Abusing Hidden Properties to Attack the Node.js Ecosystem

Feng Xiao Jianwei Huang[†] Yichang Xiong[*] Guangliang Yang
Hong Hu[‡] Guofei Gu[†] Wenke Lee

GeorgiaTech [†]*Texas A&M* [‡]*PennState* [*]*Independent*

Abstract

Nowadays, Node.js has been widely used in the development of server-side and desktop programs (e.g., Skype), with its cross-platform and high-performance execution environment of JavaScript. In past years, it has been reported other dynamic programming languages (e.g., PHP and Ruby) are unsafe on sharing objects. However, this security risk is not well studied and understood in JavaScript and Node.js programs.

In this paper, we fill the gap by conducting the first systematic study on the communication process between client- and server-side code in Node.js programs. We extensively identify several new vulnerabilities in popular Node.js programs. To demonstrate their security implications, we design and develop a novel feasible attack, named hidden property abusing (HPA). Our further analysis shows HPA attacks are subtly different from existing findings regarding exploitation and attack effects. Through HPA attacks, a remote web attacker may obtain dangerous abilities, such as stealing confidential data, bypassing security checks, and launching DoS (Denial of Service) attacks.

To help Node.js developers vet their programs against HPA, we design a novel vulnerability detection and verification tool, named LYNX, that utilizes hybrid program analysis to automatically reveal HPA vulnerabilities and even synthesize exploits. We apply LYNX on a set of widely-used Node.js programs and identify 15 previously unknown vulnerabilities. We have reported all of our findings to the Node.js community. 10 of them have been assigned with CVE, and 8 of them are rated as "Critical" or "High" severity. This indicates HPA attacks can cause serious security threats.

1 Introduction

Node.js is a cross-platform and high-performance execution environment for JavaScript programs. It has been widely used to develop server-side and desktop applications such as Skype, Slack, and WhatsApp [7,16]. According to a recent study [17], Node.js is the most widely-used technology among all kinds of developments for three years (2017-2019).

The prominence of Node.js makes its security critical. Specifically, once a widely-used module is found to be vulnerable, a huge number of Node.js applications may be impacted due to the heavy reuse phenomenon [49]. By exploiting these vulnerabilities, remote attackers may abuse powerful and privileged APIs inside vulnerable server-side applications to launch severe attacks, like stealing confidential data or executing arbitrary malicious code [23,29,37,38,43,44,49].

Node.js programs are built in the dynamic programming language – JavaScript. In the past few years, several dynamic languages, like PHP [28] and Ruby [14], suffer from a common security risk CWE-915 [9], where an internal object attribute is improperly modified by untrusted user input. Despite the severe security consequence, this issue is not well studied and understood in JavaScript and Node.js programs.

In this paper, we conduct the first systematic study on the object sharing and communication process between client- and server-side code in Node.js programs. We confirm that the above security risk also exists in JavaScript and Node.js programs. To demonstrate the security implications, we design a novel attack, named *hidden property abusing* (HPA), that enables remote web attackers to obtain dangerous abilities, such as stealing confidential data, bypassing security checks, and launching denial-of-service attacks. Our further analysis shows HPA differs from existing findings on PHP [28] and Ruby [14] in many aspects such as exploitation and attack effects (see more details in §3.4).

An HPA attack example is shown in Figure 1. As the figure shows, a remote web attacker sends well-crafted JSON data with an extra and unexpected property "`I2`" (called *hidden property*) to the target Node.js server program. Then, the victim program deals with the malicious input payload as normal. Finally, `I2` propagates to an internal object. As indicated by the red line, `I2` of input overwrites and replaces a key property of the victim internal object with the conflicting name. Thus, the attacker may abuse the propagation process (i.e., property propagation) of a hidden property to powerfully manipulate critical program logic associated with the compromised property, such as directly calling privileged APIs by assigning `I2`

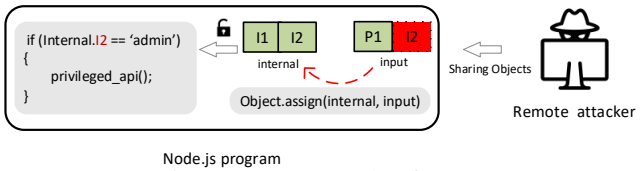

Figure 1: An example of HPA.

of input with the proper value (i.e., "admin").

Our analysis shows that the victim property can be of any type, such as critical functions or key program states. Due to this feature, input validation cannot stop attackers launching HPA attacks, as they may disable the validation logic by overwriting critical states or removing all security checks [24, 32]. We find this attack scenario is very common in practice.

To help Node.js developers detect and verify the emerging HPA issues in their Node.js applications and modules, we design and implement a vulnerability detection and verification tool, named LYNX[1]. LYNX combines the advantages of static and dynamic analysis to track property propagation, identify hidden properties, and generate corresponding concrete exploits for the verification purpose. We are releasing the source code of LYNX at https://github.com/xiaofen9/Lynx .

We evaluate LYNX by applying it on 102 real Node.js applications and modules widely used in practice. As a consequence, LYNX uncovered 15 previously unknown vulnerabilities. We have made responsible disclosure of the discovered vulnerabilities. By the time of paper writing, we have got 10 CVEs assigned; 8 of them are rated as critical or high severity by NVD (National Vulnerability Database); 7 vulnerabilities have been patched by their vendors. This indicates HPA attacks can cause serious security threats. We are collaborating with Node.js community to mitigate HPA. We first help an authoritative public vulnerability database create a new notion to describe the new type of vulnerabilities. In addition, we propose three potential HPA mitigation, with more details in §A.1.

In summary, we make the following contributions:

- We present the *hidden property abusing* attack against Node.js applications, and demonstrate its severe security consequences.
- We design and implement LYNX, a tool that automatically detects HPA issues and synthesizes exploits.
- Our evaluation reveals real-world HPA issues that can lead to serious security impacts.

2 Background

Node.js and its runtime engine. Node.js is used for executing JavaScript code outside of browsers. Many event-driving servers/middlewares and traditional web applications are deployed in Node.js. To interpret and execute JavaScript, Node.js implements a runtime engine based on Chrome's V8 JavaScript engine [19]. To satisfy the needs of server-side application scenarios, the engine provides a set of APIs to let JavaScript interact with host environment. With provided APIs, the JavaScript code can perform sensitive operations such as file operations.

However, Node.js does not enforce isolation to separate the application from host environment. Thus, serious security issues might be introduced if certain internal states of the Node.js application are compromised.

Object sharing. Most Node.js programs are deployed as web-based applications according to the official Node.js survey [1]. Similar to traditional web applications in other languages (e.g., PHP), network protocols like HTTP(S) and WebSockets are widely-used to exchange data between users and the application.

In the Node.js ecosystem, it is a common feature for applications to convert received data into an object (i.e., data serialization). With the help of this feature, Node.js applications can send/receive a very complex data structure. According to our investigation on npm, different programs are using distinct methods/code implementations to share objects. Currently, most programs share objects via JSON serialization or query-string serialization (more discussion in §4.4.1), while other channels may also be used such as HTTP headers (user-agent [18] and cookies [4]).

3 Hidden Property Abusing

In this section, we present the details of HPA attacks. First, we define our threat model. Next, we walk through a real-world example to demonstrate HPA. Then, we define the vulnerable behaviors and the associated attack vectors. In the end, we discussed the differences between HPA and other related attacks.

3.1 Threat Model

We assume that Node.js applications and modules are benign but vulnerable. In addition, we assume the target application correctly implements object sharing (i.e., data deserialization). In this setting, a remote web attacker aims to compromise the vulnerable server-side program using HPA. To exploit the vulnerability, the attacker sends a well-crafted payload to the victim application through the legitimate interfaces. When the malicious payload reaches the victim application, it is treated as normal data and dealt with as regular. Due to the lack of strict isolation between input and internal objects, the malicious payload is propagated to the internal objects of the vulnerable Node.js module. Finally, a critical internal object is corrupted and the attack is launched.

[1]The lynx is a type of wildcat. In Greek myths, it is believed that lynxes can see what others can't, and its role is revealing hidden truths.

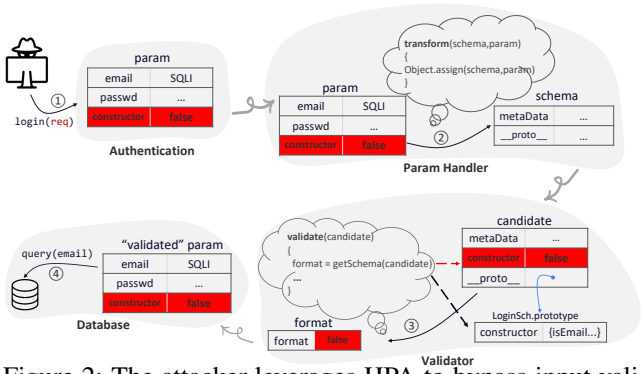

Figure 2: The attacker leverages HPA to bypass input validation and attack sensitive services behind (For illustration purpose, we use a database service as the attack target).

3.2 Running Example

To illustrate the HPA attack, we walk through a real-world exploit found in the high-profile Node.js framework "routing-controller" [13] (63,000+ monthly downloads on npm). In this example, we demonstrate although this vulnerable framework enforces a global input validation for unsafe external data, an attacker can still leverage HPA attacks to tamper its validation logic and introduce arbitrary malicious payloads.

Figure 2 shows the attack details. In the first step, the attacker adds an additional property (i.e., hidden property) `constructor:false` to the input object when accessing the authentication web API `login()` of the victim framework. Upon being called, the authentication module will instantiate an object named `param` and sends it to the parameter handler, which is responsible for validating user input. To this end, function `transform()` in the figure builds a validation candidate by merging `param` with the format specification object `schema`. As indicated in the second step, when building such a candidate, the hidden property `constructor:false` further propagates into the internal object `schema`.

The above propagation process enables the attacker to disable the input validation logic by hijacking the inheritance chain of `constructor`. In JavaScript, every object has a link to a prototype object. When the program wants to access a property of an object, the property will not only be searched on the object but on the prototype of the object, and even the prototype of the prototype, until a property with a matching name is found. As a result, every object has many inherited properties besides its own properties. However, such an inheritance chain can be hijacked if there is a conflicting name property locating at a higher level of the searching tree (Note that the hijacking process differs from prototype pollution [12]. More details will be discussed at §3.3). In the third step, function `validate()` checks all the properties within the candidate to see if the input object is legitimate or not. `validate` internally invokes function `getSchema()` to extract the format specification from `candidate`. However, because of the hijack, function `getSchema()` accesses the forged `constructor` (pointed by the red dashed line) rather than the real one (pointed by the black dashed line). As a result, the final format object used for validation is controlled by the attacker through the hidden property. To bypass the input validation, the attacker only needs to set `format` to an invalid value such as false. Finally, as indicated in the fourth step, the attacker can let a malicious `email` pass the validation and further performs SQL Injection attacks against the database module.

3.3 Attack Vectors

As demonstrated in §3.2, a remote attacker can propagate a hidden property to tamper certain internal states. In general, there are two typical attack vectors. The first one is called app-specific attribute manipulation, which involves tampering certain internal properties defined by the application developers. The second one is prototype inheritance hijacking, which hijacks the prototype inheritance chain. It is worth noting that our second attack vector is different from existing attacks, like prototype pollution [12]. Prototype pollution requires the modification of the prototype. However, as shown in the running example, the attacker of HPA does not need to tamper the prototype.

App-specific attribute manipulation. This attack vector targets the vulnerable code that falsely exposes certain app-specific attributes (e.g., access right) to a user-controlled object. As shown in Figure 1, the `I2` property is supposed to be initialized and managed by internal functions. However, with HPA, attackers might propagate a same-name property to the internal object, and thus access sensitive APIs. This attack vector can be used to abuse certain service such as order status in large applications.

Prototype inheritance hijacking. This vector hijacks the prototype inheritance chain so that the attacker can trick the vulnerable program into referencing a user-controlled property rather than the one inherited from the prototype. With this vector, attackers may forge many built-in properties, and even nested prototype properties (Two of our discovered vulnerabilities are exploited using nested properties). In our running example in §3.2, attackers forge `constructor`. If necessary, they can also forge other prototype properties such as `constructor.name`. This vector is very useful because many JavaScript developers tend to trust properties inherited from prototype and make many security-sensitive decisions based on them.

3.4 Comparing HPA with related attacks

The risks of improper modification of dynamic object attributes (CWE-915) have been identified in some dynamic languages such as Ruby and PHP. We are the first to identify such risks in Node.js. Moreover, we find HPA differs from existing vulnerabilities in multiple aspects.

Table 1: Comparing HPA and Ruby mass assignment.

Aspect	Hidden Property Abusing	Ruby Mass Assignment
Abused logics	Object sharing	Assignment
Payload Type	Literal value/nested object	Literal value
Capabilities	Overwrite	Overwrite/Create

Table 1 summarizes the difference between HPA and Ruby mass assignment, a typical vulnerability resulting from CWE-915. First of all, they abuse different logics to pass payloads: HPA leverages the object sharing to pass malicious objects into the victim programs, while Ruby mass assignment abuses a framework-specific assignment feature to modify certain existing properties on the left side of an assignment. Second, HPA can introduce hidden properties with either literal value or nested objects while mass assignment payload is merely literal value. Third, since Ruby is a strong-typed language, mass assignment vulnerability cannot create new properties to the victim object. However, JavaScript is more flexible and thus HPA can inject arbitrary properties to the victim object and even allows hidden properties to propagate over several variables before they reach the target object. Our running example is such a case: the hidden property `constructor` propagates from the input object to the internal schema object to attack the input validation logic.

It is worth noting that vulnerabilities of CWE-915 are not deserialization bugs (CWE-502 [5]). Specifically, CWE-915 is more narrowly scoped to object modification and does not necessarily exploit the deserialization procedure. For instance, HPA does not attack the logics of object deserialization. Instead, it aims at modifying the properties of internal objects.

4 LYNX Design and Implementation

4.1 Definitions

In this section, we first define several important terms used in the paper and then describe the problem we aim to address.

Hidden Property: Given a module, it contains an input object O_{input} and an internal object $O_{internal}$. A hidden property P_{hidden} exists in O_{input} only if all of the following three requirements are satisfied:

- P_{hidden} belongs to $O_{internal}$ and it is referenced in the module.
- P_{hidden} of $O_{internal}$ can be modified if a conflicting property with the same name (i.e., P_{hidden}) is added into O_{input}.
- P_{hidden} is not a default parameter of O_{input}. This means P_{hidden} of O_{input} is not initialized when the module is invoked with default parameters[2].

To help describe the problem, we use **"property carrier"** to denote all the variables that carry hidden properties (including $O_{internal}$ and O_{input}).

[2]Here "default parameters" means documented usage of the module

Harmful hidden property: A hidden property is considered harmful if an attacker can abuse this property to introduce unexpected behaviors to the module. In this paper, we consider the potential attack effects from the following three aspects:

- **Confidentiality**: The hidden property might lead to sensitive information leakage while being abused.
- **Integrity**: The attacker could violate the consistency or trustworthiness of a critical property in the module.
- **Availability**: The attacker could violate the application's expectations for the property, leading to a denial-of-service attack due to an unexpected error condition.

4.2 Challenges and Solutions

We aim to design and develop an end-to-end system that can automatically and effectively detect the HPA security issues on the target Node.js programs. However, this is not a trivial task due to the following two challenges.

C1. How to discover hidden properties for Node.js programs?

Existing techniques cannot perfectly solve this problem. In particular, static analysis can easily get the whole picture of the target program, but usually introduces high false positives, especially when dealing with points-to and callback issues. We find such cases are very commonly faced in Node.js programs. Dynamic analysis, like data flow tracking, is suitable for 1) tracking input objects and their all propagation, and further 2) discovering and flagging related property carriers, and treating their corresponding properties as potential hidden properties. However, in practice, we find the dynamic tracking often misses many critical execution paths and hidden properties, and thus causes false negatives.

Our Solution. We design a hybrid approach that leverages the advantages of both of dynamic and static analysis to discover hidden properties. First, we utilize a lightweight label system to dynamically track input objects and related properties carriers, and dump all properties of properties carriers as a part of hidden property candidates. To discover as many execution paths as possible, especially critical paths, we recursively and extensively label input objects and test the target program. Second, the above dynamic test inevitably causes false negatives. We find in many cases, critical hidden properties are still ignored even when the corresponding property carriers have been successfully flagged (see more detail in §4.4). To mitigate the problem, we introduce static analysis by greedily searching potentially ignored properties. Finally, we collect results and obtain a list of hidden property candidates.

C2. Among a large number of hidden properties, how to determine which one is valuable and exploitable for attackers?

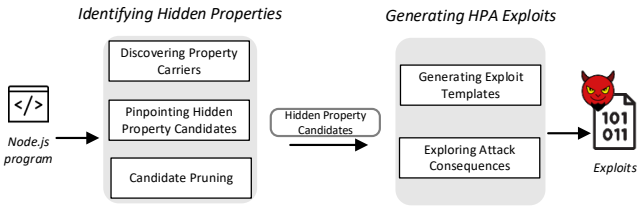

Figure 3: LYNX Overview.

We find among the collected hidden property candidates, not all of them are valuable and exploitable for attackers. Many of them do not even cause any attack consequence, and thus should be filtered out. Furthermore, the corresponding value of an identified hidden property often has specific requirements and constraints. Therefore, given a hidden property candidate, attackers need to determine its harmfulness and compute its corresponding value.

Our Solution. We leverage symbolic execution to explore all related paths, collect path constraints, detect sensitive behaviors, and finally generate exploits.

4.3 Design Overview

The overview of LYNX architecture is shown in Figure 3. As discussed in §4.2, our approach is two-fold. In the first phase, LYNX first dynamically runs a label system for recursively tracking input objects, and identifying as many property carriers as possible. We implement the dynamic label system by instrumenting the target Node.js code, and then executing the instrumented code by triggering its APIs with regular input data (e.g., test cases). Then, LYNX obtains hidden property candidates by collecting the above dynamic analysis results and applying static analysis to search ignored hidden properties. In particular, LYNX unitizes the necessary information recorded in the previous dynamic analysis step, analyzes AST (abstract syntax tree) of the target Node.js program, and detects the operations related to property access. Lastly, we prune the results based on our observations.

In the second phase, LYNX first generates exploit templates with detected hidden property candidates. Then, LYNX runs symbolic execution to reason the values of hidden properties and verify the corresponding harmfulness and attack consequences.

4.4 Identifying Hidden Properties

4.4.1 Discovering Property Carriers

We implement our dynamic analysis by instrumenting the target Node.js program. In this section, we first present the instrumentation details of labelling and tracking input, and detecting property carriers. Then, we discuss how to drive and execute the instrumented code.

Labelling and Tracking Input. We add labels to all input objects for tracking them. The newly added label is a new property, which has a unique key-value pair. For example, assuming the input object O_{input} = `{"email":"a@gmail.com"}`, LYNX instruments O_{input} with a new property. Hence, the new input object O'_{input} is `{"email":"a@gmail.com", unique_key: unique_value}`.

This above simple label-adding process works when O_{input} has a simple data structure. However, this method is not enough when O_{input} is complex. For example, when O_{input} has multiple properties such as $O_{input}.a$ and $O_{input}.b$, these child properties may propagate differently with distinct program states. If we only add one label for O_{input}, we will lose track of all these child properties. Hence, LYNX traverses O_{input} and recursively injects labels into different child properties. For instance, consider the above O_{input} with two properties, LYNX injects three different labels into the base of O_{input}, $O_{input}.a$, and $O_{input}.b$ respectively.

The labeling method outperforms classic data flow tracking (i.e., transparent tracking without changing input) in detecting property carriers since it better emulates the attack process of HPA. For example, there are cases that the tested program contains a dispatcher which distributes the input by its type. When analyzing such cases, LYNX will modifies the input in the same way as the real attack process. If the modification changes the input type, the input may trigger another path. However, the classic method may still track the path for vanilla input. Hence, our method can more accurately pinpoint the real execution paths that a real HPA payload may trigger.

However, changing the original input may also bring negative effects. For instance, assume there is a checking function that sanitizes a certain property of the input, if LYNX adds a label to the property, the program may raise an error and exit. To mitigate this problem, LYNX applies a one-label-at-one-time strategy. In each round of analysis, LYNX only adds one label to one of the properties, and then, repeats this step multiple times for testing all properties and their child properties.

Identifying Property Carriers. After adding labels to the input, LYNX executes the program with the new input and observes how the label property propagates. If LYNX finds the label propagates to an internal object, it will mark the hosting object as a property carrier. For this purpose, we instrument the target Node.js program by intercepting all variable read/write operations. When such an operation occurs on an internal object, LYNX recursively examines all properties and child properties of this object. If a label is detected, this object will be marked as a property carrier in the following form: $\langle O,L,S \rangle$, where O records the object name of property carrier, L points to the JavaScript file that contains the detected object, and S records the visibility scope of the carrier. In LYNX, "." is used to represent the scope by concatenating different function names. To differentiate function objects from variable objects, we add special suffixes `_fun` to function-type scopes. More details about the scope representation can be found in §A.2,

Driving Dynamic Analysis. LYNX runs the instrumented target Node.js program based on their types. More specifically, if the application is a web-based program (e.g., web apps), LYNX directly runs it. If the target Node.js code is in a Node.js module, LYNX needs to embed it in a simple Node.js test application. Then, LYNX calls the exposed APIs of the target Node.js module. However, in this case, LYNX needs to feed the APIs with some proper input, which is often hard to generate automatically. We mitigate this problem based on the following observation: we find most of Node.js modules are released with use cases (45 out of 50 most depended-upon packages on npm [11] have directly usable test cases). Hence, LYNX can directly use them to drive the analysis.

For triggering APIs, LYNX currently supports two types of object sharing schemes. The first is JSON serialization, which is also the most commonly used method. The second method is query-string serialization. In the Node.js ecosystem, many request parsing modules also support transferring the URL query string to objects. For example, a request parsing module called qs (100M monthly downloads on npm) converts the query string into a single object (e.g., from `?a=1&b=2` to `{a:1,b:2}`). LYNX detects hidden properties in the query string by recording and replaying web requests.

Running Example. To illustrate how LYNX identifies property carriers, we revisit our running example. As indicated in Figure 4, the injected label property propagates in a path follows the black dotted line. By tracking this flow, LYNX identifies three property carriers (`value`, `param`, and `object`) and records carrier entities for each of them. To give an example of the entity, we show how the entity of `object` is synthesized: First, to get O, LYNX checks where the label property is identified. In this case, the label property is identified from the base of `object`. As a result, LYNX directly sets O to "object". Second, to get L, LYNX obtains the file path of the current script. Third, to get S, LYNX extracts the visibility scope of the carrier. In this case, the carrier is found from an anonymous function locating from line 10 to line 22. Hence, LYNX encodes the visibility as *anon.10_1.26_1_fun*. Overall, the recorded entity will be $\langle object, script\_path, anon.10\_1.22\_1.\_fun \rangle$.

4.4.2 Pinpointing Hidden Property Candidates

Our dynamic analysis can effectively detect property carriers. However, it inevitably has false negatives on detecting hidden properties. We find in some cases important hidden properties are ignored even though the hidden property carriers have been uncovered. We mitigate the problem by applying static analysis as a complement. In this section, we first discuss the reason why dynamic analysis has false negatives. Then, we present the design details of our static analysis. Last, we discuss how to prune the analysis results.

Necessity of Static Analysis. To explain the weakness of dynamic analysis, we use a dummy vulnerable code example Listing 1 (abstracted from real code). In this example, the function `foo()` builds an internal variable `conf` based on a user-controlled variable `input` (line 2), which makes `conf` become a property carrier. The dynamic approach can capture `propertyA`, but it will miss `propertyB` if `condition` is not met. To address the issue, LYNX implements an intraprocedural static syntactic analysis that recognizes the indexing syntax, no matter if the actual code is executed or not.

Listing 1 A example code vulnerable to HPA.

```
1 function foo (input){
2   var conf = new Config(input);
3   setA(conf.propertyA);
4   // other code
5   if (condition){
6     conf.propertyB = getB();
7   }
8   return conf;
9 }
```

Extracting Hidden Property Candidates. Given a hidden property carrier "$< O, L, S >$", LYNX first identifies it in the corresponding AST (pointed by L). LYNX searches all the object references within the visibility scope recorded in S. Finally, LYNX pinpoints all the references that are child properties of O and marks them as hidden property candidates. Child properties are potential hidden properties due to the following reason: A property carrier $\langle O, L, S \rangle$ is reported because the label property can propagate to variable O. As a result, it is possible that other properties under O can also be forged/overwritten from the input. Note that not all the candidates found here can always be manipulated using inputs due to the greedy strategy. Hence, LYNX will use the next component to verify each candidate to ensure accuracy.

Due to the dynamic feature of JavaScript, child properties may be indexed in different ways. To improve the detection coverage of this module,LYNX concludes and recognizes the following three indexing methods: (1) Static indexing: properties indexed with a literal-type key (e.g., `obj.k` or `obj['k']`); (2) Function indexing: properties indexed with a built-in function (e.g., `obj.hasOwnProperty('k')`). (3) Dynamic indexing: properties indexed with a variable (e.g., `obj[kvar]`). LYNX recognizes the first two methods statically: it traverses the AST to recover the indexing semantics. To recognize properties in the third method, LYNX extracts the actual value of the `kvar` from previous execution traces. It is worth noting that, since LYNX relies on previous dynamic execution traces to support dynamic indexing, it cannot guarantee 100% coverage. That is to say, LYNX only recognizes dynamic indexing properties that are concretely indexed in the last step.

Running Example. Here we still use the example in Figure 4 to illustrate how it works. Taking the carrier `object` at line 11 as an example, LYNX first searches all its child property references within its visibility scope (the anonymous function

from line 10 to line 22) and it detects that there exists a property reference (`constructor`) exactly at where the carrier is identified. After finding this property, LYNX needs to further check whether the input object can overwrite this property or not. To this end, LYNX checks if `constructor` is a child property of O or not. After this check is passed, LYNX identifies `constructor` as a hidden property candidate.

4.4.3 Pruning the Results

As described above, hidden property candidates are discovered. However, we find some of them are known parameters rather than unknown hidden properties. This is because some Node.js modules implement optional parameters as properties of input objects. These documented properties may also be extracted in the previous step. For example, an email module by default accepts input object like `{"from": .., "to": ..}` but also accepts more options such as `{"from": .., "to": .., "cc": ..}`. It is apparent that these documented parameters are not the hidden properties.

To correct the result, we introduce a context-based analyzer to automatically "infer" whether the identified property candidate is a documented parameter or not. Our analysis is done based on the following observation: documented parameters are usually processed together by a dispatcher (e.g., a series of if-else statements).

Based on this observation, we divided the argument processing procedure into two classes: (1) The unused parameters and the used parameters (i.e., properties in original input) are processed by the same dispatcher. To deal with this case, the analyzer records the used properties from arguments of the exposed API. Then, it pinpoints hidden property candidates that reside in the same dispatcher as used parameters. (2) The unused parameters and the used parameters are processed by different dispatchers. To detect such parameters, the analyzer examines all the candidates to see if there are several candidates found from the same dispatcher. If LYNX detects that certain candidates match any of the situations, it will remove them from the result.

4.5 Generating HPA Exploits

In the previous component, LYNX discovers the key name of a hidden property. By injecting a property with such a key, the attacker may have changes overwriting/forging certain internal objects. In this section, we leverage symbolic execution to reason if the discovered properties are exploitable or not. Given a hidden property candidate, we first inject it into the input to construct the test payload. Because its corresponding value is undetermined yet, we leave the value be symbolized. Then, to decide whether a hidden property is harmful or not, we explore as many paths as possible and pinpoint sensitive sinks along the uncovered paths.

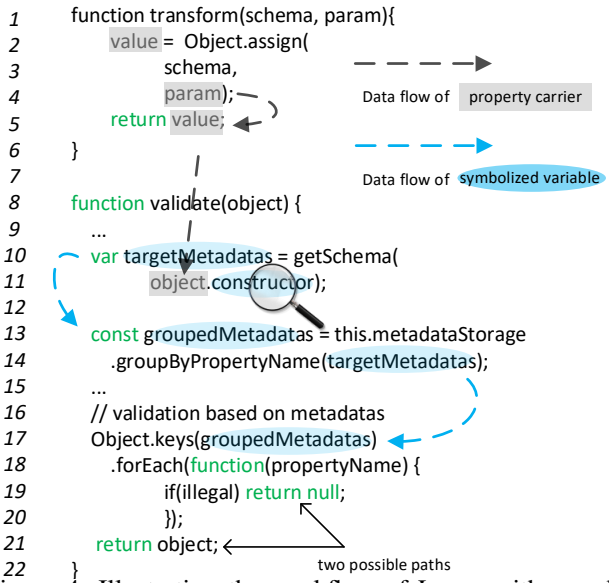

Figure 4: Illustrating the workflow of LYNX with a code snippet from our running example in §3.2 (Code is simplified for demonstration purpose).

4.5.1 Generating Exploit Templates

In this step, LYNX aims at generating the input data structure that can reach the potentially vulnerable property. We denote such structures as exploit templates since LYNX will specify a symbolic value rather than a concrete value for the value field of each hidden property. To generate the template, LYNX needs to insert a property (with the discovered key name) at the right position of the input. To figure out the insertion position (what field of the input should be modified), LYNX maintains a map between the insertion location of the label and the property carrier O.

To illustrate, we reuse the example discussed in §3.2: The original input is `{"email":"aa@gmail.com", "passwd":"11"}`. As discussed, LYNX needs to figure out the insertion position: according to the mapping, any content added to the base of the input will appear at the base of `object` at line 11 in Figure 4. Then, LYNX inserts a property named `constructor` according to the detected key name. Finally, the generated template is `{"email":"aa@gmail.com", "passwd":"11", "constructor": SYMBOL}`.

4.5.2 Exploring Attack Consequences

After generating the exploit template for each hidden property candidate, LYNX starts to analyze its potential security consequences. To this end, LYNX first symbolically executes the hidden properties to explore all possible paths. Then, LYNX pinpoints sensitive sinks along the discovered paths to decide whether a hidden property is harmful or not.

According to the definition of harmful hidden property in

Table 2: Sensitive sinks monitored by LYNX.

Category	ID	Sink	Example
Confidentiality	C_1	sensitive database query methods	The attacker leaks sensitive data from database by manipulating the SQL.
	C_2	sensitive file system operation methods	The attacker accesses confidential files by abusing the filesystem APIs.
Integrity	I_1	Critical built-in properties and code execution APIs	The attacker modifies the built-in property `constructor` to abuse property-based type checks.
	I_2	Final results of the module invocation	The attacker manipulate sanitization results to bypass security checks.
Availability	A_1	Global methods/variables	The attacker overwrites login function to crash the authentication service.
	A_2	Looping conditions	The attacker introduce an infinite loop to block the Node.js event loop [29].

§4.1, we conclude six sensitive sinks from three perspectives: confidentiality, integrity, and availability. As shown in Table 2, different sinks are used for detecting different kinds of attack consequences. In summary, sinks are implemented in two ways. The first type is keyword-based sink. Based on our observations, certain parameters of sensitive APIs can be a common sink for hidden properties. Hence, we collected a list of keywords by analyzing existing vulnerabilities reported on known vulnerability database such as snyk vulnerability DB and npmjs security advisories. We made our best effort to collect as many sensitive APIs as possible. Currently, the list contains 24 sinks: 11 filesystem operation APIs, 9 database query methods and 4 code execution methods (The API list will be released along with the source code of LYNX). While the list may be not complete, it can be easily expanded over time. Another type of sink is behavior-based sink. Many vulnerabilities are highly dependent on the code context. To identify such vulnerabilities, we focus on the behaviors that may abuse the application logic. Currently, LYNX has covered the following three malicious behaviors. (1) Return value manipulation. For vulnerabilities aiming at manipulating critical states, LYNX checks return values of the tested modules. If its return value is controllable to attackers, LYNX flags it as vulnerable. (2) Global variable tampering. If LYNX detects that a hidden property can tamper certain global variable, it will report it as a potential vulnerability. (3) Loop variable manipulation. For vulnerabilities aiming at corrupting the service by causing an infinite loop, LYNX checks looping conditions to pinpoint whether they can be manipulated through hidden properties.

After a sensitive sink is identified, LYNX prepares proof-of-concept exploits which aim at verifying whether a sink is reachable for attack-controlled value. To collect exploit, we use the input generated in the last step to re-executed the program. If the sink can be reached, the input is reported along with an attack indicator. The attack indicator is designed for helping security analysts understand how the exploit affects the sink. For different sinks, LYNX employs different rules to generate indicators. For keyword-based sinks, LYNX records what type of contents that can reach the sensitive functions/properties. For behavior-based sinks, LYNX compares execution traces of attack input and benign input to pinpoint the exploitation impact. For example, LYNX monitors the change of global objects to observe the exploitability of A_1.

Algorithm 1 Attack Exploration Algorithm

Require:
T = a set of exploit templates for the vulnerable module
m = the vulnerable module
Ensure:
$PoC = (exp, ind)$ where exp_i is the exploit and ind_i is the corresponding attack indicator.

```
1: U ← {}
2: for all t_i ∈ T do
3:     paths ← explore(m, t_i)
4:     P ← P ∪ {paths}
5: end for
6: for all p_i ∈ P do
7:     if has_sink(p_i) then
8:         exp = get_input(p_i)
9:         ind = execute(m, exp)
10:        if reach_sink(ind) then
11:            PoC ← PoC ∪ {(exp, ind)}
12:        end if
13:    end if
14: end for
```

The whole attack exploration method is summarized in Algorithm 1. The input to the search method is the tested program m and the set of exploit templates T generated in the previous step. The output of the method is the attack proof of concept denoted by (E,I) where E is the sets of the final exploits and I is the corresponding attack effect indicators. In the first phase of the algorithm, it collects the new paths discovered during symbolic execution and extracts the concrete input and the path into U. In the second phase, the algorithm examines each path P_i. After a sensitive sink is detected, it will generate the corresponding exploit to reach the sink. If LYNX detects that the sink is reachable, LYNX will report both the exploit *exp* and the attack consequence indicator *ind*.

To demonstrate the entire process, we apply the algorithm to our running example. As shown Figure 4, LYNX symbolizes the hidden property `constructor` in line 14. During the execution, two other variables are also symbolized due to the symbolic value propagation indicated by the blue dotted line. By resolving the constraints for the three symbolic values, LYNX finds two possible paths

(i.e., line 19 and line 21). Since the new path leads to the change of final module return (i.e., object or null), the exploitation hits (I_2). As a result, LYNX constructs an exploit `{"email":SQLI, "passwd":"11", "constructor":false}` (`SQLI` stands for a SQL Injection payload). After inputting the exploit to the program, LYNX collects the corresponding indicator: It detects that the return value can be changed by setting the `constructor` to false.

4.6 Implementation

We build LYNX as a Node.js application, and implement it by employing several existing tools. In the first analysis phase of LYNX (i.e., identifying hidden properties §4.4), we employ Jalangi [42] to instrument target Node.js code for implementing our label system. The instrumented Node.js code with labels is dynamically executed to discover hidden property carriers (§4.4.1). We apply Esprima [6] to generate AST (Abstract Syntax Tree) for doing static analysis on identified property carriers and extracting hidden properties (§4.4.2). In the second analysis phase of LYNX (§4.5), we use ExpoSE [36] to perform symbolic execution for determining the harmfulness of discovered hidden properties and generating exploits.

To analyze web-based applications, we implement a profiling-based pipeline that captures HTTP requests and generates corresponding test cases.

5 Evaluation

To assess the security impacts of HPA, we apply LYNX on a set of real Node.js applications and modules widely used in practice. In the following sections, we discuss our evaluation results with three research questions:

- **RQ1**: Are the hidden properties prevalent in widely-used Node.js programs? (§5.2.2)
- **RQ2**: Can LYNX effectively detect harmful hidden properties and generate corresponding exploits? (§5.2.3)
- **RQ3**: How do the discovered vulnerabilities and exploits enlarge the attack surface of the Node.js ecosystem? (§5.3, §5.5)

5.1 Data Set

Node.js has made great progress and there are already many Node.js programs available. However, we find a large number of them are rarely used or do not match our threat model. Therefore, to reduce the workload of our analysis, we restrict our data set collection process. In particular, we collect Node.js programs based on the following two criteria: (1) The tested programs should be used to interacting with external input, and their APIs should accept objects (via either JSON or query-string serialization). (2) The tested programs should be widely-used or continuously maintained.

Table 3: Overall detection results. The numbers within the parentheses indicate the number of programs that contain hidden properties. #PC, #HP, and #DA respectively denote the number of property carriers, hidden property candidates, and detected documented arguments.

Category	Tested Programs	Detection Results		
		#PC	#HP	#DA
Database	9 (8)	323	78	0
Input Validation	48 (30)	999	122	0
User Functionalities	34 (26)	584	156	24
Web	11 (7)	1269	95	0

To satisfy the first criteria, we collect programs from categories that are most likely to be exposed to input. These categories include database, input validation, user functionalities, and web-based application/middileware. To satisfy the second criteria, we collect programs from known vendors (e.g., MongoDB), and projects that have at least 1000+ star on Github or 500 monthly downloads on npm (To guarantee the volume of our samples, we might slightly lower this criteria when all the popular programs have been selected).

In total, we collected 102 Node.js programs as our analysis dataset. There are 91 Node.js modules and 11 web-based programs. Among the 11 web-based programs, 4 are minimal web frameworks/middlewares and 7 are complete web applications.

5.2 Analysis Results

5.2.1 Overview

We run LYNX on a Ubuntu 18.04 machine equipped with Intel Core i5-9600K (3.70GHz) and 32 GB memory. In total, we detected 451 hidden property candidates and confirmed 15 previously unknown HPA vulnerabilities. By the timing of writing, 10 CVEs have been assigned for our findings. More than half of them are rated as "Critical" and "High" severity[3] by NVD (national vulnerability database).

Among these vulnerabilities, two of them are identified from complete web applications. The other 13 vulnerabilities are identified from modules, which in total impact 20,402 dependent applications/modules. The Node.js community pays great attention to our findings. An authoritative public vulnerability database creates a new notion to track related vulnerabilities.

5.2.2 Phase#1: Identifying Hidden Properties

To answer RQ1 (Are hidden properties prevalent in popular Node.js programs?), we analyze how many (and what kind of) hidden properties are detected from widely-used Node.js programs.

Table 3 summarizes our detection results (Table 7 lists the complete detection results). In Table 3, from the second col-

[3]The well-known heartbleed vulnerability was also rated as "High" severity.

Table 4: Exploit results of LYNX.

Category	Reported	Exploitable	Missed
Database	2	2	1
Input Validation	7	4	2
User Functionalities	5	4	0
Web	1	1	1

umn "Tested Programs", we can observe that hidden properties widely exist in all categories that are likely to be exposed to external input. Overall, 69% (70/102) tested programs are found to contain hidden properties.

The first two columns under "Detection Results" indicate the number of property carriers hidden property candidates. In total, LYNX identifies 451 hidden property candidates by analyzing 3175 property carriers. We can observe that hidden property candidates widely exist in all categories of our dataset. The last column under "Detection Results" shows how many candidates are identified as documented arguments by LYNX. To figure out the correctness of our documented argument inferring rules, we compare the documented arguments from their official documentations with our results. we found our context-based rules correctly recognize all documented arguments from identified hidden properties.

Note that we drive our analysis based on the types of Node.js programs being tested. For the 91 npm modules, we directly reuse the use cases provided on their npm homepages as the test input. For the remaining 11 web-based programs, we manually interact with applications and generate test cases with our profiling-based pipeline. LYNX analyzes both JSON and query-string serialization channels for web-base programs. 7 out of these 11 web-based programs support both query-string and JSON serializations (in different APIs).

5.2.3 Phase#2: Exploring Attack Consequences

We assess the effectiveness (RQ2) of LYNX from the following two aspects: (1) Does LYNX effectively pinpoint potential vulnerabilities from programs of different categories? (2) Does LYNX successfully generate exploits that can directly or be easily ported to introduce real-world attack effects?

Table 4 shows the summarized exploit result during the second phase. In this table, the columns "Reported" record how many sensitive sinks are reported to be vulnerable by LYNX. The column "Exploitable" indicates how many of reported sinks that LYNX automatically exploit and are manually confirmed to be real vulnerabilities. From the two columns, we can observe that LYNX is capable of pinpointing potentially vulnerable sinks from different types of programs. Moreover, the "quality" of reported issues are good. Overall, we found 11 out of 15 reported vulnerabilities are confirmed to be vulnerable, and the other 4 cases are considered to be harmless. Among the 4 cases, although some hidden properties do lead to certain sensitive sinks, they are still constrained by the program semantics and thus no significant attack effects can be introduced. For instance, when LYNX exploiting a hidden property from a validation library, it causes an execution exception and thus triggers sink ⑫ (final result manipulation). However, since the exception is later handled by the program, it does not enable any attack effects such as validation bypass.

The last column ("Missed") of Table 4 records the hidden properties that LYNX successfully detects (phase#1) but fails to generate usable exploits (phase#2). To find out such hidden properties, we manually examine all hidden property candidates reported by LYNX. There are three types of failures. First, some hidden properties have a particular constraint that is not presented in the code semantics. For example, taffyDB (a popular JavaScript database) has a hidden property that can leak arbitrary data by forging as the internal index. However, the constraint associated with the index is in the memory rather than in the code. Thus, LYNX cannot construct a valid index even though the index is in an easily-guessable format (e.g., T000002R000001). This kind of failure results from the limitation of symbolic execution. To cover such failures, fuzzing techniques may be a good complement to cover the part that symbolic execution fails to analyze. We leave improving our symbolic execution as our future work.

Another type of failures result from multi-constraint issues: To exploit some hidden properties, some parameters of the input must be set to certain values. Such failures can be addressed by extending LYNX to explore multiple variables (not only hidden properties but also documented parameters) simultaneously. The last type of failure comes from the syntax incompatibility problem. The incompatibility results from the fact that our underlying instrumentation framework (Jalangi) is not compatible with certain grammars after ECMAScript 6. We mitigated this problem by down-compiling incompatible programs with Babel [3] or avoiding instrumenting incompatible code. To ease the process of addressing the incompatibility, we built an automatic down-compiling tool, which will be released together with LYNX.

5.3 Impact Analysis of Identified HPA Vulnerabilities

In this section, we seek to answer RQ3 by understanding how HPA vulnerabilities introduce serious attack effects into the Node.js ecosystem. As shown in Table 5, we detected 15 HPA vulnerabilities. To fix these vulnerabilities, we have made responsible disclosure and notified the vendors. They reacted immediately. So far 10 vendors have confirmed the vulnerabilities, and 7 of them have released corresponding patches. Next, we will explain the security impacts of HPA from the following three perspectives.

Confidentiality. We found that 4 of the identified vulnerabilities (i.e., HP-1, HP-2, HP-3, and HP-14) impact confidentiality of the program (e.g., leaking sensitive information from the database). The vulnerabilities HP-1 and HP-2 are found from two widely-used mongoDB drivers. By exploiting HP-1 and HP-2, the attacker can force database to always return

Table 5: Vulnerabilities detected by LYNX (C: Confidentiality; I: Integrity; A: Availability).

#ID	Product Name	Affected API	Description	Impact		Attack Effects			Disclosure	
				Downloads	Dependents	C	I	A	status	severity
1	mongoose	findOne()	SQL Injection	2,740,341	9,211	✔			Fixed (CVE1)	Critical
2	mongoDB driver	find()	SQL Injection	6,165,075	8,435	✔			Fixed (CvE2)	-
3	taffyDB	query APIs	SQL Injection	1,628,860	108	✔			Confirmed (CVE3)	High
4	class-validator	validate()	Bypass input validation	1,077,954	1,639		✔		Confirmed (CVE4)	Critical
5	jpv	validate()	Bypass input validation	481	1		✔		Fixed (CVE5)	Medium
6	jpv	validate()	Bypass input validation	481	1		✔		Reported	Medium
7	valib	hasValue()	Bypass input validation	479	8		✔		Reported	-
8	schema-inspector	validate()	Bypass input validation	35,783	104		✔		Fixed (CVE6)	High
9	schema-inspector	sanitize()	Bypass input validation	35,783	104		✔		Fixed(CVE6)	High
10	bson-objectid	ObjectID()	ID forging	142,562	298		✔		Fixed (CVE7)	High
11	component-type	type()	Type manipulation	943,555	140		✔		Reported	-
12	component-type	type()	Type manipulation	943,555	140		✔		Reported	-
13	kind-of	kindOf()	Type manipulation	196,448,574	458		✔		Fixed (CVE8)	High
14	cezerin	getValidDocumentForUpdate()	Order state manipulation	1871	–	✔			Confirmed (CVE9)	High
15	mongo-express	addDocument()	Denial of service	6,965	–			✔	Fixed(CVE10)	Medium

data/true regardless of the correctness of query condition. This can be abused to leak sensitive information or bypass access control. For example, an attacker might log into other user's accounts by forcing the authentication result to be true (we will demonstrate a real-world case of this vulnerability in §5.5). The vulnerability HP-3 is found from taffyDB. This is a serious universal SQL Injection that can be abused to access arbitrary data items in the database: It is found that a hidden property can forge as taffyDB's internal index ID. If an index ID is found in the query, taffyDB will ignore other query conditions and directly return the indexed data item. Moreover, the index ID is in an easily-guessable format (e.g., T000002R000001), so that attackers can use this vulnerability to access any data items in the DB. Vulnerability HP-12 is found from cezerin, an eCommerce web application. It is found that a hidden property can modify the critical data stored in database (i.e., payment status `ispaid`).

Integrity. We found that 10 of the identified vulnerabilities (i.e., HP-4, HP-5, HP-6, HP-7, HP-8, HP-9, HP-10, HP-11, HP-12, and HP-13) compromise the integrity of Node.js applications. 4 widely-used input validation modules are impacted by HPA. Our running example, class-validator (HP-4), allows attackers to overwrite the format schema object, which leads to the arbitrary input validation bypass. Jpv (HP-5 and HP-6) checks the type of unsafe objects on the their prototype. However, since HPA can modify properties in the prototype, the validation result of jpv can be manipulated. The other three validation bypass vulnerabilities are found from one API (HP-6) from valib and two APIs (HP-7 and HP-8) from schema-inspector: By modifying `hasOwnProperty` function under the unsafe object's prototype, security checks can be skipped. Note that these three cases have limited exploit scenario: Attackers needs to pass valid function definitions, which is not a widely supported feature [8].

The other 4 vulnerabilities (HP-10, HP-11, HP-12, and HP-13) that impact program integrity are from user functionalities modules. These 4 vulnerabilities are exploited in a similar way: By manipulating some critical properties under the input object, attackers can manipulate the final result of the module invocation. Such manipulation might introduce serious risk to the application. For example, clone-deep, an object cloning module used in 1,822,028 projects according to Github, uses vulnerable kind-of (HP-13) to perform type checking before cloning. If the variable `var` to be cloned is detected as `array`, clone-deep recursively calls itself `var.length` times to clone all elements under `var`. With HP-13, a malicious object can forge as an array with a very large `length`. When cloning such an object, clone-deep will go into a super big loop, and thus freeze the whole application (Time-consuming tasks can block Node.js applications due to its single-thread model).

Availability. We found that the availability of 1 web framework (i.e., HP-15) can be affected by HPA. This vulnerability is detected from mongo-express, a web-based application. It is found that a hidden property can introduce an infinite loop to the application, which blocks the whole application. We will include more details of the case in §5.5.

Community Impact. Our findings have been corroborated by the Node.js community. To help developers be aware of this new risk, we proposed a new notion should be used to describe and track related issues. An authoritative public vulnerability database maintained by snyk has accepted the proposal and starts using the notion in related security issues [10].

Remark. Based on the impact analysis, we posit that the HPA attack indeed enlarges the attack surface of the Node.js

ecosystem. The claim is supported by the following two insights. (1) By establishing unexpected data dependencies to internal objects in the application, the HPA attack effectively compromises previously unreachable program states and introduces different kinds of attack effects. (2) Classic defense techniques (e.g., input validation) can not mitigate the HPA. As shown in Table 5, some widely-used validation modules are vulnerable to the HPA attack.

Listing 2 The online game is vulnerable to HPA because it calls vulnerable mongoDB APIs to handle input.

```
1 GameServer.loadPlayer = function(socket,id){
2   GameServer.server.db.collection('players').findOne({
3     _id: new ObjectId(id)},
4     function(err,doc){...}
5   });
6 };
```

5.4 Analysis Coverage and Performance

We measure the code coverage of LYNX for each Node.js program based on ExpoSE [36]'s coverage monitoring, which computes 'LoC being executed' / 'total LoC in executed files' (dependencies not counted). We discuss our coverage measurement results below, based on the different types of tested Node.js programs: modules and web-based programs.

For Node.js modules, the code coverage varies (i.e., 10% - 80%). While a large portion of modules achieve decent coverage (more than 40%), we argue the code coverage does not necessarily indicate the effectiveness of LYNX: To find practical vulnerabilities, we selectively test APIs that match our threat model (likely to be exposed to external user and accepting objects). As a result, even though test cases are available for most APIs, we are not blindly testing all of them. For instance, if an API does not accept parameters at all, we will not include it into our test, and the code coverage contribute by such API testing does not help us vetting HPA from tested programs.

For web-based programs, LYNX achieves 21% code coverage on average. We find this is because web applications usually have a large number of functionalities/APIs, and our profiling-based testing may not cover all of them. To help LYNX discover more web APIs, incorporating active web scanners [2] could be a promising future work.

Besides code coverage, we also measure the running time of each phase. As an offline tool, LYNX achieves reasonable analysis speed: For detecting hidden properties, it typically takes no more than 10 seconds to analyzing one API (90% cases). For very large programs such as web applications, the analysis may take more than 200 seconds per API (no more than 10 cases). For exploiting hidden properties, it takes longer time because LYNX needs to explore multiple paths for each candidate. Typically, it takes around 50 seconds per hidden property. Detailed results can be found at §A.3.

5.5 Case Studies

Accessing Confidential User Data. LYNX reports a harmful hidden property (`_bsontype`) from mongoDB Node.JS driver. This property is used to decide the query type and should not be provided by input. However, it is found that mongoDB allows input to modify this property via HPA. Since mongoDB handles query objects according to pre-defined types. The attacker can specify an unknown `_bsontype` (e.g., aaa) to force mongoDB not serializing certain objects. For example, this can be abused to force the query result to be always true (i.e., by not serializing the query filer). By exploiting this vulnerability, an attacker can launch unauthorized access to confidential data in the mongoDB.

To demonstrate one of the attack vectors, we use Phaser Quest, an online game that uses the vulnerable mongoDB driver module. As shown in Listing 2, the program loads/deletes user profile by a user-provided secret identifier (`id`). By abusing the discussed vulnerability, the attacker can force the database to return a valid user regardless of the correctness of the identifer. By doing this, the attacker can log in/delete arbitrary player's accounts.

We have made responsible disclosure to MongoDB team. They has patched the vulnerability and acknowledged us at their security advisories.

Blocking the event handler. Since Node.js is based on a single-thread model, the availability of its event handler is very critical and has been discussed a lot [29, 37, 43]. In the second case, we would like to demonstrate how HPA can attack the event handler and thus freeze the entire program.

LYNX reports a harmful hidden property (`toBSON`) from mongo-express, a web-based mongoDB admin interface. By abusing this property, an authenticated user issues a time-consuming task to block the event handler of Node.js. As shown in the upper part of Listing 3, a hidden property `toBSON` is identified in line 3. By tracking the data flow of this property, we found that it reaches a sensitive sink [15] in line 12, which is for executing code in a sandbox. Hence, the attacker can pass a time-consuming function (e.g., an infinite loop) to block the event handler.

After receiving our vulnerability report, the project team confirmed it immediately and added this issue to their security advisories. By the time of paper writing, we are working together with them on the bug fixing.

6 Discussion

Countermeasures. We conclude three major countermeasures against HPA. For example, one of them is validating input objects. Since the first step of HPA is injecting additional properties, removing unwanted (malicious) properties could be a feasible mitigation. Due to the page limit, more details

Listing 3 HPA impacts the availability of this program by attacking the unique single-thread model of Node.js.

```
// code from bson module
if (object.toBSON) {
  object = object.toBSON();
}

// code from mongodb-query-parser module
const SANDBOX = new SaferEval(FILTER_SANDBOX);
SANDBOX.runInContext(input);
```

about the three approaches are discussed in Appendix§A.1.

Limitations. First of all, LYNX needs external input (i.e., module test cases or user interactions on the web) to trigger analysis. Since APIs of different modules/applications have different context dependencies and parameter formats, it is hard to automatically infer and resolve these prerequisites. For example, during our evaluation, we found that we need to log into the tested web program to access certain APIs. To address the issue, we have implemented a pipeline that automatically replays and mutates API invocations. To test web-based programs, security analysts just need to act like normal users to perform interactions. In the future, we are considering introducing an automatic input format reasoning component to LYNX to ease the input generation process. Second, like many other dynamic analysis tools, LYNX may have false negatives. For example, it is possible that the test input we use does not explore all the branches of certain tested programs. To improve coverage, we can combine LYNX with fuzzing techniques. Third, Lynx does not cover all input channels existed in the Node.js ecosystem: In the ecosystem, different programs may use distinct methods/code implementations to share objects, so it is difficult to systematically cover all channels and it is not the focus of this paper. While we acknowledge that Lynx does not cover all input lines, it does cover the two most popular methods and can support a large number of programs. As future work, we are considering to support more input channels.

7 Related Work

7.1 Vulnerabilities of Node.js Ecosystem

Recently, researchers have discovered many security issues in the Node.js ecosystem. Existing offensive research in Node.js can be divided into two categories: attacks launching from external users and attacks launching from internal modules. In the first category, Ojamaa et al. [37] studies the security of Node.js and discussed potential risks such as command injection attack. Synode [44] further studies command injection attack and presents an automatic mitigation approach. Staicu et al. [43] show how ReDoS (regular expression denial of service) affects real Node.js websites. Davis et al. [29] identify and mitigate a new type of denial of service (DoS) attack, Event Handler Poisoning (EHP), which targets the event-driven architecture of Node.js. Arteau et al. identify prototype pollution [12] (PP), a security risk that tampers object prototypes in Node.js applications. PP and HPA differ from the following two aspects. (i) Attack behavior: PP introduces attack effects by tampering one special kind of JavaScript data type (prototype), while HPA does not modify prototype. (ii) Exploit condition: The exploitation of PP requires the attacker to explicitly assign a value to the prototype. For example, the code `obj[__proto__] = input` is vulnerable to PP while `Object.assign(obj, input)` is not. In addition, we can observe that data serialization is not necessary for PP. However, HPA does not require prototype assignment. In contrast, it passes the attack payload through data serialization. Because of these differences, the above counterexample of PP is vulnerable to HPA since `input` may carry "hidden" properties and propagates them to `obj`.

In the second category [23, 38, 49], researchers study how malicious/buggy third-party modules impact the Node.js applications. Brown et al. [23] detect and prevent binding-layer bugs in both server-side and browser-side platforms. Patra et al. [38] define and classify JavaScript module conflicts and propose ConflictJS to detect such risks. Zimmermann et al. [49] present a large-scale study on the Node.js ecosystem and identify several weak spots in the ecosystem. In contrast to these vulnerabilities, HPA does not require planting malicious code into the victim application.

7.2 Analysis of JavaScript Code

Researchers also developed tools to help detect JavaScript bugs/vulnerabilities. Many existing analysis tools [25, 31, 34, 36, 38–40, 45, 47] are based on information flow analysis. For example, Stock et al. [47] propose dynamic taint tracking to prevent DOM-based XSS. Lekies et al. [34] propose a system that leverages byte-level dynamic taint tracking to detect and validate DOM-based XSS. Typedevil [39] performs variable-level information flow analysis to report inconsistent types. Although LYNX also performs data flow analysis, it subtly differs from existing tools [39, 45] by using a new labeling and tracking method to analyzes HPA related data structures (e.g., property carriers). Arteau et al. proposes a fuzzing approach to detect prototype pollution [12], which injects a static payload into the test input and flags vulnerabilities if any prototypes are modified. However, the fuzzer cannot be used to detect HPA because (1) HPA does not necessarily need to modify the prototype so that the fuzzer will not report any vulnerabilities; (2) Hidden properties are internal states with various random name variable (e.g., `_bsontype`), so syntactic analysis is essential when we want to extract these hidden properties. However, the fuzzer does not have the capability to extract these syntax information (The fuzzer only runs with the fixed input `__proto__`).

There are also tools in other language platforms designed

to detect security issues similar to HPA. Dahse et. al [28] proposed a static object-sensitive approach to detect PHP objection injection. However, this approach cannot be used to detect HPA: (1) The analysis is designed for analyzing object-oriented code, and it relies on the object-oriented programming (OOP) semantics such as `new()` to guide its analysis. However, many of our analysis targets are not OOP; (2) The approach focuses on exploiting potentially vulnerable magic methods, while HPA does not have a corresponding sink. Cristalli et. al [26] proposed a sandbox-based approach for preventing Java deserialization vulnerabilities. The proposed approach traces benign deserialization executions and detects suspicious Java method invocation based on the previous execution traces. Since HPA exploits logic bugs rather than arbitrary command execution bugs, this approach is not suitable for mitigating HPA.

7.3 Security vulnerabilities of Browser-side JavaScript

Security researchers also discovered many vulnerabilities the browser-side scripts. One of the most important classes of browser-side vulnerabilities is Cross-site scripting (XSS) [27, 30, 33–35, 41, 46, 48]. Recently, Lekies et al. [35] systematically investigate and mitigate a class of vulnerability, Cross-Site Script Inclusion attack (XSSI). XSSI is a browser-side attack that can leak sensitive user data by including a script from an attacker-controlled domain. Fass et al. [30] propose HideNoSeek, a general camouflage attack that evades syntactic-based malware detectors. Steffens et al. [46] propose Persistent Client-Side XSS attack and investigate its severity on the Web. Schewarz et al. [41] propose two new side-channel attacks in JavaScript to automatically infer host information. In contrast to related work, we focus on vulnerabilities in the server-side Node.js programs.

8 Conclusion

In this paper, we conduct the first systematic study on the object sharing of Node.js programs and design a new attack named hidden property abusing. By exposing previously unreachable program states to adversaries, the new attack enlarges the attack surface of Node.js. The new attack surface leads to the discovery of 15 zero-day vulnerabilities, all of which can be exploited to introduce serious attack effects. To detect HPA, we build LYNX, a novel vulnerability finding and verification tool that combines static and dynamic analysis techniques to pinpoint and exploit vulnerable internal objects in Node.js programs. Using LYNX against 102 widely-used Node.js programs, we show that LYNX can effectively detect HPA vulnerabilities.

Acknowledgement

We would like to thank our paper shepherd Giancarlo Pellegrino and the anonymous reviewers, for their insightful feedback that helped shape the final version of this paper. We also thank Yuhang Wu for his contribution during the early stage of the project. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-17-1-2895, N00014-15-1-2162, N00014-18-1-2662 and N00014-20-1-2734, the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031, and the National Science Foundation (NSF) under grants 1700544, 1617985. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflects the views of ONR, DARPA, or NSF.

References

[1] *2018 Node.js User Survey Report*. https://nodejs.org/en/user-survey-report.

[2] *Acunetix: Web Application Security Scanner*. https://www.acunetix.com/.

[3] *babel: A JavaScript Compiler*. https://babeljs.io/.

[4] *cookies package on npm*. https://www.npmjs.com/package/cookies.

[5] *Deserialization of Untrusted Data*. https://cwe.mitre.org/data/definitions/502.html.

[6] *ECMAScript parsing infrastructure for multipurpose analysis*. https://esprima.org/.

[7] *Electron (software framework)*. https://en.wikipedia.org/wiki/Electron_(software_framework).

[8] *Functions in JSON*. https://teamtreehouse.com/community/functions-in-json.

[9] *Improperly Controlled Modification of Dynamically-Determined Object Attributes*. https://cwe.mitre.org/data/definitions/915.html.

[10] *Internal Property Abusing in snyk*. https://snyk.io/vuln/SNYK-JS-BSON-561052.

[11] *npm most depended upon packages*. https://www.npmjs.com/browse/depended.

[12] *Prototype pollution attacks in NodeJS applications*. https://www.youtube.com/watch?v=LUsiFV3dsK8.

[13] *routing-controllers: A Typescript Routing Controllers Framework*. https://github.com/typestack/routing-controllers.

[14] *Ruby mass assignment vulnerability on Github*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2054.

[15] *safe-eval Documentation*. https://www.npmjs.com/package/safe-eval.

[16] *Skype, Slack, other Electron-based apps can be easily backdoored*. https://arstechnica.com/information-technology/2019/08/skype-slack-other-electron-based-apps-can-be-easily-backdoored/.

[17] *StackOverflow Developer Survey*. https://insights.stackoverflow.com/survey/2019.

[18] *useragent package on npm*. https://www.npmjs.com/package/useragent.

[19] *V8 JavaScript Engine*. https://v8.dev/.

[20] *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[21] *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[22] *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[23] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and Preventing Bugs in JavaScript Bindings. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[24] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (Security)*, Baltimore, MD, August 2005.

[25] Ravi Chugh, Jeffrey A Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. *ACM Sigplan Notices*, 44(6):50–62, 2009.

[26] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. Trusted execution path for protecting java applications against deserialization of untrusted data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 445–464. Springer, 2018.

[27] Johannes Dahse and Thorsten Holz. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium (Security)* [20].

[28] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 42–53, 2014.

[29] James C Davis, Eric R Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the 27th USENIX Security Symposium (Security)* [21].

[30] Aurore Fass, Michael Backes, and Ben Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[31] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Baltimore, Maryland, July 2015.

[32] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[33] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Vancouver, British Columbia, Canada, May 2009.

[34] Sebastian Lekies, Ben Stock, and Martin Johns. 25 Million Flows Later: Large-scale Detection of DOM-based XSS. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.

[35] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The unexpected dangers of dynamic javascript. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 723–735, 2015.

[36] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 425–438. ACM, 2019.

[37] Andres Ojamaa and Karl Düüna. Assessing the Security of Node.js Platform. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 348–355. IEEE, 2012.

[38] Jibesh Patra, Pooja N Dixit, and Michael Pradel. Conflictjs: Finding and Understanding Conflicts between JavaScript Libraries. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, May – June 2018.

[39] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 314–324. IEEE Press, 2015.

[40] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2010.

[41] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)* [22].

[42] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon 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*, pages 488–498. ACM, 2013.

[43] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 361–376, 2018.

[44] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[45] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. An Empirical Study of Information Flows in Real-World JavaScript. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pages 45–59, 2019.

[46] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)* [22].

[47] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise Client-side Protection against DOM-based Cross-site Scripting. In *Proceedings of the 23rd USENIX Security Symposium (Security)* [20].

[48] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From Facepalm to Brain Bender: Exploring Client-side Cross-site Scripting. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[49] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the NPM Ecosystem. In *Proceedings of the 27th USENIX Security Symposium (Security)* [21].

A Appendix

A.1 Countermeasures

Validating Input Objects. First of all, objects generated from input should be validated. Since the first step of the HPA attack is to inject additional properties into the input data, one straightforward mitigation is to remove

Table 6: Examples of *S* and their meanings

Scope	Refers to
*	the carrier is globally visible to the whole script
login_fun	the carrier is only visible to function `login`
login_fun.is_admin_fun	the carrier is only visible to a nested function `is_admin` defined in function `login`
anon.12.1.12.5._fun	the carrier is visible to an anonymous function locating at line 12 from column 1 to column 5

unwanted (malicious) properties by performing input validation. There are two possible validation methods. The first method is using a blacklist to prevent properties that have the same name as the critical internal properties (e.g., constructor) from entering the application. The advantage of this method is that it is flexible to deploy and requires no major changes to the whole module. Several vulnerabilities we reported (e.g., CVE1 and CVE7) have been patched by this method. The disadvantage of this method is that it may be bypassed due to an incomplete blacklist. The second method is to enforce a whitelist input format check for every API, which means it only permits known properties entering into the program. The advantage is that it ensures better input validation coverage, while the disadvantage is that it is more difficult to deploy since developers have to manually declare the input schema case by case.

However, we should be aware that input validation is not the cure for HPA, because the validation module itself might also be vulnerable to HPA. As shown in Table 5, 5 HPA vulnerabilities are identified from input validation modules. Hence, we suggest that the input validation module should be carefully designed (e.g., by following the other two suggestions below).

Avoiding packing multiple variables into one argument. Second, we advocate that developers should avoid putting different variables into one object and uses it as an argument when invoking APIs. This is a very common programming style in Node.js because it complies with the classic class model in Object-oriented programming (OOP) which treats a variable as a certain instance that consists of different members. For example, we found that exposed APIs (e.g., `findOne()`) of mongoDB's driver packs all query data as a single object (i.e., `query`). However, this practice could be risky in Node.js because: (1) Unlike other OOP languages that have member access control (e.g., modifiers like `private` and `public` in C++ and Java), JavaScript enforces no property access control for its objects. Hence, arbitrary internal properties can be overwritten when a user-controlled object is copied/assigned to certain internal objects. (2) Developers adapting this style are likely to define some properties (e.g., `userRole`) within the objects to store their meta information. An attacker might forge these properties to introduce security risks. For example, mongoDB driver differentiates differentiate types of `query` according a self-defined property `_bsontype`. It turns out that this self-defined property can be forged to leak data from the database.

Isolating internal program state from input. It is important to put unsafe external objects and internal state objects into different domains so that they will not affect each other. For example, one potential solution is to label data from the external interfaces (e.g., Network APIs) and perform validation when overwriting properties in internal objects at the Node.js runtime engine level. Though this solution fundamentally mitigates HPA, it also has two disadvantages. First, it incurs overhead into the runtime engine because additional data structures need to be attached to the object implementation. Second, in some scenarios, developers do want external input to change certain properties of an internal object. Hence, developers will have to add additional code to declare a permission for such cross-domain behaviors if this feature is implemented in the engine.

A.2 Scope Representation in LYNX

Table 6 shows several examples of the scope representations in LYNX and the corresponding meanings.

A.3 Complete Result

Table 7 shows the complete detection results of the 102 tested Node.js programs.

Table 7: **Complete detection results**. Downloads with (g) are counted from github, the major release channel of these projects.

Category	Program	Version	LOC	Downloads	Coverage	Time		Detection Results	
						Detection	*Exploitation*	*#PC*	*#HPC*
Database	json-records	1.0.5	169	52	0.34	12s	37.3s	15	1
	keyv	4.0.0	93	12,781,403	0.64	2.1s	52.5s	10	3
	levelup	4.3.2	353	1,162,162	0.31	6.1s	39.2s	28	2
	LokiJS	1.5.8	6372	1,025,170	0.10	27.2s	49.4s	53	3
	Lowdb	1.0.0	486	857,106	0.60	540.7s	N/A	7	0
	mongoDB	3.3.3	22256	6,165,075	0.28	329.8s	74.2s	63	8
	mongoose	5.8.1	41750	2,941,692	0.19	359.2s	328.1s	92	41
	mongoist	2.4.0	2041	10,646	0.39	60.3s	239.7s	40	14
	Taffydb	2.7.3	1478	1,628,860	0.12	10.9s	49.6s	15	6
Input Validation	Ajv	6.10.2	10997	101,694,541	0.36	240s	N/A	6	0
	AnotherJsonSchema	3.8.2	10994	267	0.15	2.2s	N/A	18	0
	allow	2.1.0	658	132732	0.55	7.6s	17.1s	7	8
	async-validator	3.4.0	1972	2,502,423	0.29	3.5s	N/A	17	0
	async-validate	1.0.1	4349	1,731	0.41	2.6s	14.6s	38	5
	amanda	1.0.1	9281	30,392	0.22	2s	N/A	28	0
	assert-args	1.2.1	1792	146	0.35	13s	17.7s	21	2
	class-validator	0.9.1	5668	1,077,954	0.45	1409.0s	91.4s	42	8
	congruence	1.6.11	10268	146	0.14	446.5s	N/A	48	0
	Consono	1.0.6	564	1,107	0.43	8.8s	91.17s	18	5
	DataInspector	0.5.0	1349	29	0.41	33.3s	447s	11	4
	enforce	0.1.7	1546	14,047	0.29	3s	15s	14	1
	fastest-validator	1.7.0	2315	130,804	0.37	6.4s	N/A	3	0
	Forgjs	1.1.11	3562	167 (g)	0.61	16.1s	354.9s	31	4
	fieldify	1.2.2	2189	73	0.49	2.2s	41.0s	14	2
	fefe	2.0.2	729	146	0.52	1.2s	55.8s	7	1
	hannibal	0.6.2	2847	2,668	0.31	3.1s	21.8s	46	4
	have	0.4.0	579	1,591	0.55	1.2s	15.3s	3	3
	indicative	7.3.0	311	31,235	0.30	2.8s	N/A	4	0
	isMyJsonValid	2.20.0	554	6,428,255	0.34	1.5s	N/A	4	0
	is-extendable	1.0.1	8	103,501,348	0.36	1.0s	13.9s	3	1
	is2	2.0.6	1969	2,944,841	0.28	1.2s	N/A	4	0
	joi	16.1.7	7435	12,575,750	0.31	142s	N/A	16	0
	jpv	2.0.1	206	481	0.20	1.6s	55.4s	25	14
	Jsonschema	1.2.4	335	53,884,848	0.18	3.5s	57.5s	39	8
	json-gate	0.8.23	732	2,228	0.29	1.3s	28.4s	18	2
	legalize	1.3.0	2297	1,745	0.43	54.2s	55.3s	23	1
	Object-inspect	1.7.0	701	40,736,308	0.44	5.6s	104.6s	31	6
	obj-schema	1.6.2	511	207	0.24	5.6s	N/A	23	0
	OW	0.15.0	311	624,684	0.37	36.9s	43.5s	16	1
	Property-Validator	0.9.0	4130	1,242	0.35	4.5s	N/A	15	0
	schema-inspector	1.6.8	5161	35,783	0.24	51.0s	53.8s	48	8
	satpam	4.4.1	57151	4,256	0.51	47.8s	201.9s	27	1
	typeof-properties	3.1.3	1047	1,184	0.43	2.6s	N/A	20	0
	typical	6.0.1	192	2,629,970	0.13	1.2s	N/A	6	0
	treat-like	1.0.0	767	47,832	0.36	0.9s	N/A	31	0
	themis	1.1.6	5081	942	0.26	45.7s	62.7s	28	1
	validate.io-object	1.0.4	6	15,176	0.31	0.9s	N/A	6	0
	ValidatorJS	3.18.1	68823	106,038	0.19	3.9s	48.7s	33	3
	validate.js	0.13.1	933	662,549	0.19	5.2s	N/A	21	0
	validate-arguments	0.0.8	725	1,788	0.08	257.4s	319.4s	21	3
	validated	2.0.1	1561	2101	0.49	4.3s	72.4s	18	5
	valida	2.4.1	2704	731	0.42	2.2s	57.1s	16	8
	validall	3.0.17	1202	341	0.33	2.3s	50.6s	31	6
	Valib	2.0.0	327	479	0.27	2.3s	51.2s	15	1
	value-schema	3.0.0	1909525	1,900	0.46	2.1s	N/A	31	0
	Yup	0.27.0	2088	4,455,577	0.46	8.0s	24.2s	42	5
	Z-schema	4.2.2	33221	2,434,914	0.29	15.6s	38.8s	19	1
User functionalities	Avsc	5.4.16	6508	108,450	0.18	19s	N/A	9	0
	Analytics	3.4.0	185	105,510	0 *	19.7s	51.3s	20	8
	bson-objectid	1.3.0	259	142,562	0.21	1.1s	40.7s	5	4
	Cookies	0.8.0	503	2,549,728	0.46	46.7s	97.4s	6	1
	component-type	1.2.1	2893	943,555	0.55	4.3s	48.0s	8	5

* Our underlying instrumentation (Jalangi) does not detect any code execution in the module, which results in the 0 here. In fact, code in the module does execute and we even detect hidden properties.

Category	Program	Version	LOC	Downloads	Coverage	Time		Detection Results	
						Detection	*Exploitation*	*#PC*	*#HPC*
	check-types	11.1.2	573	9,983,393	0.36	26.5s	225.7s	88	2
	DumperJS	1.3.1	284	6,797	0.57	2.9s	580.4s	28	18
	deep-extend	0.6.0	83	39,395,270	0.35	5.5s	45.0s	3	6
	deep-copy	1.4.2	60	402,884	0.44	1.2s	49.2s	22	3
	deepmerge	4.2.2	325	39,856,800	0.58	4.9s	53.3s	12	3
	fast-clone	1.5.13	87	23,424	0.43	1.3s	44.2s	11	4
	fast-stringify	2.0.0	184	33,4536	0.34	1.3s	N/A	4	0
	immutability-helper	3.0.1	259	1,395,820	0.32	0.8s	N/A	10	0
	iap	1.1.1	1250	8,227	0.32	0.5s	17.5s	12	5
	Js-yaml	3.13.1	5719	60,478,990	0.24	47.8s	172.4s	40	14
	jsonfile	5.0.0	110	5,637	0.29	1.5s	N/A	42	0
	js2xmlparser	4.0.1	364	2,796,779	0.47	67.4s	94.1s	45	2
	json-to-pretty-yaml	1.2.2	163	1,052,996	0.34	2.1s	5.1s	19	2
	just-extend	4.1.0	41	7,891,960	0.44	1.2s	13.46s	10	3
	kind-of	6.0.2	97	196,448,574	0.56	1.2s	49.1s	16	16
	mailgun-js	0.22.0	6569	1,200,173	0.61	614.0s	485.9s	22	6
	map-obj	4.1.0	76	51,062,828	0.78	1.0s	26.8s	14	6
	merge-deep	3.0.2	162	12,158,104	0.58	2.5s	15.2s	6	5
	mongo-parse	2.1.0	1435	1,291	0.13	1s	N/A	15	0
	mongodb-extjson	3.0.3	8845	42,141	0.20	6s	75.5s	23	9
	node-cache	5.1.0	618	2,917,617	0.33	1.3s	1.11s	14	6
	object-hash	2.0.2	4277	20,002,794	0.33	4.2s	40.7s	15	2
	Object-is	1.0.1	56	25,466,395	0.53	1.6s	N/A	6	0
	papaparse	5.1.1	4710	1,290,026	0.08	8.9s	32.6s	11	11
	set-value	3.0.2	83	60,184,464	0.57	1.0s	17.1s	4	6
	table	5.4.6	2283	36,535,762	0.38	11.5s	39.3s	7	3
	WriteJsonFile	4.2.1	160	6,792,576	0.54	6.8s	N/A	12	0
	vnopts	1.0.2	2571	166,521	0.22	13s	N/A	3	0
	xtend	4.0.2	106	64,552,908	0.71	1.9s	78.5s	15	6
Web	cezerin	0.33.0	48808	1,871 (g)	0.37	63s	740s	9	49
	connect	3.7.0	125	15,621,960	0.20	46s	N/A	4	0
	derby	0.10.27	5060	1,156	0.12	237s	N/A	5	0
	Datalize	0.3.4	628	231	0.27	71s	91.2s	69	12
	express	4.17.1	1829	55,134,711	0.14	62.0s	14.0s	1	2
	Express-form	0.12.6	1569	4,183	0.31	1.3s	2.2s	17	2
	express-cart	1.1.16	6904	1,554 (g)	0.14	45s	N/A	8	0
	ghost	3.39.3	58776	32,719	0.32	71s	88.4s	468	5
	mongo-express	0.54.0	2789	6,965	0.30	75s	29s	45	25
	nodebb	1.4.0	70549	55	0.14	38s	N/A	637	0
	total.js	3.3.0	38214	14,267	0.14	340s	N/A	6	0