Securing Node-RED Applications

Mohammad M. Ahmadpanah[1(✉)], Musard Balliu[2], Daniel Hedin[1,3], Lars Eric Olsson[1], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology, Gothenburg, Sweden
mohammad.ahmadpanah@chalmers.se
[2] KTH Royal Institute of Technology, Stockholm, Sweden
[3] Mälardalen University, Västerås, Sweden

Abstract. Trigger-Action Platforms (TAPs) play a vital role in fulfilling the promise of the Internet of Things (IoT) by seamlessly connecting otherwise unconnected devices and services. While enabling novel and exciting applications across a variety of services, security and privacy issues must be taken into consideration because TAPs essentially act as persons-in-the-middle between trigger and action services. The issue is further aggravated since the triggers and actions on TAPs are mostly provided by third parties extending the trust beyond the platform providers.

Node-RED, an open-source JavaScript-driven TAP, provides the opportunity for users to effortlessly employ and link *nodes* via a graphical user interface. Being built upon Node.js, third-party developers can extend the platform's functionality through publishing nodes and their wirings, known as *flows*.

This paper proposes an essential model for Node-RED, suitable to reason about nodes and flows, be they benign, vulnerable, or malicious. We expand on attacks discovered in recent work, ranging from exfiltrating data from unsuspecting users to taking over the entire platform by misusing sensitive APIs within nodes. We present a formalization of a runtime monitoring framework for a core language that soundly and transparently enforces fine-grained allowlist policies at module-, API-, value-, and context-level. We introduce the monitoring framework for Node-RED that isolates nodes while permitting them to communicate via well-defined API calls complying with the policy specified for each node.

1 Introduction

Trigger-Action Platforms (TAPs) play a vital role in fulfilling the promise of the Internet of Things (IoT). TAPs empower users by seamlessly connecting otherwise unconnected *trigger* and *action* services. Popular TAPs like IFTTT [24] and Zapier [57], as well as open-source alternatives like Node-RED [36], offer users the ability to operate simple trigger-action *applications* (or, for short, *apps*) such as "Tweet your Instagrams as native photos on Twitter", "Get emails via Gmail with new files added to Dropbox'', and "Covid-19 live Ticker via Twitter".

D. Dougherty et al. (Eds.): Guttman Festschrift, LNCS 13066, pp. 1–21, 2021.
https://doi.org/10.1007/978-3-030-91631-2_1

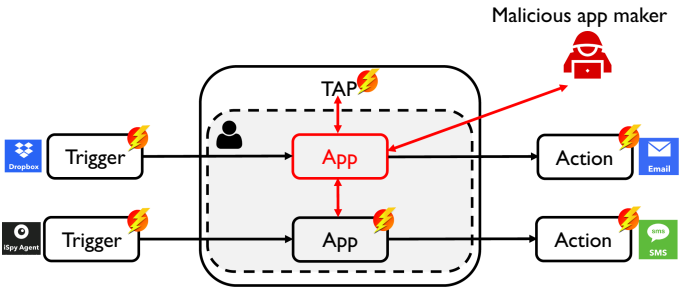

Fig. 1. Threat model of a malicious app deployed on a single-user TAP [3].

A TAP is effectively a "person-in-the-middle" between trigger and action services. While greatly benefiting from the possibility of apps to run third-party code, TAPs are subject to critical security and privacy concerns. Attacks by third-party app makers on the platform may lead to compromising the integrated trigger and action services. Figure 1 illustrates how a malicious app deployed by a user on a TAP like Node-RED can compromise the associated trigger and action services, another installed app, and the platform [3]. Depending on the security configuration of the TAP's deployment, the attacker may also compromise the underlying system.

In contrast to proprietary centralized platforms such as IFTTT and Zapier, Node-RED can be entirely run on a user's own server. Node-RED is an open-source platform built on top of Node.js, enabling users to inspect and customize the source code of the platform and the apps as desired. Moreover, Node-RED relies on JavaScript packages from third parties to facilitate the integration of new functionalities. In fact, Node.js *nodes* are the basic building blocks of Node-RED apps (also named as *flows*), freely available on the Node Package Manager (NPM) [43] and automatically added to the Node-RED Library [41]. Node-RED is inspectable and thus can be verified by users in terms of the platform's correctness and security. Third-party apps integrated into the underlying platform, however, can still threaten the security of the users and the entire system.

The starting point of this paper is the recently identified attacks on Node-RED by malicious nodes, ranging from exfiltrating users' sensitive data to taking over the platform and the host system [3]. A Node-RED flow is technically a static representation of how nodes are wired together; therefore, a malicious node controlled by an attacker can be employed in any user-defined or third-party flows, resulting in malicious behaviors.

This observation motivates the need for controlling APIs invoked in nodes to ensure the security of the platform and the users. Although the enforcement mechanism must guarantee security, it also should restrict access only if it is against the node's policy, according to the *least privilege principle* [47]. Only the APIs which are necessary for the intended functionality should be accessible in a node; thus, if none of the APIs of a module are required, loading of the module must be denied. In some cases, the interaction through APIs needs to be

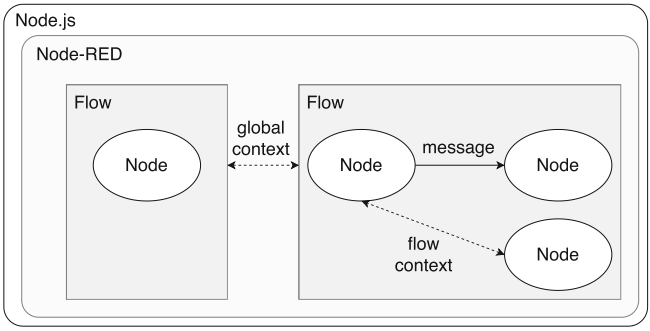

Fig. 2. Node-RED architecture [3].

value-sensitive when an API call should be permitted only with a list of defined arguments, for instance, when it comes to allowing a node to make an HTTPS request to a specific trusted domain. Furthermore, Node-RED makes use of both message passing and the shared context [40] to exchange information between nodes and flows, and both types of exchange need to be secured. Previous work proposes SandTrap [3], a runtime monitor for JavaScript-driven TAPs. However, SandTrap's security guarantees are argued only informally.

Motivated by SandTrap, this work is a step toward formally understanding how to monitor Node-RED apps. We present a sound and transparent monitoring framework for Node-RED for enforcing fine-grained allowlist policies at module-, API-, value-, and context-level. In the following, we discuss Node-RED along with overviewing platform- and app-level vulnerabilities and attacks (Sect. 2); propose an essential model for Node-RED, suitable to reason about nodes and flows, be they benign, vulnerable, or malicious; and present a monitoring framework to express and enforce fine-grained security policies, proving its soundness and transparency (Sect. 3).

2 Node-RED Vulnerabilities

Node-RED is "a programming tool for wiring together hardware devices, APIs and online services", which provides a way of "low-code programming for event-driven applications" [36]. As an open-source platform, Node-RED is mainly targeted for deployment as a single-user platform, although it is also available on the IBM Cloud platform [23]. We overview the architecture of Node-RED (Sect. 2.1) and explain two types of vulnerabilities with respect to our attacker model, i.e., malicious app makers: (i) *platform-level isolation vulnerabilities* (Sect. 2.2) and (ii) *application-level context vulnerabilities* (Sect. 2.3). Our discussion expands the condensed presentation of these vulnerabilities from previous work [3].

```
module.exports = function(RED){
  function NodeName(config){
    RED.nodes.createNode(this, config);
    var node = this;
    // register a callback when a message is received...
    node.on("input", function(msg){
      ... // functionality of node
      node.send(msg); // or an array of messages for multiple
                      outputs
    });
  }
  RED.nodes.registerType("type-name", NodeName);
}
```

Fig. 3. Node-RED node structure.

2.1 Node-RED Platform

Figure 2 illustrates the Node-RED architecture, consisting of a collection of apps, known as *flows*, linking components called *nodes*. The Node-RED runtime is built on the Node.js environment and can run multiple flows simultaneously. It supports inter-node and inter-flow communication via direct messages through the wiring between nodes in a flow, while the *flow* and the *global* contexts [40] are alternative communication channels between the nodes of a flow and across the nodes of different flows, respectively.

A node is a reactive Node.js application triggered by receiving messages on at most one input port (dubbed *source*) and sending the results of (side-effectful) computations on output ports (dubbed *sinks*), which can be potentially multiple, unlike the input port. Figure 3 illustrates the code structure of a Node-RED node. A special type of node without sources and sinks, called *configuration* node, is used for sharing configuration data, such as login credentials, between multiple nodes.

A flow is a representation of nodes connected together. End users can either create their own flows on the platform's environment or deploy existing flows provided by the official Node-RED catalog [33] and by third parties [41]. As shown in Fig. 4, flows are JSON files wiring node sinks to node sources in a graph of nodes where messages, represented by JavaScript objects, are passed between. Multiple messages can be sent by any given node, although instances of a single message can be repeatedly sent to multiple nodes as well. To facilitate end-user programming [55], flows can be shown visually via a graphical user interface and deployed in a push-button fashion. As an example, Fig. 5 demonstrates a flow that retrieves earthquake data for logging and notifying the user whenever the magnitude exceeds a threshold. Specifically, the flow retrieves data of the recent quakes (either periodically or by clicking on the button), parses the given CSV file, and shows the data (stored in `msg.payload`) to the user. For each magnitude value exceeding the specified threshold, it also branches and the payload triggers an alarm notification.

```
[                          // list of nodes
  {                        // node 0
    /* parameters of interest in every node */
    id: NODE0,             // unique ID of node, string
    type: function         // type of node, string
    wires: [               // array of array of strings
      [ NODE1 ],           // first output port to node 1
      [ NODE2, NODE3 ]     // second output port to nodes 2 and 3
    ],
    ...                    // other parameters
  },
  ...                      // other nodes
]
```

Fig. 4. Node-RED flow structure.

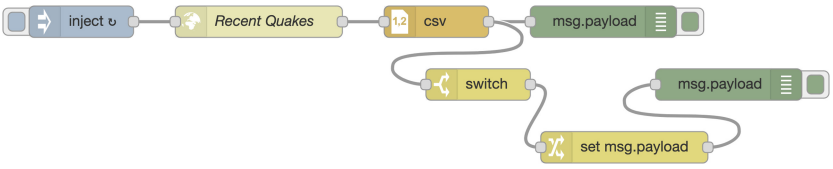

Fig. 5. Earthquake notification and logging.

In Node-RED, *contexts* provide a shared communication channel between different nodes without using the explicit messages that pass through a flow [40]. Therefore the node wiring visible in the user interface reflects only a part of the information flows that are possible in the flow. It introduces an implicit channel that is not visible to the user via the graphical interface of a flow. Node-RED defines three scope levels for the contexts: (i) *Node*, only visible to the node that sets the value, (ii) *Flow*, visible to all nodes on the same flow, and (iii) *Global*, visible to all nodes on any flow. For instance, a sensor node may regularly update new values in one flow, while another flow may return the most recent value via HTTP. By storing the sensor reading in the global shared context, the data is accessible for the HTTP flow to return.

Node-RED security relies on the platform running on a trusted network, ensuring that users' sensitive data is processed in an environment controlled by the users. The official documentation [37] also includes programming patterns for securing Node-RED apps. These patterns include basic authentication mechanisms to control access to nodes and wires. The official node `Function` runs user-provided code in a `vm` sandbox [42], suggesting that it may protect the user from unauthorized access. However, the `vm`'s sandbox "is not a security mechanism" [42], and there are known breakouts [26].

TAPs generally lack the means to specify user's security policies [9]. Fortunately, Node-RED's user-centric setting enables us to *interpret* intended security policies. In fact, Node-RED's GUI for flows provides an intuitive way to inter-

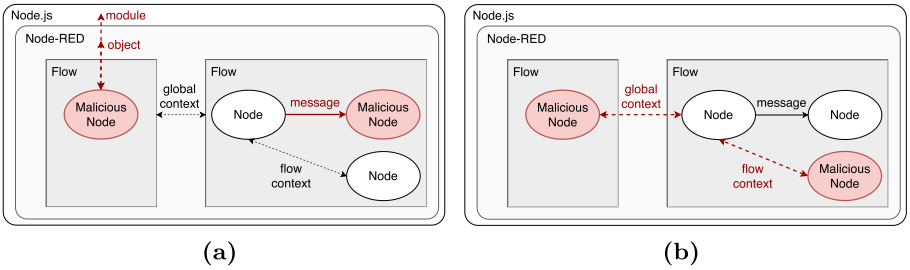

Fig. 6. Node-RED vulnerabilities: (a) Isolation vulnerabilities; (b) Context vulnerabilities [3].

pret top-level user policies; it is reasonable to consider that the user endorses the flow of information between the nodes connected by the graph that depicts a flow in the GUI. For instance, the Earthquake notification flow in Fig. 5 implies a policy where notification data may only flow to the notification message. Only the `Inject` node can trigger updates. The policy allows no other node (from any flow) to tamper with the `Recent Quakes` node, preventing any malicious node from corrupting the source of quake information. Such an interpretation provides us with a *baseline* security policy. For more fine-grained policies, e.g., the list of permitted URLs to retrieve the recent quakes, it is reasonably presumed that the node developer designs these *advanced* policies since they know the precise specification of the node. The provided policies can later be vetted by the platform and the user, before deploying the node. SandTrap [3] offers a policy generation mechanism to aid developers in designing the policies, enabling both baseline and advanced policies customized by developers or users to express fine-grained app-specific security goals.

In the following, we discuss Node-RED attacks and vulnerabilities that motivate enriching the policy mechanism with different granularity levels. These policies will further be formalized in Sect. 3.

2.2 Platform-Level Isolation Vulnerabilities

While facilitating the integration and automation of different services and devices, due to imposing insufficient restrictions on nodes, Node-RED is exploitable by malicious node makers. All APIs provided by the underlying runtimes, Node-RED and Node.js, are accessible for node developers, as well as the incoming messages within a flow. As shown in Fig. 6a, there are various attack scenarios for malicious nodes [3]. At the Node.js level, an attacker can create a malicious Node-RED node including powerful Node.js libraries like `child_process`, allowing the attacker to execute arbitrary shell commands with `exec`, e.g., taking full control of the user's system [44]. Restricting library access is laborious in Node-RED; while access to a sensitive library like `child_process` is required for the functionality of Node-RED, attackers can exploit trust propagation due to transitive dependencies in Node.js [45,58]. A malicious node enables

the attacker to compromise the confidentiality and integrity of sensitive data and libraries stored by other flows in the global context. A malicious node within a sensitive flow may also indirectly read and modify sensitive data by manipulating the flow context.

At the platform level, the main object in the Node-RED structure, named `RED` [39], is also vulnerable. There are different ways for a malicious node to misuse the `RED` object, such as aborting the server (e.g., by `RED.server._events = null`) or introducing a covert channel shared between multiple instances of the node in different flows by modifying existing properties or adding new properties to the `RED` object (like `RED.dummy`). Therefore, *access control at the level of modules and shared objects* is necessary for Node-RED nodes.

On the other hand, a malicious node can directly manipulate incoming messages resulting in accessing or tampering with the sensitive data. As a subtle example of this scenario to invade users' privacy, the official Node-RED email can be modified to send the email body to the original recipient and also forward a copy of the message to an attacker's address. The benign code sets the sending options `sendopts.to` to contain only the address of the intended recipient:

```
sendopts.to = node.name || msg.to; // comma separated list
    of addresses
```

It can be modified to the following by a malicious node maker to include the attacker's address as well:

```
sendopts.to = (node.name || msg.to) + ", me@attacker.com";
```

In this example, we demonstrate that an attacker can alter the value that is placed as the argument of an API call, which is necessary for the functionality of the node, to steal sensitive information of the user without being noticed. As a result, the combination of function identity and its arguments needs to be considered in security checks. This attack motivates us to provide *fine-grained access control at the level of APIs and their input parameters.*

We refer the interested reader to other types of investigated vulnerabilities in Node-RED [3], such as the impact of compromised package repository and *name squatting* [58] attack. The latter is critical since the "type" of nodes (what flows use to identify them) is simply a string, which multiple packages can possibly match. A flow defined by a third party can include the attacker's malicious node unless the user inspects each and every node to verify that there are no deviations from the expected "type" string.

The empirical study shows the implications of such attacks [3]: privacy violations may occur in 70.40% of Node-RED flows and integrity violations in 76.46%. The vast number of privacy violations in Node-RED reflects the power of malicious developers to exfiltrate private information.

2.3 Application-Level Context Vulnerabilities

Node-RED uses various levels of the shared context to exchange data across nodes and flows in an implicit manner. Figure 6b depicts the attack scenarios to

exploit vulnerabilities by reading and writing to libraries and variables shared in the global and flow contexts [3]. The *Node* context shares data with the node itself; thus only the shared contexts at the levels of *Flow* and *Global* are intriguing to investigate. Malicious nodes in these scenarios can exploit other vulnerable Node-RED nodes, even if the platform is secured against attacks in Sect. 2.2.

Several Node-RED core nodes [38] make use of the shared context for their purposes, including the nodes executing any JavaScript function (`Function`), triggering a flow (`Inject`), generating text to fill out a template (`Template`), routing outgoing messages to branches of a flow by evaluating a set of rules (`Switch`), and modifying message properties and setting context properties (`Change`). It is shown that more than 228 published flows utilize flow or global context in at least one of the member nodes and more than 153 of the published Node-RED packages directly read from or modify the shared context [3].

The main purpose of using the shared context is data communication between nodes. Malicious operations on the shared data, such as tampering, adding, or erasing, may lead to integrity and availability attacks, as well as to disrupting the functionality entirely. As a real-world example, the Node-RED flow "Water Utility Complete Example" is vulnerable considering misuse of the *Global* context. Targeting SCADA systems, this flow manages two tanks and two pumps; the first pump pumps water from a well into the first tank, and the second pump transfers water from the first to the second tank. The status of the tanks are stored in globally shared variables as follows:

```
global.set("tank1Level", tank1Level);
global.set("tank1Start", tank1Start);
global.set("tank1Stop",  tank1Stop);
```

Later, to determine whether a pump should start or stop, the flow retrieves the shared status from the *Global* context:

```
var tankLevel  = global.get("tank1Level");
var pumpMode   = global.get("pump1Mode");
var pumpStatus = global.get("pump1Status");
var tankStart  = global.get("tank1Start");
var tankStop   = global.get("tank1Stop");
if (pumpMode === true && pumpStatus === false && tankLevel
    <= tankStart){
  // message to start the pump
}
else if (pumpMode === true && pumpStatus === true &&
    tankLevel >= tankStop){
  // message to stop the pump
}
```

A malicious node installed by the user and deployed in the platform could alter the context relating to the tank's reading to either exhaust the water flow (never start) or cause physical damage through continuous pumping (never stop).

One can also use the context feature to share resources such as common libraries. In addition to integrity and availability concerns, this approach opens up possibilities for exfiltrating private data. An attacker can encapsulate a library to collect any sensitive information sent to the library. For instance, by modifying the `opencv` shared library inside a malicious node, the attacker can exfiltrate private information of video streaming for motion detection. More details and examples of such vulnerabilities are also studied [3].

These vulnerabilities motivate the need for monitoring *access control at the level of context*.

3 Formalization

Section 2 motivates the need for secure integration of untrusted code in general and restricting node-to-node and node-to-environment communications (i.e., between nodes, library functions, and contexts) for Node-RED in particular. To achieve this, we propose a runtime monitoring framework capable of enforcing allowlist policies at the granularity of modules, APIs and their input parameters, and variables used in the shared context. Our runtime framework formalizes the core of the flow-based programming model of Node-RED and was the basis when developing the JavaScript monitor SandTrap [3].

This section presents a security model for Node-RED apps and characterizes the essence of a fine-grained access control monitor for the platform. We show how to formalize and enforce security policies for nodes at the level of APIs and their values, along with the access rights to the shared context. Our main formal results are the soundness and transparency of the monitor.

3.1 Language Syntax and Semantics

Syntax. We define a core language to capture the reactive nature of nodes and flows. Nodes are reactive programs triggered by input messages to execute the code of an event handler and potentially produce an output message. Flows model connections between nodes by specifying the destination nodes for each node's output port. Given the set of member nodes with their handlers, it is sufficient to state the successor nodes on each output port to construct a flow.

A flow is syntactically defined as a set of nodes, written $F = \{N_k \mid k \in K\}$, where K is a finite subset of $\mathbb{N}$, and k indicates a unique node identifier. A Node-RED environment may execute flows simultaneously and the global environment is defined by a set of flows, written $G = \{F_l \mid l \in L\}$, where L is a finite subset of $\mathbb{N}$, and l denotes a unique flow identifier. Based on a generalization of Node-RED nodes, Fig. 7 presents the syntax of a reactive language inspired by Devriese and Piessens [17], where Val, Var, and Fun denote the set of all possible values, variables, and functions, respectively. A handler $handler\,(x)\{c\}$ is defined by an input parameter x, which is bound in a command c to perform a computation. While most commands are standard imperative constructs, we

use command $send(e, i)$ to pass the value of expression e to the node's output port identified by i. For simplicity, we use functions $f(e)$ to model module imports, API calls, user-defined functions, and primitive operations such as addition and concatenation. To model the shared context, we distinguish between *node* variables Var_N, *flow* variables Var_F, and *global* variables Var_G such that $Var = Var_N \uplus Var_F \uplus Var_G$.

$$
\begin{aligned}
&v \in Val,\ x \in Var,\ f \in Fun,\ i \in \mathbb{N}\\
&e ::= v \mid x \mid f(e)\\
&c ::= skip \mid x := e \mid if\ e\ then\ c\ else\ c \mid while\ e\ do\ c \mid c\,;\,c \mid send\ (e, i)\\
&h ::= handler\ (x)\{c\}
\end{aligned}
$$

Fig. 7. Syntax of node handlers.

Semantics. We model the execution of Node-RED apps by defining the node semantics, flow semantics, and global semantics, respectively. Our trace-based semantics records the sequence of events produced during the execution of a flow. We use these events to define a semantic security condition that our monitor will enforce in a sound and transparent manner.

Node Semantics. A node $N = \langle config, wires, l\rangle_k$ is defined by a node configuration $config$, an array $wires$ that specifies the connected nodes in the flow associated with output ports, an identifier l that indicates the flow that the node belongs to, and a unique node identifier k. Index k refers to an element of node N_k, as in $config_k$ for the configuration of node k.

A node configuration $config = \langle c, M, I, O\rangle$ stores the state of the node during the execution, where c is a command, a handler, or a termination signal ($stop$), $M = [m_N, m_F, m_G]$ represents the memory for the three scopes of node ($m_N : Var_N \to Val$), flow ($m_F : Var_F \to Val$), and global ($m_G : Var_G \to Val$), where Var_N, Var_F, and Var_G are disjoint sets, I is the input channel, and O is the array of output channels, reflecting that a node has one input port and as many output ports as it requires . We model an input (output) channel as a sequence of values that a node receives (sends). A class of nodes, called *inject* nodes, is triggered by external events such as button click or time. Inject nodes send new messages to a flow, thus triggering the execution of the flow. The $wires$ array records the nodes that can read the content of the output channel for the corresponding output port. A node receives a message if the node identifier is listed in $wires$ among the recipients of the output port assigned in a send command.

Trace-Based Semantics. Figure 8 illustrates the small-step semantics of nodes. We annotate transitions with the trace of events thus generated, where $\to \subseteq Config \times Config$ and $\Downarrow : (Exp \times Mem) \to Val$. A trace T is a finite sequence of events $t_k \in E$ defined by variable reads $R_k(x)$, variable writes $W_k(x)$, or function calls $f_k(v)$ generated by the execution of node k in a flow.

Expression Evaluation

$$\frac{}{\langle v, M_k\rangle \Downarrow v}\ (\textsc{Value}) \qquad \frac{\langle e, M_k\rangle \Downarrow^{T_k} v}{\langle f(e), M_k\rangle \Downarrow^{T_k \cdot f_k(v)} \bar{f}(v)}\ (\textsc{Call}) \qquad \frac{}{\langle x, M_k\rangle \Downarrow^{R_k(x)} M_k(x)}\ (\textsc{Read})$$

Command Evaluation

$$\frac{I = I'.v \qquad x \in Var_N}{\langle handler(x)\{c\}, M, I, O\rangle_k \rightarrow \langle c, M[x \mapsto v], I', O\rangle_k}\ (\textsc{Input})$$

$$\frac{}{\langle skip, M, I, O\rangle_k \rightarrow \langle stop, M, I, O\rangle_k}\ (\textsc{Skip})$$

$$\frac{\langle e, M_k\rangle \Downarrow^{T_k} v \qquad M'_k = M_k[x \mapsto v]}{\langle x := e, M, I, O\rangle_k \xrightarrow{T_k \cdot W_k(x)} \langle stop, M', I, O\rangle_k}\ (\textsc{Write})$$

$$\frac{c = \mathit{if}\ e\ \mathit{then}\ c_{true}\ \mathit{else}\ c_{false} \qquad \langle e, M_k\rangle \Downarrow^{T_k} b}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle c_b, M, I, O\rangle_k}\ (\textsc{If})$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{body} \qquad \langle e, M_k\rangle \Downarrow^{T_k} \mathit{true}}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle c_{body}; c, M, I, O\rangle_k}\ (\textsc{While-T})$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{body} \qquad \langle e, M_k\rangle \Downarrow^{T_k} \mathit{false}}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle stop, M, I, O\rangle_k}\ (\textsc{While-F})$$

$$\frac{\langle c_1, M, I, O\rangle_k \xrightarrow{T_k} \langle c'_1, M', I', O'\rangle_k}{\langle c_1; c_2, M, I, O\rangle_k \xrightarrow{T_k} \langle c'_1; c_2, M', I', O'\rangle_k}\ (\textsc{Seq-1})$$

$$\frac{}{\langle stop; c, M, I, O\rangle_k \rightarrow \langle c, M, I, O\rangle_k}\ (\textsc{Seq-2})$$

$$\frac{c = send(e, i) \qquad \langle e, M_k\rangle \Downarrow^{T_k} v \qquad O'[i] = O[i].v}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle stop, M, I, O'\rangle_k}\ (\textsc{Output})$$

Fig. 8. Node semantics.

Expression evaluation is standard and records the sequence of events produced during the evaluation, where M_k denotes the memory M in $\langle c, M, I, O\rangle_k$. Command evaluation models the execution of a node's handler. The handler executes whenever there is a message in the input channel I by consuming the message and updating the memory accordingly. Assignments operate in a similar manner and record the trace of events produced by variable reads and writes. An assignment updates the memory M_k to M'_k, subsequently triggering an update of the flow and global memories, as stated in the rule (Step) in Fig. 9 and in the rule (Global) in Fig. 10. Send commands evaluate the expression e in the current memory, update the associated output channel, and record the trace of events. The index k distinguishes between events of different nodes. We write $\rightarrow^*$ for the reflexive and transitive closure of the $\rightarrow$ relation, and $\rightarrow^n$ for the n-step execution of $\rightarrow$.

Flow and Global Semantics. We lift node semantics to formalize the semantics of flows and the environment. A global configuration $G = \langle m_G, \{F_l \mid l \in L\}\rangle$ consists of a global shared memory m_G and a finite set of flows that are executing concurrently, where $L \subset \mathbb{N}$ is the set of flow identifiers. A flow configuration $F = \langle m_F, \{N_k | \, k \in K\}\}\rangle_l$ is a tuple consisting of a flow shared memory m_F, a finite set of nodes where $K \subset \mathbb{N}$ is the set of node identifiers, and l is the flow identifier. We use $Nodes(F_l)$ for the set of nodes in a specific flow and $Flows(G)$ for the set of flows in the environment. Nodes are distinguished by unique node identifiers in the environment and the node N_k can be present in only one flow. To unify the trigger point of the flow, we assume that a flow has only one inject node and denote it by N_l where $N_l \in Nodes(F_l)$; in practice, it can be considered as a dummy node which is the predecessor of all the inject nodes of the flow.

We model a flow by linking the output channels of a node to the input channels of the next ones based on the flow specification. Note that a node can have more than one output channel but only one input channel. The inject node of a flow, which does not appear in any of the *wires* arrays, triggers the flow execution by injecting a new message. An initial value is assigned to the input channel of the inject node to model the behavior of the external event such as a button click. We write $Exec(F_l, v_l)$ to refer to executions of a flow F_l with an initial value v_l. Also, $Exec(G, V)$ denotes executions of the environment G with the set of initial values $V = \{(N_l, v_l) \mid F_l \in Flows(G)\}$ for the member flows.

We remark that message passing in Node-RED is asynchronous and message objects traverse the graph in a non-deterministic manner, as reported in the documentation ("no assumptions should be made about ordering once a flow branches" [35] and "flows can be cyclic" [34]). Hence, we model the execution of nodes in a flow and the environment, as shown in Figs. 9 and 10, respectively. We overload the notation $\rightarrow$ for transitions between flow and global configurations. In a nutshell, the flow and global semantics implements the non-deterministic behavior of flows and the environment, and lifts the node semantics to ensure that the flow of messages follows the flow specification.

The intuition of the rules is that the inject node of a flow, i.e., the node N_l of the flow F_l, starts the execution by consuming the initial value (rule Init), and then the execution continues according to the node semantics (rule Step). When a node reaches a send command, it adds the output value to the input channels of the next nodes in the flow; the output value transmits out to the output channel indicated by the send command and the input channels of all nodes in the corresponding elements of the array *wires* get updated with the value (rule Send); $wires_k$ denotes the array *wires* in $\langle config, wires, l\rangle_k$. The execution proceeds until it terminates and gets back to the initial state, ready to consume the next value in the input channel (rule Term). Note that nodes are running concurrently; any of the ready nodes can make one execution step. The only rule in the global semantics (rule Global) shows that any of the flows with at least one ready node can make an execution step.

$$\frac{\begin{array}{c} I_l = v_l \qquad \forall N_k \in (Nodes(F_l) \setminus N_l).\, I_k = \varnothing \\ M_l = [m_N, m_F, m_G] \qquad M_l' = [m_N', m_F, m_G] \\ config_l = \langle handler(x)\{c\}, M, I, O\rangle_l \qquad config_l' = \langle c, M[x \mapsto v_l], \varnothing, O\rangle_l \\ config_l \to config_l' \qquad N_l = \langle config_l, wires, l\rangle_l \qquad N_l' = \langle config_l', wires, l\rangle_l \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \to \langle m_F, (Nodes(F_l) \setminus \{N_l\}) \cup \{N_l'\}\rangle_l} \quad (\textsc{Init})$$

$$\frac{\begin{array}{c} I_l = \varnothing \qquad M_k = [m_N, m_F, m_G] \qquad M_k' = [m_N', m_F', m_G'] \\ config_k = \langle c, M, I, O\rangle_k \qquad config_k' = \langle c', M', I', O\rangle_k \\ config_k \xrightarrow{T_k} config_k' \\ N_k = \langle config_k, wires, l\rangle_k \qquad N_k' = \langle config_k', wires, l\rangle_k \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \xrightarrow{T_k} \langle m_F', (Nodes(F_l) \setminus \{N_k\}) \cup \{N_k'\}\rangle_l} \quad (\textsc{Step})$$

$$\frac{\begin{array}{c} config_k = \langle send(e, i); c, M, I, O\rangle_k \qquad config_k' = \langle stop; c, M, I, O'\rangle_k \\ O_k'[i] = O_k[i].v \qquad config_k \xrightarrow{T_k} config_k' \\ N_k = \langle config_k, wires, l\rangle_k \qquad N_k' = \langle config_k', wires, l\rangle_k \\ \omega = \{N_k\} \cup \{N_j \mid j \in wires_k[i]\} \\ \omega' = \{N_k'\} \cup \{N_j' \mid j \in wires_k[i],\ I_j' = v.I_j\} \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \xrightarrow{T_k} \langle m_F, (Nodes(F_l) \setminus \omega) \cup \omega'\rangle_l} \quad (\textsc{Send})$$

$$\frac{\begin{array}{c} config_k = \langle stop, M, I, O\rangle_k \qquad config_k' = \langle handler(x)\{c\}, M, I, O\rangle_k \\ N_k = \langle config_k, wires, l\rangle_k \qquad N_k' = \langle config_k', wires, l\rangle_k \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \to \langle m_F, (Nodes(F_l) \setminus \{N_k\}) \cup \{N_k'\}\rangle_l} \quad (\textsc{Term})$$

Fig. 9. Flow semantics.

$$\frac{\begin{array}{c} M_k = [m_N, m_F, m_G] \qquad M_k' = [m_N', m_F', m_G'] \\ F_l = \langle m_F, Nodes(F_l)\rangle_l \qquad F_l' = \langle m_F', Nodes(F_l')\rangle_l \\ F_l \xrightarrow{T_k} F_l' \end{array}}{\langle m_G, Flows(G)\rangle \xrightarrow{T_k} \langle m_G', (Flows(G) \setminus \{F_l\}) \cup \{F_l'\}\rangle} \quad (\textsc{Global})$$

Fig. 10. Global semantics.

Generally speaking, any node that is able to progress continues the execution for one execution step, and it might affect the flow and global contexts. An execution step of a node corresponds to one execution step of the flow it belongs to and one execution step of the environment. Considering the non-deterministic behavior of Node-RED's scheduler, any ready node can be selected for the next execution step.

3.2 Security Condition and Enforcement

We leverage our trace-based semantics to define a semantics-based security condition. The condition is parametric on node-level security policies, represented as allowlists of API calls and accesses to the shared context. Then, we present

the semantics of a fine-grained node-level monitor and prove its soundness and transparency with respect to the security condition.

Security Condition. We extend the definition of nodes with allowlist policies $N = \langle config, wires, l, P, V, S \rangle_k$, where $P \subseteq APIs \subseteq Fun$ describes permitted API functions, $V : P \rightarrow 2^{Val}$ defines the allowlist of arguments for each API function, and S specifies read/write permissions on the shared global and flow variables, such that $S = \{(x, RW) \mid x \in Var_F \uplus Var_G, RW \in \{R, W\}\}$.

The security condition matches the trace of events produced by the semantics with the allowlist policies to check that any event produced by an execution is permitted by the policy.

Definition 1 (*Event Security*). *Let t_k be an event emitted from an execution of node N_k. We define a secure event with respect to $\langle P_k, V_k, S_k \rangle$, written $secure(t_k, \langle P_k, V_k, S_k \rangle)$, as follows:*

$$\begin{aligned} secure(R_k(x),\ \langle P_k, V_k, S_k \rangle) &\triangleq x \in Var_F \cup Var_G \Rightarrow (x, R) \in S_k \\ secure(W_k(x),\ \langle P_k, V_k, S_k \rangle) &\triangleq x \in Var_F \cup Var_G \Rightarrow (x, W) \in S_k \\ secure(f_k(v),\ \langle P_k, V_k, S_k \rangle) &\triangleq f \in APIs \Rightarrow f \in P_k \wedge v \in V_k(f). \end{aligned}$$

We lift the security of events to define the security condition for node traces $secure(T_N)$, flows traces $secure(T_F)$, and global traces $secure(T_G)$ as expected. A finite sequence of events forms a trace. Hence a trace is secure if all its events are secure. We define trace security by the conjunction of security checks on the composing events.

Definition 2 (*Trace Security*). *Trace T is secure, written $secure(T)$, if*

$$T = t_k.T' \Rightarrow secure(t_k, \langle P_k, V_k, S_k \rangle) \wedge secure(T').$$

A node starts executing when it receives a value over its input channel. An execution of a node is secure if the corresponding trace is secure, according to the node policy.

Definition 3 (*Node-Level Security*). *The execution of a node $N_k = \langle config, wires, l, P, V, S \rangle_k$ with an input message $I = v$ is secure with regard to $\langle P_k, V_k, S_k \rangle$ if each step of the node execution complies with $\langle P_k, V_k, S_k \rangle$, i.e.,*

$$\forall \langle c', M', I', O' \rangle_k . \langle handler(x)\{c\}, M, v, O \rangle_k \xrightarrow{T_k}{}^{*} \langle c', M', I', O' \rangle_k \Rightarrow secure(T_k).$$

We now define the security of Node-RED app executions based on the flow and global semantics. The inject node of a flow initiates the flow execution, and it triggers other nodes by traversing the flow graph. At the global level, nodes in flows generate events while they are executing concurrently in the environment. We present flow and global execution security for the trace of events produced by their nodes at each execution step.

Expression Evaluation

$$\frac{secure(R_k(x),\ \langle P_k,\ V_k, S_k\rangle)}{\langle x, M_k\rangle \Downarrow_{\mathcal{M}}^{R_k(x)} M_k(x)} \quad (\textsc{Read}_{\mathcal{M}})$$

$$\frac{\langle e, M_k\rangle \Downarrow^{T_k} v \qquad secure(f_k(v),\ \langle P_k,\ V_k, S_k\rangle)}{\langle f(e), M_k\rangle \Downarrow_{\mathcal{M}}^{T_k \cdot f_k(v)} \bar{f}(v)} \quad (\textsc{Call}_{\mathcal{M}})$$

Command Evaluation

$$\frac{secure(\,W_k(x),\ \langle P_k,\ V_k, S_k\rangle) \qquad \langle e, M_k\rangle \Downarrow^{T_k} v \qquad M' = M[x \mapsto v]}{\langle x := e, M, I, O\rangle_k \xrightarrow{T_k \cdot W_k(x)}_{\mathcal{M}} \langle stop, M', I, O\rangle_k} \quad (\textsc{Write}_{\mathcal{M}})$$

Fig. 11. Excerpt of monitor semantics.

Definition 4 (*Flow-Level Security*). *Let F_l be a flow and v_l be an initial value for the inject node of the flow, i.e., $N_l = \langle\langle handler(x)\{c\}, M, v_l, O\rangle_l, wires, l\rangle_l$. We define flow executions $Exec(F_l, v_l)$ secure if*

$$N_l \in Nodes(F_l) \wedge \forall F_l'.\ F_l \xrightarrow{T_F}{}^{*} F_l' \Rightarrow secure(T_F).$$

The trace T_F is secure if $secure(T_F)$ holds, i.e., every event of the trace is secure according to the security policy of the corresponding node.

Definition 5 (*Global-Level Security*). *Let G be an environment and V_{init} be a set of initial values for the inject nodes of the flows in G, i.e., $\forall (N_j, v_j) \in V_{init}$. $F_j \in Flows(G) \wedge N_j \in Nodes(F_j) \wedge N_j = \langle\langle handler(x)\{c\}, M, v_j, O\rangle_j, wires, j\rangle_j$. We define global executions $Exec(G, V_{init})$ secure if*

$$\forall G'.\ G \xrightarrow{T_G}{}^{*} G' \Rightarrow secure(T_G).$$

Enforcement Mechanism. Figure 11 presents the core of our fine-grained monitor to enforce the above-mentioned security condition with respect to allowlist policies. We annotate evaluation relations with $\mathcal{M}$ to distinguish between the monitored behavior and the original one. We only present the rules that differ from the semantic rules given in Fig. 8; we replace $\rightarrow$ with $\rightarrow_{\mathcal{M}}$, and $\Downarrow$ with $\Downarrow_{\mathcal{M}}$. We add security constraints to the semantic rules for reading a variable from the shared context (rule $\textsc{Read}_{\mathcal{M}}$), calling an API function (rule $\textsc{Call}_{\mathcal{M}}$), and writing to a shared variable (rule $\textsc{Write}_{\mathcal{M}}$).

For the email example in Sect. 2, the policy requires allowlisting the API for sending the message and the list of intended recipients. The monitor intervenes whenever the API is called and ensures that the recipient is in the allowlist policy. An execution of a flow containing the malicious email node will be suppressed because the attacker's email address is not listed in the permitted values of the API call. The malicious event is detected by the rule $\textsc{Call}_{\mathcal{M}}$, i.e., `sendMail` $\in P_k \wedge$ `"me@attacker.com"` $\notin V_k($`sendMail`$)$.

For context vulnerabilities, such as Water Utility Complete Example, the allowlist consists of access rights to shared variables for each node deployed in the environment. The monitor observes the interaction of nodes with the shared context and blocks the execution whenever the allowlist policy does not permit access to the shared variable. The attack scenario in the vulnerable water utility flow can also be prevented by specifying an allowlist policy $(\texttt{tank1Level}, W)$ only for the nodes that must write to a shared variable, which stops any attempt from other nodes to write to the global context (rule $\textsc{Write}_{\mathcal{M}}$).

We prove the soundness and transparency properties of our monitor. The soundness theorem shows that any global traces produced by an execution of the monitor are secure with respect to the allowlist policy.

Theorem 1 (Soundness). *The monitor enforces global-level security for any finite executions,*

$$\forall (G, V).\, \forall G'.\, G \xrightarrow{T_G}{}^{*}_{\mathcal{M}}\, G' \Rightarrow secure(T_G).$$

The transparency theorem shows that if a monitored execution is secure, the monitor semantics and the original semantics generate the same trace. Moreover, if both semantics run under the same scheduler, the monitor preserves the longest secure prefix of a trace.

Theorem 2 (Transparency). *The monitor preserves the longest secure prefix of a trace yielded by an execution,*

$$\begin{aligned}\forall (G_0, V).\, \forall n \in \mathbb{N}.\, G_0 \xrightarrow{T}{}^{n} G_n \Rightarrow \exists m \leq n.\, G_0 \xrightarrow{T'}{}^{m}_{\mathcal{M}}\, G_m \;\wedge \\ \Big[\Big(secure(T) \Rightarrow T = T' \wedge n = m\Big) \vee \\ \Big(\big(\exists i < n.\, G_0 \xrightarrow{T_{pre}}{}^{i} G_i \wedge G_i \xrightarrow{T_i} G_{i+1} \wedge G_{i+1} \xrightarrow{T_{post}}{}^{n-i-1} G_n \wedge secure(T_{pre}) \wedge \\ \neg secure(T_i)\big) \Rightarrow T' = T_{pre} \wedge i = m\Big)\Big].\end{aligned}$$

The proofs of the theorems are reported in the online appendix [2].

4 Related Work

We discuss the most closely related work on Node-RED security and modeling, monitor implementation, and securing trigger-action platforms in general. We refer the reader to surveys on the security of IoT app platforms [7,13] for further details.

Node-RED Security and Modeling. Ancona et al. [5] investigate runtime monitoring of parametric trace expressions to check the correct usage of API functions in Node-RED. Trace expressions allow for rich policies, including temporal patterns over sequences of API calls. By contrast, our monitor supports both coarse and fine access control granularity of modules, functions, and contexts. Schreckling et al. [49] propose COMPOSE, a framework for fine-grained

static and dynamic enforcement that integrates JSFlow [21], an information-flow tracker for JavaScript. COMPOSE focuses on data-level granularity, whereas our monitoring framework supports module- and API-level granularity.

Clerissi et al. [15] use UML models to generate and test Node-RED flows to provide early system validation. A preliminary set of guidelines has also been proposed to assist Node-RED flow makers in terms of user comprehension and for testing activities [16]. Focusing more on end users and less on developers, Kleinfeld et al. [27] introduce an extension of Node-RED called glue.things, enabling Node-RED easier to use by predefined trigger and action nodes. Blackstock and Lea [12] propose a distributed runtime for Node-RED apps such that flows can be hosted on various platforms. Tata et al. [53] propose a formal modeling for decomposing process-aware applications deployed in IoT environments using Petri nets; Node-RED indeed fits in this setup, thus extended as a prototype for their approach [25].

In terms of modeling, Node-RED can be intrinsically seen as a concurrent system, thus our approach shares similarities with the broad range of formal approaches such as process calculi [8,46], CSP [22], and CCS [31]. In the same spirit, our formalization is targeted to capture the execution model of Node-RED flows consisting of concurrent node executions that trigger the execution of code upon receiving messages, and modify, create, and dispatch messages to the next nodes. In contrast, our modeling is explicit and it captures the essence of the execution semantics of Node-RED. Focusing on security policies in concurrent systems, KLAIM [11,32] is a programming language providing a mechanism to customize access control policies. The mechanism, based on a hierarchical capability-based type system, enforces policies that control resource usage and authorize migration and execution of processes. While KLAIM is designed for programming distributed applications with agents and code mobility, our Node-RED model is simple and expressive enough to describe the API-based access control enforcement mechanism.

Monitor Implementation. Regarding the possible candidates for implementing our theoretical framework, it should be noted that the dynamic nature of JavaScript requires more precise analysis provided by dynamic approaches. Andreasen et al. [6] survey available methods for dynamic analysis for JavaScript, and outline three general categories: runtime instrumentation, source code instrumentation, and metacircular interpreters.

DProf [19] and NodeProf [52] are two well-known runtime instrumentation tools. DProf instruments a program at the instruction level, targeting a variety of languages, including JavaScript. NodeProf instead instruments a program at the abstract syntax tree (AST) level and is specifically made as a dynamic analysis framework for Node.js. However, some important Node.js features, such as `module.exports`, commonly used in Node-RED nodes, are not supported by NodeProf yet. In addition, to obtain the desired results, it requires the instrumentation of the entire Node-RED environment. NodeMOP [48] is a Monitoring-Oriented Programming (MOP) tool built on top of NodeProf that also looks interesting for our purposes, while the challenges in practice remain unresolved.

Ferreira et al. [18] propose a lightweight permission system to enforce the least-privilege principle at the Node.js packages level at runtime, restricting access to security-critical APIs and resources. Sharing some of our motivations, however, this work does not enforce access control policies at the context and value levels. Pyronia [29] is a fine-grained access control system for IoT applications restricting access at the function level via runtime and kernel modifications. To detect access to sensitive resources, Pyronia leverages OS-level techniques such as system call interposition and stack inspection. By contrast, our monitor needs to be implemented in language-level isolation to prevent access to sensitive resources at different levels of granularity.

Membrane-based approaches [1,3,20,30,50] seem to be the most promising compared to other techniques. Membranes are a "defensive programming pattern used to intermediate between sub-components of an application" [54]. This pattern is implemented in Node.js by recursively wrapping an object in a proxy with respect to prototype hierarchies such that the wrapped object can only be modified in protected ways. Staicu et al. [51] provide an example of this technique applied to Node.js, isolating libraries to extract taint specifications automatically.

SandTrap [3] combines the Node.js`vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. SandTrap has been integrated with Node-RED and evaluated on a set of flows while enforcing a variety of policies yet incurring negligible runtime overhead. Our framework is a step toward providing the formal grounds for characterizing the soundness and transparency of the SandTrap instantiation to Node-RED. The formalization can be further enhanced by modeling the Node.js environment and full-featured JavaScript [28].

Securing Trigger-Action Platforms. IoTGuard [14] is a monitor for enforcing security policies written in the IoTGuard policy language. Security policies describe valid transitions in an IoT app execution. Bastys et al. [9,10] study attacks by malicious app makers in IFTTT and Zapier. They develop dynamic and static information flow control (IFC) in IoT apps and report on an empirical study to estimate to what extent IFTTT apps manipulate sensitive information of users. Wang et al. [56] develop NLP-based methods to infer information flows in trigger-action platforms and check cross-app interaction via model checking. Alpernas et al. [4] propose dynamic coarse-grained IFC for JavaScript in serverless platforms. Our presented monitor is based on access control rather than IFC. Hence, these works are complementary, focusing on information flow after access is granted. IFC supports rich dependency policies, yet arduous to track information flow in JavaScript without breaking soundness or giving up precision.

5 Conclusion

We have investigated the security of Node-RED, an open-source JavaScript-driven trigger-action platform. We have expanded on the recently-discovered

critical exploitable vulnerabilities in Node-RED, where the impact ranges from massive exfiltration of data from unsuspecting users to taking over the entire platform. Motivated by the need for a security mechanism for Node-RED, we have proposed an essential model for Node-RED, suitable to reason about nodes and flows, be they benign, vulnerable, or malicious. We have formalized a principled framework to enforce fine-grained API control for untrusted Node-RED applications. Our formalization for a core language shows how to soundly and transparently enforce global security properties of Node-RED applications by local access checks, supporting module-, API-, value-, and context-level policies.

Acknowledgments. This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

References

1. Agten, P., Van Acker, S., Brondsema, Y., Phung, P.H., Desmet, L., Piessens, F.: JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In: ACSAC (2012). https://doi.org/10.1145/2420950.2420952
2. Ahmadpanah, M.M., Balliu, M., Hedin, D., Olsson, L.E., Sabelfeld, A.: Securing Node-RED Applications. Proofs. https://www.cse.chalmers.se/research/group/security/SandTrap/proofs.pdf (2021)
3. Ahmadpanah, M.M., Hedin, D., Balliu, M., Olsson, L.E., Sabelfeld, A.: SandTrap: securing JavaScript-driven trigger-action platforms. In: USENIX Security (2021). https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadpanah
4. Alpernas, K., et al.: Secure serverless computing using dynamic information flow control. In: OOPSLA (2018). https://doi.org/10.1145/3276488
5. Ancona, D., Franceschini, L., Delzanno, G., Leotta, M., Ribaudo, M., Ricca, F.: Towards runtime monitoring of node.js and its application to the internet of things. In: ALP4IoT@iFM (2017). https://doi.org/10.4204/EPTCS.264.4
6. Andreasen, E., et al.: A survey of dynamic analysis and test generation for JavaScript. ACM Comput. Surv. (2017). https://doi.org/10.1145/3106739
7. Balliu, M., Bastys, I., Sabelfeld, A.: Securing IoT Apps. IEEE S&P Magazine (2019). https://doi.org/10.1109/MSEC.2019.2914190
8. Balliu, M., Merro, M., Pasqua, M., Shcherbakov, M.: Friendly fire: cross-app interactions in IoT platforms. ACM Trans. Priv. Secur. (2021). https://doi.org/10.1145/3444963
9. Bastys, I., Balliu, M., Sabelfeld, A.: If this then what? controlling flows in IoT apps. In: CCS (2018). https://doi.org/10.1145/3243734.3243841
10. Bastys, I., Piessens, F., Sabelfeld, A.: Tracking information flow via delayed output - addressing privacy in IoT and emailing apps. In: NordSec (2018). https://doi.org/10.1007/978-3-030-03638-6_2
11. Bettini, L., et al.: The klaim project: theory and practice. In: Global Computing (2003). https://doi.org/10.1007/978-3-540-40042-4_4
12. Blackstock, M., Lea, R.: Toward a distributed data flow platform for the web of things (distributed node-RED). In: WoT (2014). https://doi.org/10.1145/2684432.2684439
13. Celik, Z.B., Fernandes, E., Pauley, E., Tan, G., McDaniel, P.D.: Program analysis of commodity IoT applications for security and privacy: challenges and opportunities. ACM Comput. Surv. (2019). https://doi.org/10.1145/3333501

14. Celik, Z., Tan, G., McDaniel, P.: IoTGuard: dynamic enforcement of security and safety policy in commodity IoT. In: NDSS (2019). https://doi.org/10.14722/ndss.2019.23326
15. Clerissi, D., Leotta, M., Reggio, G., Ricca, F.: Towards an approach for developing and testing node-RED IoT systems. In: EnSEmble@ESEC/SIGSOFT FSE (2018). https://doi.org/10.1145/3281022.3281023
16. Clerissi, D., Leotta, M., Ricca, F.: A set of empirically validated development guidelines for improving node-RED flows comprehension. In: ENASE (2020). https://doi.org/10.5220/0009391101080119
17. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: S&P (2010). https://doi.org/10.1109/SP.2010.15
18. Ferreira, G., Jia, L., Sunshine, J., Kästner, C.: Containing malicious package updates in NPM with a lightweight permission system. In: ICSE (2021). https://doi.org/10.1109/ICSE43902.2021.00121
19. Gregg, B., Mauro, J.: DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD. Prentice Hall Professional (2011)
20. Groef, W.D., Massacci, F., Piessens, F.: NodeSentry: least-privilege library integration for server-side JavaScript. In: ACSAC (2014). https://doi.org/10.1145/2664243.2664276
21. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: tracking information flow in JavaScript and its APIs. In: SAC (2014). https://doi.org/10.1145/2554850.2554909
22. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM (1978). https://doi.org/10.1145/359576.359585
23. IBM Cloud (2021). https://cloud.ibm.com/
24. IFTTT: If This Then That (2021). https://ifttt.com
25. Jain, R., Klai, K., Tata, S.: Formal modeling and verification of scalable process-aware distributed iot applications. In: ISPA/BDCloud/SocialCom/SustainCom (2019). https://doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00047
26. jcreedcmu: Escaping NodeJS vm (2018). https://gist.github.com/jcreedcmu/4f6e6d4a649405a9c86bb076905696af
27. Kleinfeld, R., Steglich, S., Radziwonowicz, L., Doukas, C.: glue.things: a mashup platform for wiring the internet of things with the internet of services. In: WoT (2014). https://doi.org/10.1145/2684432.2684436
28. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for JavaScript. In: APLAS (2008). https://doi.org/10.1007/978-3-540-89330-1_22
29. Melara, M.S., Liu, D.H., Freedman, M.J.: Pyronia: intra-process access control for IoT applications. CoRR abs/1903.01950 (2019). http://arxiv.org/abs/1903.01950
30. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Johns Hopkins University (2006)
31. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
32. Nicola, R.D., Ferrari, G.L., Pugliese, R.: Programming access control: the KLAIM experience. In: CONCUR (2000). https://doi.org/10.1007/3-540-44618-4_5
33. Node-RED: Community Node Module Catalogue (2021). https://github.com/node-red/catalogue.nodered.org
34. Node-RED: Cyclic Flows (2021). https://groups.google.com/g/node-red/c/C6M3HokoSTI/m/B2tqcb_cAQAJ
35. Node-RED: Making Flows Asynchronous by Default (2021). https://nodered.org/blog/2019/08/16/going-async

36. Node-RED (2021). https://nodered.org/
37. Node-RED: Securing Node-RED (2021). https://nodered.org/docs/user-guide/runtime/securing-node-red
38. Node-RED: The Core Nodes (2021). https://nodered.org/docs/user-guide/nodes
39. Node-RED: The RED Object (2021). https://github.com/node-red/node-red/blob/master/packages/node_modules/node-red/lib/red.js
40. Node-RED: Working with Context (2021). https://nodered.org/docs/user-guide/context
41. Node-RED Library (2021). https://flows.nodered.org/
42. Node.JS: VM (executing JavaScript) (2021). https://nodejs.org/api/vm.html#vm_vm_executing_javascript
43. NPM: Node Package Manager (2021). https://www.npmjs.com/
44. OWASP: NodeJS Security Cheat Sheet (2021). https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html#do-not-use-dangerous-functions
45. Pfretzschner, B., ben Othmane, L.: Identification of Dependency-based Attacks on Node.js. In: ARES (2017). https://doi.org/10.1145/3098954.3120928
46. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR (1997)
47. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proc. IEEE (1975). https://doi.org/10.1109/PROC.1975.9939
48. Schiavio, F., Sun, H., Bonetta, D., Rosà, A., Binder, W.: NodeMOP: runtime verification for node.js applications. In: SAC (2019). https://doi.org/10.1145/3297280.3297456
49. Schreckling, D., Parra, J.D., Doukas, C., Posegga, J.: Data-centric security for the IoT. In: IoT 360 (2) (2015). https://doi.org/10.1007/978-3-319-47075-7_10
50. Simek, P.: Proposal for VM2: advanced vm/sandbox for Node.js (2021). https://github.com/patriksimek/vm2
51. Staicu, C., Torp, M.T., Schäfer, M., Møller, A., Pradel, M.: Extracting taint specifications for JavaScript libraries. In: ICSE (2020). https://doi.org/10.1145/3377811.3380390
52. Sun, H., Bonetta, D., Humer, C., Binder, W.: Efficient dynamic analysis for Node.js. In: CC (2018). https://doi.org/10.1145/3178372.3179527
53. Tata, S., Klai, K., Jain, R.: Formal model and method to decompose process-aware IoT applications. In: OTM (2017). https://doi.org/10.1007/978-3-319-69462-7_42
54. Van Cutsem, T.: Isolating Application Sub-components with Membranes (2018). https://tvcutsem.github.io/membranes
55. Ur, B., McManus, E., Ho, M.P.Y., Littman, M.L.: Practical trigger-action programming in the smart home. In: CHI (2014). https://doi.org/10.1145/2556288.2557420
56. Wang, Q., Datta, P., Yang, W., Liu, S., Bates, A., Gunter, C.A.: Charting the attack surface of trigger-action IoT platforms. In: CCS (2019). https://doi.org/10.1145/3319535.3345662
57. Zapier (2021). https://zapier.com
58. Zimmermann, M., Staicu, C., Tenny, C., Pradel, M.: Small world with high risks: a study of security threats in the NPM ecosystem. In: USENIX Security (2019). https://dl.acm.org/doi/10.5555/3361338.3361407