APIs, "network" means that we pass data to the module via a network request, "file system" means that the module reads input data from a file, and "command line" means that we pass data as a command line argument to the module.

For each module, we created both benign and malicious inputs. As benign inputs, we use example usages provided in the documentation of the module, whenever possible. As malicious inputs, we carefully crafted payloads that will accomplish a specific goal. The goal for `eval` is to add a particular property to the globally available `console` object. For `exec`, the goal is to create a file in the file system. We manage to deliver for each module at least one command injection payload that achieves this goal, as summarized in Figure 18. In total, this experimental evaluation involves 56 benign inputs and 65 malicious inputs. Note that in some of the modules, we observe multiple benign values at the call site of the injection APIs, even though we provided only one such input to the module. The reason is that some values produced by the module may propagate to the injection APIs, independent of our crafted input.

False positives: Across the 121 inputs, we observe zero false negatives and five false positives. Three of our false positives are caused by limitations of our static analysis.

Example 4 For example, Figure 19 contains code that

Type		Benchmark module	Injection vector	Inputs benign	Inputs malicious	False negatives	False positives	Average overhead (ms)
Known advisories	(see II)	gm	interface	1	2	0	0	0.41
		libnotify	interface	4	2	0	1	0.19
		codem-transcode	network	1	4	0	0	0.80
		printer	interface	1	4	0	0	0.28
Reported by us	(see III-D)	mixin-pro	interface	2	4	0	0	0.16
		modulify	interface	1	2	0	1	0.04
		mol-proto	interface	1	2	0	1	0.07
		mongoosify	interface	1	2	0	0	0.04
		mobile-icon-resizer	file system	1	5	0	0	0.39
		m-log	interface	11	1	0	0	0.05
		mongo-parse	interface	1	2	0	0	0.11
		mongoosemask	interface	1	1	0	0	0.04
		mongui	network	1	2	0	0	0.05
		mongo-edit	network	1	1	0	0	0.04
		mock2easy	network	1	2	0	0	0.03
Case study	(see III-E))	growl	interface	1	2	0	0	2.72
		autolint	file system	4	4	0	0	1.59
		mqtt-growl	network	1	2	0	0	3.19
		chook-growl-reporter	interface	1	1	0	0	1.60
		bungle	file system	14	4	0	0	1.99
Other exec-based	(see III-B)	fish	interface	1	4	0	0	0.21
		git2json	interface	1	4	0	1	0.37
		kerb_request	interface	3	4	0	0	0.25
		keepass-dmenu	command line	1	4	0	1	0.52
		Total		56	65	0	5	Avg. 0.74

Fig. 18: Summary of results for runtime enforcement.

```
var keys = Object.keys(dmenuOpts);
var dmenuArgs = keys.map(function (flag) {
        return '-' + flag
            + ' "' + dmenuOpts[flag] + '"';
    }).join(' ');
var cmd = 'echo | dmenu -p "Password:" '
            + dmenuArgs;
exec(cmd);
```

Fig. 19: Example of a false positive.

constructs a command passed to `exec` by transforming an array `keys` of strings using `Array.map`. Because of the lack of static modeling of `Array.map`, our analysis assumes that the second part of `cmd` is unknown, leading to a PAST with a single unknown subtree. Our restrictive runtime policy allows to fill this subtree with only a single argument, and therefore rejects benign values of `dmenuArgs` that contain two arguments. □

The remaining two false positives are caused by our runtime mechanism which only allows safe AST nodes in the holes. This may be too restrictive in some situations, like in the case of the mol-proto module where `eval` is used to define arbitrary function objects whose body are specified by the users of the module. Overall, we conclude that the approach is effective at preventing injections while having a false positive rate that is reasonably low, in particular for a fully-automated technique.

Runtime overhead: The last column of Figure 18 shows the average runtime overhead per call of an injection API that is imposed by the runtime mechanism (in milliseconds). We report absolute times because the absolute overhead is more meaningful than normalizing it by a fairly arbitrary workload. Our enforcement mechanism costs 0.74 milliseconds per call, on average over 100 runs of the modules using all the inputs. This result demonstrates that the overhead of enforcement is generally negligible in practice.

Overhead as a function of the input size: To demonstrate the scalability of our runtime enforcement, we consider input data of different size and complexity and pass it to the injection APIs. Here, we focus on `eval` call sites from Figure 18 only. As inputs, we use a diverse sample of 200 JavaScript programs taken from a corpus of real-world code[8]. For every call to `eval`, we pass all 200 JavaScript programs 100 times each and measure the variance in enforcement times.

Figure 20 shows the enforcement time, in milliseconds, depending on the size of the JavaScript program, measured as the number of AST nodes. For each input size, the figure shows the 25% percentile, the median value, and the 75% percentile. We find that the enforcement time scales linearly. The reason is that all steps of the runtime enforcement, i.e., parsing the input, matching the AST with the PASTs, and checking whether nodes are on a whitelist, are of linear complexity.

[8] http://learnbigcode.github.io/datasets/

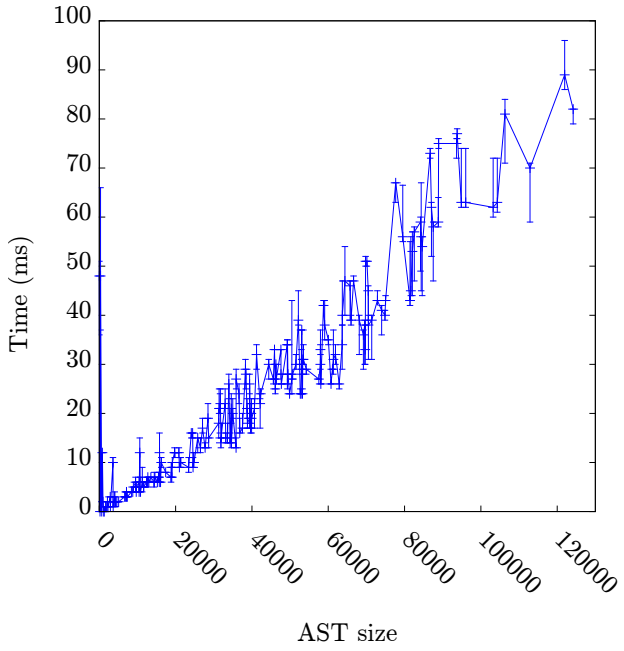


Fig. 20: Overhead of runtime checks depending on input size.

IX. Related Work

We restrict our focus to program analysis for security.

A. Analysis of Node.js Code

Injections into NODE.JS code are known to be exploitable [32] and there is a community-driven effort[9] to identify such problems. Ojamaa and Düüna [23] discuss several security problems of server-side JavaScript. They identify denial of service as one of the main threats for the NODE.JS platform and also mention `eval` as a security risks. We take these observation further by making an in-depth analysis of injection vulnerabilities on NODE.JS and present an automated technique to prevent injections.

NodeSentry [6] is a security architecture that supports least-privilege integration between NODE.JS modules, for limiting the power of third-parties libraries. The mechanism uses membranes to enforce security checks at different integration levels. Using the terminology introduced in the aforementioned work, our runtime enforcement can be seen as a lower-bound policy check on `exec` and `eval`. However, our mechanism is much more powerful than NodeSentry, integrating information collected by a static analysis to perform a fine-grained policy enforcement.

Madsen et al. [17] enhance the call graph construction for NODE.JS applications with event-based dependences. They show that their analysis can capture real life bugs related to event propagation which traditional static analysis can not. Even though the main focus of such a work is not security, it can enable the creation of more specialized techniques like taint analysis. The static analysis component of our work is intra-procedural, but it can benefit from integrating an inter-procedural approach, possibly one that considers events. This could potentially reduce the false positives rate as discussed in Section VIII-B.

[9] https://nodesecurity.io/advisories

B. Program Analysis for JavaScript

Empirical studies of client-side JavaScript code [36], [24] show that `eval` is prevalent in practice but often unnecessary. Our work shows that these findings extend to server-side JavaScript code and adds a new category of `eval` uses to the existing classification [24]. We also categorize uses of `exec`, an API not studied by existing work. Other studies of JavaScript code focus on inclusions of third-party libraries [22], the `postMessage` API [28], and injection attacks on JavaScript-based mobile applications [12]. Our work differs by studying NODE.JS code and by addressing vulnerabilities specific to this platform.

Blueprint [16] prevents XSS attacks by enforcing that the client-side DOM resembles a parse tree learned at the server-side, except for a small set of benign differences. Their work and ours shares the idea of comparing data prone to injections to a tree-based template. In contrast to Blueprint, we learn templates statically and focus on command injections in NODE.JS code.

Stock et al. [30] study DOM-based XSS injections and propose a technique to prevent them based on dynamic taint tracking. Similar to our runtime enforcement, their prevention policy is grammar-based. However, their strict policy to reject any tainted data that influences JavaScript code except for literals and JSON would break many of the usages of `eval` found during our study. Another difference is that we avoid the need for taint tracking by statically computing sink-specific templates.

Recent defenses against XSS attacks [29], [20] advocate signature-based whitelisting as a way to reject scripts not originating from the website creator. SICILIAN [29] uses an AST-based signature that computes a cryptographically secure hash function for the parts of the script that do not change. nsign [20] creates signatures that use script dependent elements (keywords frequency, URLs) and context based information (URL that triggered the script, stack trace). Both these techniques rely on a training phase in which the developer discovers valid signatures for the scripts, using testing. Our work is similar in the sense that we employ a set of templates as a whitelisting mechanism. However, we do not use testing to collect these templates, but rather compute them statically. As we showed in our evaluation, for some cases there are hundreds or even thousands of paths in which an injection API call site can be reached, therefore constructing valid signatures for every individual path would not be feasible.

DLint [5] is a dynamic checker to find violations of code quality rules, including uses of `eval` missed by static analyses. Dynamic [8] and hybrid (static and dynamic) [3] information flow analyses for JavaScript track how values at some program location influence values at another program location. The FLAX system [26] and a system proposed by Lekies et al. [14]

use dynamic taint analysis to detect vulnerabilities caused by missing input validation and then generate inputs that exploit these vulnerabilities. Jin et al. [12] use a static taint analysis to detect code injection vulnerabilities. In contrast to these approaches, we do not require an information flow (or taint) analysis, but instead perform lightweight runtime checks at possible injection locations, without dynamically tracking the values that flow into these locations.

Several approaches rewrite JavaScript code to enforce security policies. Yu et al. [35] propose a rewriting technique based on edit automata that replaces or modifies particular calls. Gatekeeper [7] is a sound static analysis to enforce security and reliability policies, such as to not call particular functions. Instead of conservatively preventing all possibly insecure behavior, our approach defers checks to runtime when hitting limitations of purely static analysis. Other techniques [11], [19] replace `eval` calls with simpler, faster, and safer alternatives. Their main goal is to enable more precise static analysis, whereas our focus is on preventing injections at runtime.

C. Program Analysis for Other Languages

CSAS [25] uses a type system to insert runtime checks that prevent injections into template-based code generators. Livshits et al. [15] propose to automatically place sanitizers into .NET server applications. Similar to our work, these approaches at first statically address some code locations and use runtime mechanisms only for the remaining ones. CSAS differs from our work by checking code generators instead of final code. The approach in [15] addresses the problem of placing generic sanitizers, whereas we insert runtime checks specific to an injection call site.

There are several purely dynamic approaches to prevent injections. XSS-Guard [2] modifies server applications to compute a shadow response along each actual response and compares both responses to detect unexpected, injected content. In contrast to their approach, which compares two strings with each other, our runtime mechanism compares runtime strings against statically extracted templates. ScriptGuard [27] learns during a training phase which sanitizers to use for particular program paths and detects incorrect sanitization by comparing executions against the behavior seen during training. Their approach is limited by the executions observed during training and needs to check all execution paths, whereas our approach statically identifies some locations as safe from injections. Su and Wassermann [31] formalize the problem of command injection attacks and propose a grammar-based dynamic analysis to prevent them. Their work shares the idea to reject runtime values based on a grammar that defines which parts of a string may be influenced by attacker-controlled values. Their analysis tracks input data with special marker characters, which may get lost on string operations, such as `substring`, leading to missed injections. Our analysis does not need to track input values through the program, but instead uses statically computed templates.

Constraint-based static string analysis, e.g., Z3-str [37] is a more heavy-weighted alternative to our static analysis. Even though such techniques have the potential of producing more precise templates, we opted for efficiency, enabling us to apply the analysis easily to thousands of npm modules.

Analyses by Wassermann et al. address the problem of finding program inputs that trigger SQL injections [34] and XSS vulnerabilities [33] in PHP code. Ardilla [13] finds and exploits injection vulnerabilities in PHP through a combination of taint analysis and test generation. Instead of triggering attacks, our work addresses the problem of preventing attacks. Similar to our preliminary study of dependences on injection APIs, a recent work analyzes the usage of the unsafe API in Java [18].

X. Conclusions

Our exploration of security issues in Node.js applications confirms that injection vulnerabilities are both present and not adequately addressed. This paper presents Synode, an automated technique for mitigating injection vulnerabilities in Node.js applications. To aid with its adoption, our techniques require virtually no involvement on the part of the developer. At the same time, it effectively prevents a range of attacks with very few false positives (false positive rate under 10%) and sub-millisecond overheads. We find that, despite the severity of these injection issues, Node.js developers are reluctant to use analysis tools on a consistent basis. To match the realities of the situation, our technique can be deployed automatically as part of module installation, further easing the process of deployment. Our work represents an important first step toward securing the increasingly important class of Node.js applications, and we hope it will inspire future work in this space.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, 2008.

[3] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Conference on Programming Language Design and Implementation*, pages 50–62. ACM, 2009.

[4] D. Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[5] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *International Symposium on Software Testing and Analysis*, pages 94–105, 2015.

[6] W. D. Groef, F. Massacci, and F. Piessens. NodeSentry: least-privilege library integration for server-side JavaScript. In *Annual Computer Security Applications Conference*, pages 446–455, 2014.

[7] S. Guarnieri and V. B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, pages 151–168, 2009.

[8] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing*, pages 1663–1671, 2014.

[9] D. Herman. *Effective JavaScript: 68 Specific ways to harness the power of JavaScript*. Addison-Wesley, 2013.

[10] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, pages 1–16, Aug. 2011.

[11] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis*, pages 34–44, 2012.

[12] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *Conference on Computer and Communications Security*, pages 66–77, 2014.

[13] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *International Conference on Software Engineering*, pages 199–209, 2009.

[14] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *Conference on Computer and Communications Security*, pages 1193–1204, 2013.

[15] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Symposium on Principles of Programming Languages*, pages 385–398, 2013.

[16] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Symposium on Security and Privacy*, pages 331–346, 2009.

[17] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 505–519, 2015.

[18] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: the Java unsafe API in the wild. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 695–710, 2015.

[19] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 607–620, 2012.

[20] D. Mitropoulos, K. Stroggylos, D. Spinellis, and A. D. Keromytis. How to train your browser: Preventing XSS attacks using contextual script fingerprints. *Transactions on Privacy and Security*, 19(1):2, 2016.

[21] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, second edition edition, 2005.

[22] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM Conference on Computer and Communications Security*, pages 736–747, 2012.

[23] A. Ojamaa and K. Düüna. Assessing the security of Node.js platform. In *International Conference for Internet Technology and Secured Transactions*, pages 348–355, 2012.

[24] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming*, pages 52–78, 2011.

[25] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Conference on Computer and Communications Security*, pages 587–600, 2011.

[26] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Network and Distributed System Security*, 2010.

[27] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *Conference on Computer and Communications Security*, pages 601–614, 2011.

[28] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *Network and Distributed System Security*, 2013.

[29] P. Soni, E. Budianto, and P. Saxena. The SICILIAN defense: Signature-based whitelisting of web JavaScript. In *Conference on Computer and Communications Security*, pages 1542–1557, 2015.

[30] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against DOM-based cross-site scripting. In *USENIX Security Symposium*, pages 655–670, 2014.

[31] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[32] B. Sullivan. Server-side JavaScript injection. *Black Hat USA*, 2011.

[33] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, pages 171–180, 2008.

[34] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *International Symposium on Software Testing and Analysis*, pages 249–260, 2008.

[35] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Symposium on Principles of Programming Languages*, pages 237–249, 2007.

[36] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *International Conference on World Wide Web*, pages 961–970, 2009.

[37] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *Joint Meeting on Foundations of Software Engineering*, pages 114–124, 2013.

APPENDIX

The following table lists recurring usage patterns of `exec` and `eval` that we found during the study described in Section III.

Type	Class	Description	Example	Perc.	Easy to refactor
`exec`	OS command	Run an installed command, either standard (e.g. grep, git) or custom.	`exec('curl -f -s -S --negotiate -u : ' + url)`	58%	yes
`exec`	Piped commands	Combine the multiple unix-like commands using pipes.	`var cmd = 'echo | dmenu -p "Password:" ' + dmenuArgs;` `return exec(cmd);`	20%	no

`exec`	Script	Execute a local script using a relative path.	`exec('./bin.js', error => {`
			`t.error(error, 'Should run without error.');`
			`t.end();`
			`});`
				10%	yes

`exec`	Terminal emulator	Run custom commands as they are provided by the user.	`exec(`
			`process.argv.slice(2)`
			`);`
				6%	yes

`eval`	Library	Load external JavaScript code.	`var _map = view.map ?`
			`eval('(' + view.map + ')')`
			`: undefined;`
				29%	yes

`eval`	JSON	Legacy way to parse JSON files.	`var body = eval(req.body);`	23%	yes

`eval`	Higher-order	Generates functions on-the-fly.	`opts.filter = 'function filter (doc) {';`
			`opts.filter +=`
			`opts.collection.map(function (c) {`
			`return 'if (doc.type === \''`
			`+ c`
			`+ '\') {return true;}';`
			`}).join('\n');`
			`opts.filter += 'return false;}';`
			`eval(opts.filter);`
				17%	no

`eval`	Read	Access an object's property.	`eval('originObj'`
			`+ generateObjectPath(value)`
			`+ '');`
				6%	yes

`eval`	Engine check	Use a specific functionality to check engine's support.	`eval('(function *(){})');`	6%	no

`eval`	Calculation	Use `eval` to compute mathematical expressions.	`eval('('`
			`+ this.executionTimes.join('+')`
			`+ ')/'`
			`+ this.executionTimes.length);`
				4%	yes

`eval`	Call	Call a given method.	`eval(_js + '=extend(' + _js + ',_o);');`	4%	yes

`eval`	Adhoc object creation	Generates code on the fly for convenience.	`var expression =`
			`eval(`
			`'/.*'`
			`+ param.toLowerCase()`
			`.replace(/\//gi, '\\\\')`
			`+ '.*/ig');`
				3%	no

`eval`	Write	Write an object's property.	`eval('theme = {' + i + ':[' + v + ']}');`	3%	yes