CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Securing IoT Apps in Node-RED

Master's thesis in Computer science and engineering

Lars Eric Olsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Securing IoT Apps in Node-RED

Lars Eric Olsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Securing IoT Apps in Node-RED
Lars Eric Olsson

Supervisor: Andrei Sabelfeld, Department of Computer Science and Engineering
Examiner: Gerardo Schneider, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Securing IoT Apps in Node-RED
Lars Eric Olsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Node-RED, an Internet of Things (IoT) platform, provides the opportunity for users to connect devices and services in novel and useful ways. This platform gives users a graphical web interface for easily linking pre-defined pieces of code (nodes) encoding devices and services. By being built in Node.js, third-party developers are given the opportunity of easily extending the functionality of the platform through publishing nodes and configurations of these nodes, otherwise known as flows. In this paper, we analyze Node-RED from a language-based security perspective, modeling the application developer as an attacker, and demonstrating attacks misusing sensitive APIs within nodes. API access control provides a security guarantee around the execution of these nodes. We collect and survey published nodes and flows to establish the presence of these security challenges within the Node-RED ecosystem.

Keywords: Computer security, Internet of Things, Node-RED, Node.js, JavaScript

Acknowledgements

I would like to thank my supervisor Andrei Sabelfeld for both suggesting Node-RED as an interesting platform to study, and providing continual encouragement and advice during the process of this thesis.

Mohammad Ahmadpanah has also been invaluable in both discussion and research into the specifics of Node-RED security, especially into the more theoretical modelling aspects. The assistance of JSFlow developers Daniel Hedin and Alexander Sjösten in implementing Node.js support within JSFlow was also much appreciated in evaluating that technology's use in this application. Finally, I would like to thank Gerardo Schneider, Cristian-Alexander Staicu, and Henry Ly for providing feedback on the writing and content of this thesis.

Lars Eric Olsson, Gothenburg, March 2020

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Internet of Things (IoT) applications have the potential to connect the world in new and interesting ways. The IoT has been described as "a new computing paradigm, in which a continuum of devices and objects are interconnected with a variety of communication solutions" [54]. These sorts of IoT applications provide utility through connecting previously disconnected Internet-enabled components. Components of these apps can range from the variety of "smart" cyber-physical "things" (such as home automation) to more pedestrian online services and social networks. However, connecting these components in an ad-hoc manual basis is infeasible for the layman without some common shared interface. Node-RED provides a visual browser-based editor to connect these components, while also providing the necessary configuration for individual components through an extensible "palette" of applets. The popularity of this solution can be extrapolated from that of other IoT platforms such as If This Then That (IFTTT), which have millions of users running billions of applets connecting hundreds of different components [4].

However, this utility comes with new security concerns. Privacy becomes threatened as applets access new sensitive information sources. Application domains of the IoT such as transportation, home automation, and healthcare are such sources of information [54]. In healthcare, two general classes of usage are monitoring and management; Pawar and Ghumbre enumerate the possibilities within these [40]. Glucose level, electrocardiogram (ECG), and blood pressure can all be monitored, while systems such as wheelchairs, medication, and rehabilitation can be managed. In just this handful of applications, IoT can improve central concerns to the healthcare industry such as quality of care.

Security is already an issue within healthcare; this industry has been subject to "Ransomware" attacks such as WannaCry, where hospital data is encrypted and held ransom until payment is made [9]. Scaife et al. (2016) estimate these extortion payments amount to tens of millions of dollars annually, while O'Gorman and McDonald (2012) give an upper bound of $400,000 USD for any one of these same ransom payments [39, 43]. This class of malware attack only affects the availability of information, while also being so valuable; how does the value of the privacy and integrity of this data compare? While we cannot concretely answer this question, we can say that the application of IoT to the domain of healthcare will also make

these facets of security relevant.

In addition, within these applications, integrity and availability both become even more immediate concerns when applets are given control of cyber-physical objects and the corresponding ability to manipulate the real-world. Within healthcare management applications, the modification or outright denial of actions taken by an IoT application has the very real consequence of injury or death. The Node-RED platform has the possibility of ensuring some of this security.

In contrast to centralized platforms such as IFTTT, Zapier, and Microsoft Flow, Node-RED instead can be entirely run on a user's own server. Whether due to outright malice or simply insecure coding practices, the possibility of attacks or vulnerabilities existing in an otherwise un-inspectable centralized platform's software and deployment makes trusting these parties responsible for centralized platforms with the integral security of both the information and actions required for IoT applications a steep cost. Running the Node-RED platform instead, on one's own hardware with inspectable open-source code, makes forming this otherwise blind trust of an external platform unnecessary. However, this only avoids the problem of security regarding the underlying platform itself - components integrating into it such as third-party applications can still be threats to the entire system's security.

1.2 Goals

This work focuses on ensuring the security of these third party components. Security within IoT applications is of vital importance when considering the consequences if this security is compromised. Privacy is not the only concern, or perhaps even the most important; integrity and availability are also of great importance when the ability of these IoT applications to manipulate the physical world is considered. Given the emphasized importance of security in this domain, others have also studied the security of both abstract and specific IoT platforms such as IFTTT - see Section 9.1 for an overview of related work in this area. However, Node-RED has not been subject to any such review, especially when considering language-based security in particular. Our main goals can be summarized by the following research questions.

1.2.1 Research Questions

1. What attacker models and associated attacks are present in Node-RED? Are there specific vulnerabilities in this platform as well?
2. How prevalent are these issues in the Node-RED ecosystem?
3. How can language-based security techniques be used to address these attacks and vulnerabilities?

1.3 Methodology

We answer these questions and resultant security issues through empirical analysis and study of appropriate mitigation mechanisms. Empirical studies lend additional weight to the attacks and vulnerabilities discovered in Node-RED, while also giving some figures for the prevalence of these issues in practice. Mitigation mechanisms, such as dynamic monitors, are then used to address these problems.

1.4 Contributions

While our research questions are all interdependent on each other, they can be separated by the methods we use to answer each of them, also logically separated in this work. Therefore, we present our contributions that correspond to the initial research questions in the following list, where entries in this list correspond to the matching question.

1. Developing an attacker model for answering the first research question requires the specific context of the Node-RED platform; Chapter 2 develops this context. Namely, Section 2.1 details the architecture of the platform, and section 2.2 explores the associated publishing model for applications. Most importantly, nodes are published through the Node Package Manager (NPM) and automatically added to the Node-RED catalog.

   Chapter 3 discusses possible attacker models. We detail the case of the attacker being an application developer, with the ability to publish both malicious nodes and flows. Finally, in Chapter 4 we use this attacker model to explore possible attacks and vulnerabilities. We develop an attack where the attacker is able to inappropriately use Node.js APIs within Node-RED to both collect and exfiltrate information, while also allowing further sensitive capabilities such as executing shell commands. We also highlight the "context" feature of Node-RED as an additional vulnerability in the platform.

2. In Chapters 5, 6, and 7, we conduct a series of empirical studies to illustrate the real-world importance of these attacks and vulnerabilities on Node-RED.

   Chapter 5 provides an overview of how our dataset is constructed, the composition of data, and how these properties relates to underlying issues in Node.js and NPM in general. This dataset consists of 2122 node packages and 1181 flows. We find that Node-RED node packages contain non-trivial amounts of code, with many relying on multiple external packages. Code complexity strengthens an attackers ability to obfuscate malicious code, while package dependencies introduce security issues found within the NPM ecosystem more broadly. "Squatting" attacks in particular make any Node-RED specific attack more feasible than already possible.

Chapter 6 looks into the vulnerability of a shared context feature in Node-RED, and how this can be exploited in published applications. We illustrate the dangers found in using this feature for either inter-node communication or in sharing resources. This vulnerability grants additional sensitive capabilities to attackers, without their needing to directly misuse Node.js APIs as done in our other demonstrated attack (Chapter 3).

Chapter 7 categorizes applications in Node-RED to form some statistics on the possibility of security violations within those applications. This experiment reconstructs a similar one performed on the IFTTT platform by Bastys et al. (2018) [4]. From a limited selection of the top 100 most popular packages, we find substantially increased privacy, decreased integrity, and slightly increased availability violations when compared to the statistics formed on the IFTTT platform. We attribute these differences in prevalence of possible attacks as symptoms of the generalized function of nodes, preventing more fine-grained security labels. This in turn stems from the underlying difference in focus of the compared platforms, with Node-RED being more aimed at developers and systems, while IFTTT has a more general, consumer-oriented outlook.

3. Finally, we attempt to mitigate the uncovered issues. Chapter 8 explores methods for addressing these attacks and vulnerabilities using language-based security techniques, such as various forms of dynamic monitoring. We apply API access control through a dynamic monitor, applying security policies specific to node instances within flows. Proxies are chosen as the best dynamic monitoring technique to apply this mitigation. We also outline program rewriting to avoid vulnerabilities when using the Node-RED shared context feature, with different usages and their corresponding securely rewritten replacements enumerated.

Chapters 9 and 10 provide additional context for the choices we have made while fulfilling our goals. Chapter 9 enumerates related work, while Chapter 10 concludes, with Section 10.1 discussing possibilities for future work.

2
Node-RED

Node-RED markets itself as "a programming tool for wiring together hardware devices, APIs and online services" [26]. It can be used similarly to other IoT platforms such as IFTTT to achieve customizable functions. However, the Node-RED platform differs in a few important ways from IFTTT and others.

2.1 Architecture

IFTTT, Zapier, and other commercial IoT platforms coordinate applications on their own centralized servers. In contrast, instead of being hosted on a central server, Node-RED seems targeted for deployment to an individual's own device. Instructions for setting up Raspberry Pi instances are specifically detailed [34]. Having multiple users on one shared server is also supported, if a larger organization wants to provide a centralized service. However, we concentrate on what we judge to be the more common use-case, where a user runs their own server.

Figure 2.1 shows a simplified view of the components of the Node-RED architecture. Simply put, applications are equivalent to "flows" in Node-RED, where these flows are composed of components called "nodes".

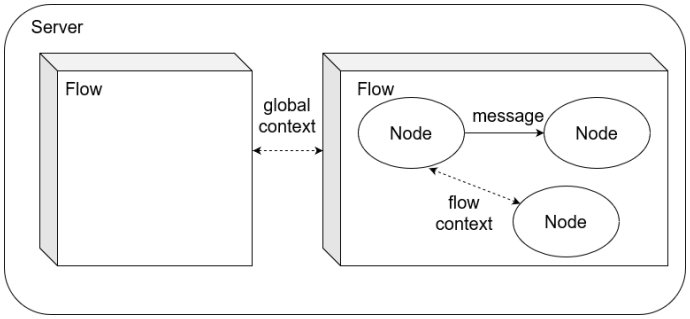

Figure 2.1: Node-RED architecture: A server can run multiple flows, which are composed of a graph of nodes. Besides messages sent explicitly as defined by the graph, information can also be communicated through the shared global and flow contexts.

2.1.1 Nodes

```
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);
        });
    }
    RED.nodes.registerType("type-name", NodeName);
}
```

Figure 2.2: Node JavaScript code structure

Nodes are segments of Node.js code that receive and send events; the configuration of this event handling defines up to one input port, and any number of output ports. Wires connect nodes through these ports, with events containing the messages being passed between. Messages take the form of a JS object. Multiple messages can be sent by any given node, though a single message can be repeatedly sent to multiple nodes as well. In addition to these messages received at runtime, nodes also receive information in the form of static configuration parameters defined in the flow. Figure 2.2 provides a general structure of a node's code.

We classify nodes into four classes (input, output, intermediary, configuration) based on the number of I/O ports. "Input" nodes are defined as nodes that only have output ports, lacking any for input. These wait on some external trigger to start the execution of the flow graph by passing a message on through their output port(s). "Output" nodes are defined as nodes that only have an input port, lacking any for output. These take a message, and perform some outside action with it, ending the execution of that branch of the flow. "Intermediary" nodes are defined as nodes with at an input and at least one output port. These form the majority of a flow's graph, as they are all connected nodes between input and output nodes. They generally transform the message, or perform some side-effect. Finally, "configuration" nodes are special nodes that lack any ports at all. Instead of being part of the graph of a node, these instead share configuration data, such as login credentials, between multiple nodes in the flow.

2.1.2 Flows

Node-RED servers run flows deployed to them, with these flows provided by the user, and visually shown through the web interface as the graph of nodes connected by wires. Pre-configured flows can be imported, either from the official catalog or

```
[ // List of nodes
    { // Node
        // 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], // 1st output to node 1
            [NODE2, NODE3] // 2nd to nodes 2, 3
        ],
        ... // other parameters unique to the node
        ... // or not of interest
    },
    ... // more nodes
]
```

Figure 2.3: Flow JSON structure

any other source [24]. Flows are stored, imported, and exported through JSON files, whose structure is shown in Figure 2.3 [20].
Figure 2.4 shows an example flow given in the Node-RED beginner documentation [23].

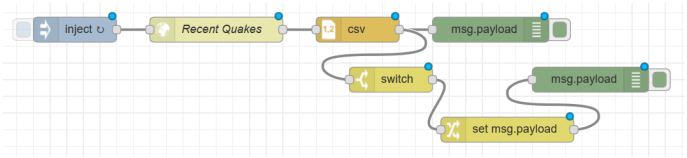

Figure 2.4: Example Flow: This flow regularly retrieves earthquake data, logs this result, then conditionally displays an additional message depending on an event's magnitude [23].

Flows are deployed through a simple button in the web interface. Once a flow is deployed, the Node-RED runtime will use event-handling code from input nodes to check for triggering conditions continually. When a node's registered callback is executed, this input node will eventually pass a JavaScript message through its output port(s). These messages are received by subsequent nodes in the flow, leading to the execution of some portion of the graph. After passing through some number of intermediary nodes, execution will terminate at a number of output nodes.

2.2 Publishing

With nodes and flows both providing opportunity for configuring a specific IoT application, there are two routes for publishing code that can wholly or in part define these applications: node and flow definitions.

2.2.1 Node definitions

Nodes are defined by Node.js packages. The "palette" of available Node-RED nodes is defined by an official list [22]. As of writing, this catalog contains 2122 published nodes. This official list is generated by automatic scraping of all Node Package Manager (NPM) packages, looking for the manifest of a package to satisfy a small number of rules [33]. Generally, a Node.js package for Node-RED will define a set of related nodes, instead of only one. Figure 2.5 shows a set of nodes related to the file-hosting service Dropbox, contained in the package node-red-node-dropbox; this package defines a dropbox configuration, a dropbox in, a dropbox out, and a more general dropbox node [30]. This is a suitable example of a typical Node-RED package; we find that 2.51 nodes are defined on average per package.

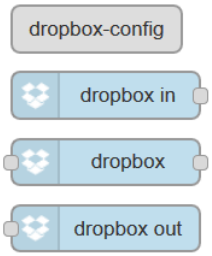

Figure 2.5: Nodes defined in node-red-node-dropbox [30]

As the publishing model of Node-RED nodes is Node.js packages, issues known within that technology also apply. Particularly, the problems of package dependency based attacks such as trust propagation occur [41]. Such attacks can leak global variables, while also manipulating both local and global variables. Encouragingly, we have found that an average package has only 1.85 dependencies in Node-RED, which is considerably below the average 5.8 dependencies that packages in Node.js have across the entire ecosystem [15]. However, there exist outliers, such as a Node-RED package with 68 direct dependencies, where the problems stemming from a large dependency tree are still evident; a comparison is shown in Table 2.1.

Table 2.1: Direct dependencies in Node.js and Node-RED packages

	Min.	Mean	Max.
Node.js Dependencies	0	5.80	41,550
Node-RED Dependencies	0	1.85	68

While the issues with Node.js packages are relevant, we suggest that Node-RED also has unique aspects that make a security analysis of this platform specifically interesting. We find a shared context functionality particularly intriguing. Besides messages passed via links explicitly defined and graphically shown through the user interface, there is a notion of shared context. This data store is defined at the node, flow, and global levels. Context at the node level is restricted to the particular node in which JS code is running under; other levels are defined similarly, with global

having no limitations in what nodes have access to this store [38]. This leads to the wiring of the nodes (shown in the user interface) not completely reflecting the information flow possible in the application. This disconnect between what the user is readily aware of and what actually occurs is exploitable.

2.2.2 Flow definitions

Flows are published via an official catalog [24]. At the time of writing, there are 1181 valid flows (parsable, non-empty, non-duplicate) in this library. One can publish to this collection by creating an account on Node-RED's website, and submitting an entry. These entries consist of the JSON defining the flow configuration, and HTML description, and some associated metadata such as tags. There does not seem to be any substantial validation run on these JSON configurations - we find empty, invalid, and duplicate entries, reducing the 1453 entries present to only 1181 importable flows.

3
Attacker Model

With the outline of the architecture and publishing model of the Node-RED platform developed in the previous chapter, we can now develop models of attackers within this system.

3.1 Possible Attacker Models

IoT architecture generically consists of three layers: sensing, transport, and application [54].

The sensing layer is responsible for collecting the information used in any IoT deployment. Attacks at this layer are critical; the application will be rendered nonfunctional without this base level. However, we judge these attacks as not as interesting from a language-based security perspective; they do not consider concepts such as the entire application's data flow, or even how individual nodes within that flow behave.

Transport layer attackers, such as those on the network, are of concern; Pawar and Ghumbre (2016) apply cryptographic algorithms to protect sensitive data used in medical IoT applications [40]. However, the specifics of the Node-RED platform do not significantly alter the opportunities for network attack from those considered in that survey; data still passes from sensors to the IoT machine, and from there on to external servers that provide services used. With this lack of novelty in mind, we also do not consider transport layer attackers.

Let us then focus on the application layer. At this layer of the Node-RED ecosystem, we identify three actors: the service provider (platform), the application maker (developer), and the user. These actors can be characterized by their assorted capabilities.

- The service provider, or platform, publishes code defining the Node-RED server and interface. The platform also manages the publishing system; this system specifies which NPM packages are included as valid Node-RED nodes, and what flows are listed in the official library. In addition, the platform also has the responsibility for documentation and other guidance for both developers and users, such as APIs available to nodes, and general user guides and

tutorials.

- The application maker, or developer, publishes code that the user can then run in their local instance of the Node-RED server. Developers can publish packages defining nodes to the NPM, which can be installed by users when they use the relevant nodes in the flows they run. Developers can also publish entire flow configurations, which a user can import and run directly.

- The user imports, edits, or creates from scratch flows that are run on their own server running an instance of Node-RED.

In all cases, we are concerned with the user being attacked. Some scenarios can be immediately eliminated; for example, the user attacking themselves is not a very realistic scenario, and not considered. The service provider attacking the user is also of less concern in the context of Node-RED, where users run their own instances of the platform software. The typical relationship of the user relying on the service provider to provide platform instances on outside servers is not applicable. However, the service provider could still conceivably provide malicious code for users running instances of the platform software on their own servers. This scenario can also be mitigated; a technical user can audit the platform's code themselves, and a non-technical user can rely on these audits as well.

We concentrate on analyzing the remaining scenario, where the attacker assumes the role of a malicious application maker. This actor has the capability of publishing applications in the form of nodes, flows, or conjunctions of the two. Through these angles of attack, malicious app makers can impact privacy, integrity, and availability. A general breakdown of the ways in which this attack can proceed is given in Figure 3.1, and the possibilities for attack shown in that diagram are further illustrated using modified versions of the Node-RED architecture diagram (Figure 2.1) in Figures 3.2, 3.3, and 3.4.

3.2 Privacy

Privacy is affected when there are nodes or flows that do not follow the user's policy on their information, however that policy is defined. Nodes and flows can be considered as separate attack vectors, though there can be some interplay between the two. As our definition of privacy depends on the definition of a policy, that must be developed first.

Let us first consider nodes. The Principle of Least Privilege, as defined by Saltzer and Schroeder, states that "every program and every user of the system should operate using the least set of privileges necessary to complete the job." [42] This minimal trust is a good starting policy - a node should not perform sensitive actions, beyond what is the minimum necessary for functionality. Automatically extracting this set of minimal actions is infeasible, but one can arrive at a first approximation by us-

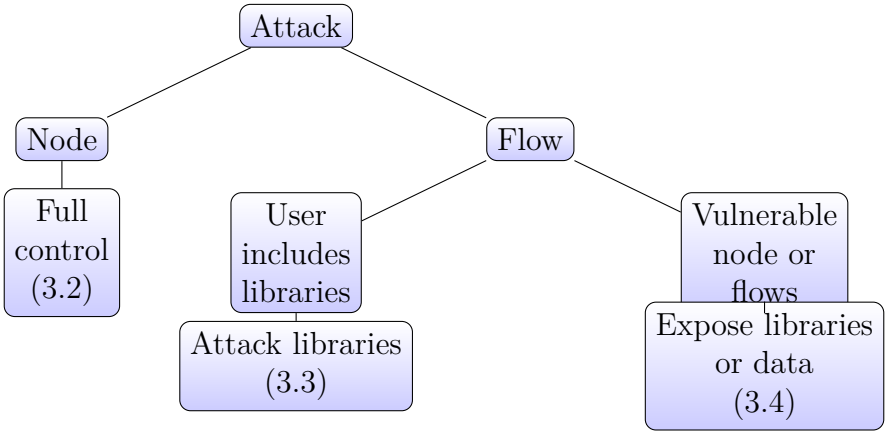

Figure 3.1: This tree diagram gives a rough overview of some of the possibilities for attack in Node-RED. Non-leaf nodes correspond to the choices in how the attack is made, while leaf nodes, denoting a particular attack, have a corresponding figure which shows where the attacker inserts themself.

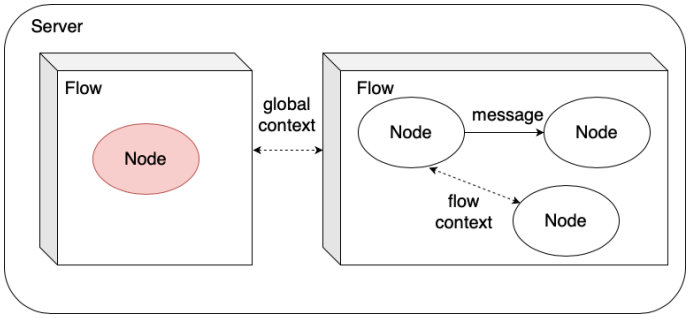

Figure 3.2: Full control attack: In this attack, the attacker publishes a malicious node. The user just needs to include this node in any flow, not necessarily any particularly sensitive one. With the ability for this node to include powerful packages, such as those that allow injection attacks, the attacker can assume full control of the user's system.

ing common-sense applied to characterizations such as the application's description. Nodes are defined through arbitrary JS code, providing functionality such as internet and filesystem access. Through these means, a node can communicate the message or other sensitive information gathered at runtime. The requirements for a malicious node to obtain and exfiltrate this information varies, though an important consideration is that various imported Node.js libraries allow the node more or less full system control, such as running arbitrary shell commands with "exec" through the "child_process" package. The exploitability of this particular library is not unique to Node-RED; among others, Staicu et al. (2018) consider this when they study the injection mechanisms available in Node.js [48]. However, a node (node-red-contrib-func-exec) already exists providing this function, though in a sandbox (a detail sure to be neglected by an attacker) [28]. While preventing these critical angles of attack is vital, we also consider the more subtle ways in which information can be obtained by the attacker, in this case a node.

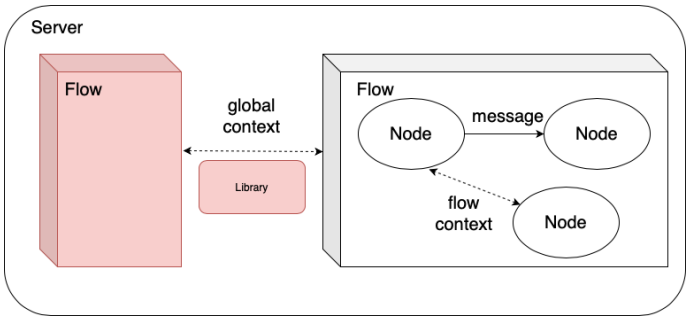

Figure 3.3: Attacking libraries: In this attack, the attacker instead controls a flow that the user runs alongside another vulnerable flow. This vulnerable flow uses a library stored in the shared context, which the attacker can then modify so as to observe and manipulate sensitive information.

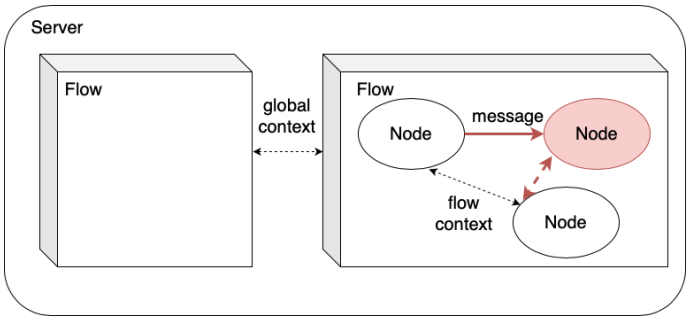

Figure 3.4: Exposing libraries or data: In this attack, the attacker manipulates the user into using a malicious node within a sensitive flow. Using either the messages passed directly to this node, or the shared context otherwise available within this flow, the attacker is able to manipulate libraries and expose sensitive information.

A first case is when the information is contained in a message. In this scenario, an attack requires that the node be wired in a particular flow configuration. We consider this to not be a very fruitful method for a possible attacker to obtain sensitive information, as it does not violate a policy we develop for flows as a whole later in this section when evaluating possibilities for attack in flows. A second case is when the information is not necessarily contained in the wired messages. Instead, it could be in the global context, or on another shared resource like the filesystem. Obtaining access here only requires that the node be run, not that it be integrated into a particular flow.

Both of these scenarios can be achieved in a number of different ways. A legitimate package can have their repository or publishing system compromised, and malicious code inserted. A package could be defined with a name similar to others, tricking users into installing a malicious version of an otherwise useful and secure package. This "name squatting" attack is especially easy in Node-RED, as the "type" of nodes (what flows use to specify them) is simply a string, which multiple packages can match. Finally, a pre-defined flow can include the attacker's malicious node; the user would then have to manually inspect each and every node to verify that there

are no deviations from the expected "type" string, further increasing the ease with which an attacker's package can be substituted into a previously secure flow.

Let us now consider flows. Defining a policy of what the user wants to be allowed for the application is also not immediately clear, but an approximation can be obtained from the graph of node connections. Flows are presented to the user through the GUI as a graphical model defined by the nodes and their wiring. As such, the ability to communicate through messages on wires between nodes can be easily inspected, and it is reasonable to assume these information flows are desired by the user. Communication between nodes that are not linked on the graph then goes against this policy.

Flows also define parameters that nodes can take. While also presented in the web interface, this information can be daunting to process for an end-user - importable flows often have very long arguments, such as the JS code given as a parameter to the Function node (capable of running JS code on the message). However, some aspects of these parameters should be relevant to policy - specifying what files a node should access, or what address an email should be sent to, should alter the access that a node has to the filesystem, or which addresses information can be sent to.

3.3 Integrity, and Availability

Similar methods also affect integrity and availability. While the main focus of this paper has been on analyzing the privacy implications, we argue that IoT applications have a greater focus on the security requirements of integrity and availability than many other areas. In addition to the security attributes of the data used by IoT apps, these programs must also preserve the same security around the physical actions that might be taken. Chapter 6 shows the vulnerability of published Node-RED flows that physically manipulate systems, including the "Water Utility Complete Example" and "Sprinkler Control Example" [35, 37].

In the previous section, we outline the ability of a node to both obtain and exfiltrate data surreptitiously through the misuse of sensitive Node.JS APIs. These same APIs also directly impact integrity and availability. Integrity of data is affected when shared resources which either store or generate this information is manipulated by the attacker. Availability of this data is affected when system resources are occupied or exhausted. The impact the attacker has on the data also directly affects that of actions reliant on the same data. All of these capabilities are available through the ability to run arbitrary shell commands through "child_process", while the attacker is granted limited but still sufficient powers with more plausibly useful libraries involving other shared resources such as the filesystem.

Obtaining access to these sensitive APIs is not even necessary when the attacker only wants to affect availability. If an attacker is able to get a malicious node into a flow's configuration, perhaps through the earlier name-squatting attack or through com-

promising an otherwise legitimate repository, this node can simply throw an error to interrupt the runtime of the flow. In addition, if the attacker is able to generate an "UncaughtException", not only the flow's execution but the entire Node-RED runtime will be unable to resume after this exception is thrown, and halt [19]. One method for generating these UncaughtExceptions is by registering an asynchronous task that will hit an error - this can be done with code as short and innocuous as trying to access a non-existent file.

4

Attacks and Vulnerabilities

4.1 Attacks

When we look at the malicious behavior of nodes, using the policies laid out in the previous section, attacks amount to nodes using APIs they are not required to for functionality. We distinguish attacks by the level of the application these occur on, either node or flow.

4.1.1 Node

At the node level, privacy attacks take the form of exfiltration attacks. For example, a node can be modified to send sensitive information from its received message on to an attacker's servers. The following code outlines the additional code necessary to modify any given node to perform the function of exfiltrating the data passed to it. This exfiltration is done through a simple HTTPS request, which can be added without disturbing original application code (the location of which is noted in comments).

```
node.on('input', function(msg) {
    const https = require('https')
    const data = JSON.stringify({
      message: msg.payload
    })
    const options = {
      hostname: 'attacker.com',
      port: 443,
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'Content-Length': data.length
      }
    }

    const req = https.request(options, function(res) {
      res.on('data', function(d) {
        process.stdout.write(d)
      })
    })

    req.on('error', function(error) {
      console.error(error)
```

```
    })

    req.write(data)
    req.end()
    // actual (non-exfiltrating) node code goes here
    node.send(msg);
});
```

This attack demonstrates the need for an access control policy - nodes should only be allowed to access what APIs are necessary, and no more. In this case, the node whose general skeleton is outlined here should not be allowed access to the 'https' package.

In a slightly more subtle example, the official Node-RED email node can be modified to always forward a second copy of whatever it is configured to send to an attacker's address in addition to the original recipient [31]. The original code sets the sending options to have the "to" field only contain the address of the intended recipient.

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

This can be modified to

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

In this example, we demonstrate that a simple whitelist/blacklist based solely on the identity of a function is not possible - APIs can be necessary for the functionality of a node, but still be used maliciously. Therefore, an access control policy needs to take into account both function identity (perhaps package and name), as well as arguments.

Integrity can be attacked by modifying the shared environment - resources available to the computer and program. Availability can be also influenced by disrupting this same shared environment.

4.1.2 Flow

At the flow level, attacks function similarly. However, the nodes a malicious flow is composed of are not necessarily malicious themselves. Instead, the specification of how the node functions - wiring and parameters - can modify the behavior of otherwise innocuous nodes to become an attack.

This is possible in Node-RED due to the generality of nodes. For example, there is a "Function" node present, which runs a sandboxed JS function with the message as input, as one might expect. However, this Function node also has access to the flow and global contexts, which can contain means to communicate with unconnected nodes or with the outside world more generally.

4.2 Vulnerabilities

The running of arbitrary JS, whether or not this JS is in a node or given as the parameters to a node in a flow, extends a broad threat surface. However, a notable feature of Node-RED is just this easy extensibility through user-defined nodes in Node.js. With this trait, the running of arbitrary JS code in these Node.js packages can be seen as a feature, and not a bug, of the Node-RED platform. Given this view, we also focus on context as a specific area of interest. In Section 6 we conduct an empirical study on vulnerabilities in the use of context in Node-RED.

Considering these attacks, we now proceed to measure the likelihood of a malicious adversary being able to deploy them in the Node-RED ecosystem.

5
Empirical Study - Data Collection

Investigating the prevalence of the attacks and vulnerabilities we develop in the previous chapters requires empirical data. In this chapter, an outline is given of how we collected available nodes and flows into a dataset for analysis. In addition, characteristics of the collected node packages lead to attacks within the broader NPM ecosystem also being applicable here in Node-RED.

5.1 Package Complexity

We use the list of Node-RED packages to obtain source code through the NPM catalog. We also scrape the official Node-RED catalog for flows. This catalog also provides some metadata such as download statistics for both flows and nodes, though these are unusable (entirely zero) for flows.
As nodes are mostly generic Node.js packages, individual nodes can differ greatly in how they function. Therefore, we gather basic statistics about node packages to give hints about what areas might be of particular interest when regarding security. This data was gathered over the entire listed collection as of the time of writing - 2122 packages. The following statistics come from this analysis.

Packages contain an average of 4.16 JS files, with official packages containing much less with a mean of 1.76. These files sum to 793.45 Lines of Code (LoC) on average, with official packages following the earlier trend with a lesser mean of 506.77 LoC. Given the relative complexity present here, it seems reasonable for us to conclude that attacks in nodes can be obfuscated within an extensive codebase, especially when considering that there exist listed nodes with 329 JS files containing a sum of 129,231 lines of code at the maximum end of complexity. Figures 5.1 and 5.2 provide additional summary statistics and trends around these measures of complexity.

5.2 Node.js Packages

Another relevant data point we gather is how many libraries Node-RED node packages use - 1.85 Node.js packages are direct dependencies of a Node-RED package on average. Figure 5.3 shows the basic trends in dependencies in Node-RED nodes. This is lower than the slightly more than two direct dependencies a Node.js package has on average [55]. We have not gathered data on indirect dependencies, but the relative similarity of the direct dependencies Node-RED packages have to an average

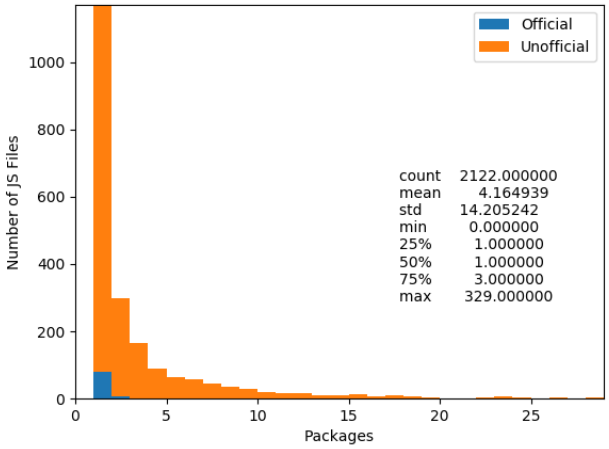

Figure 5.1: JS files per Node-RED package: An average of 4.16 JS files are contained in a Node-RED package. The distribution given by only considering official packages can be considered "good practice" - encouragingly, the distribution of third-party packages resembles this. However, the larger number of these unofficial packages makes the long tail of the data relevant; extreme examples are present with even hundreds of JS files.

Node.js package indicates that the same problems introduced in the broader NPM ecosystem are also relevant here. This can include attacks that utilize specific JS / Node.js features for undesirable ends such as "leakage of global variables, manipulation of global variables, manipulation of local variables, and manipulation of the dependency tree" [41]. Otherwise, issues of maintenance of multiple moving parts can take its toll, as security bugs in dependencies can be left unfixed, or not updated to a fixed version [11].

Which libraries are specifically used in Node-RED packages is also interesting, to get an idea of what sensitive APIs are the most useful to consider. Usage of third-party libraries can be identified by the dependencies of packages again, in Figure 5.4. First-party Node.js utilities are also considered by examining what is imported via 'require' statements in code - see Figure 5.5.
From these most prevalent dependencies, we see that Node-RED nodes do not typically use APIs relating to a specific service. We only start seeing specific services in the 23rd and 25th most popular entries, respectively aws-sdk and node-red. Otherwise, the most used package dependencies relate more generally to resources such as HTTP requests (request), serial data (serialport) and other developer tools. This illustrates the difference between Node-RED and a platform such as IFTTT; Node-RED is more focused on low-level customizable automation, while IFTTT is conversely more focused on consumer-oriented services. This can be further seen in Section 7, where we conduct an empirical study to generate statistics about the possibility of security violations. When we look at what code is imported via 'require' statements, we are told a similar story, with no specific service APIs being imported within the top 25 required libraries. Instead, these libraries deal with the filesystem

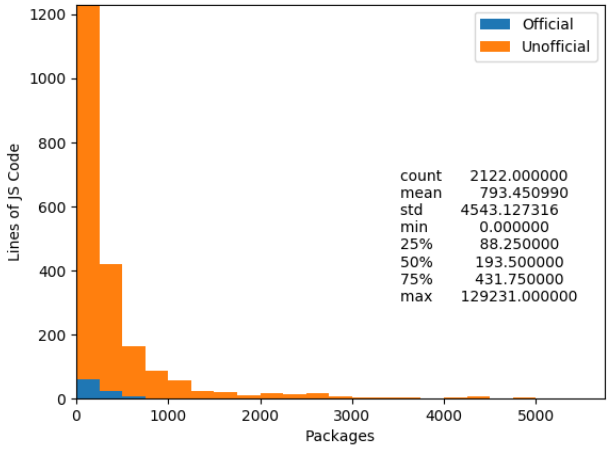

Figure 5.2: JS lines of code (LoC) per Node-RED package: The distribution of LoC within unofficial packages when compared to official packages is also encouragingly similar. However, published applications with tens of thousands of lines of code increase the possibility of obfuscating or otherwise concealing malicious code within.

(fs), HTTP requests (request), OS features (os), and other similar underpinning technologies.

5.2.1 Squatting Attacks

With the summary we have developed of a typical Node-RED package, we can now start considering how these characteristics affect security. One category of attack possible in Node.js is the various "squatting" attacks, such as typo-squatting. Vaidya et al. (2019) characterize typo-squatting as publishing malicious packages with 'names similar to existing, benign package' [53]. Users of Node-RED are presented with a GUI dialog for extending their "palette" of available nodes with more packages.

Node-RED nodes are registered to particular Node.js packages by a string "type", as in Figure 2.3. The GUI for installing packages only searches over package name, not type. Types registered in a package are listed in the package.json manifest, but are also defined in code:

```
RED.nodes.registerType("type", function);
```

This additional layer of names that can be impersonated makes typo-squatting attacks even more relevant to Node-RED, as a user seemingly has to search outside of the provided GUI to find out which package corresponds with the missing node type in a flow. In addition, a variant of the typo-squatting attack called "import-squatting" becomes important. In Python, these are found when the "name of the package differs from the top-level module name provided by the package" [53]. If the equivalency between what is defined in the package.json and code is similarly unenforced in the system, Node-RED is also vulnerable to this variant of the attack.

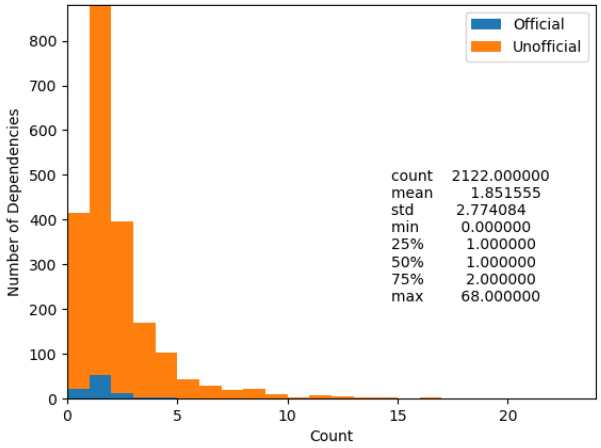

Figure 5.3: Node.js dependencies per Node-RED package: Dependencies in NPM introduce myriad problems; while Node-RED packages generally have less direct dependencies on average, this difference is minor, and not helped by outliers including dozens of Node.js packages.

We propose a few ways in which these squatting attacks can be mitigated. The search functionality presented in the GUI for installing packages can be extended to also index over types. The types registered per package can have their contents based on the package.json and code enforced, and an equivalency achieved. Types can also be guaranteed to be unique, perhaps by reserving the types as unique strings after packages are first indexed and collected into the Node-RED catalog.

5.3 Threats to Validity

While we have taken effort to collect all officially published flows, the scope of this data is small when compared to the number of published nodes. Given that there are 2122 unique published node packages, with 5316 nodes defined in these packages, one would expect more than the 1453 published flows (of which only 1181 are valid).

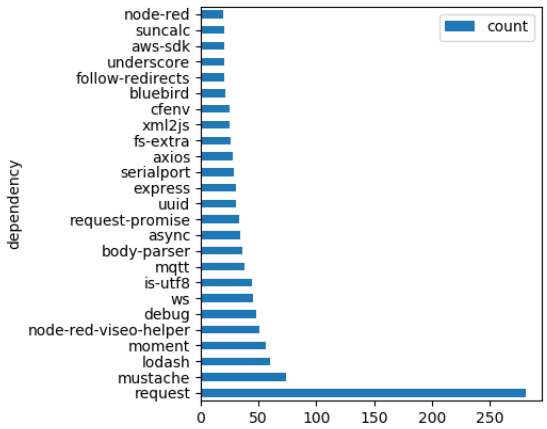

Figure 5.4: Top 25 Node.js dependencies in Node-RED packages: Node-RED package dependencies are usually general, as all but two of these packages provide a way to interface with resources or other developer tools.

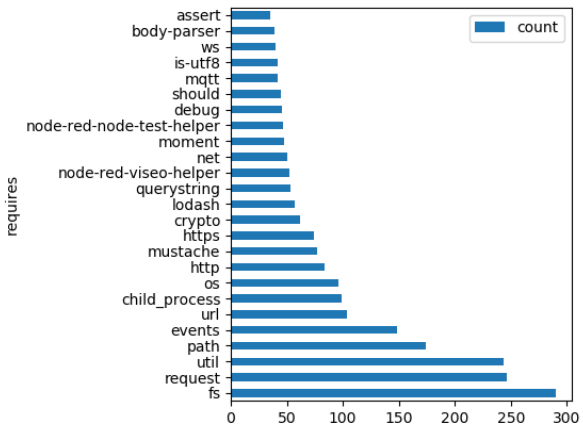

Figure 5.5: Top 25 Node.js requires in Node-RED packages: Requires within Node-RED enforce the same trend as with dependencies; these inclusions are low-level and developer focused.

6

Empirical Study - Context Vulnerabilities

Besides the attacks misusing APIs highlighted in Chapter 4, we briefly mention a vulnerability involving the context feature in Node-RED. In this chapter, we explore the details of this feature, as well as the vulnerabilities resulting from its use.

6.1 Context Usage

Context objects are obtained through (for node, flow, and global scopes):

```
this.context
this.flow
this.global
```

Here the current "this" is a Node-RED node's definition or inside the Function node. Values within context can be accessed with (where "c" is the context object):

```
c.get(name, value)
c.set(name, value)
c.keys()
```

We judge context usage at the node scope as innocuous, and do not consider it further - it only allows a node to share data with itself, and is generally used for preserving data between multiple executions of the flow, which does not violate the security policies described in Chapters 3 and 4. Instead, we detect pertinent context usage (at the scopes of flow and global, excluding node) in a node definition with regexes "flow.(get|set)" and "global.(get|set)" for flow and global scopes. This static analysis misses usages where layers of indirection are added - for example using:

```
var a = this.flow;
a.set("name", 1);
```

However, extending the analysis to include specific forms of indirection will never be comprehensive when performed statically.

Some built-in nodes also have built in the possibility of using context. The main offender is the Function node, but Inject (starting a flow), Template (generating text with a template), Switch (route outgoing messages), and Change (modify message properties) nodes also provide this functionality. We define particular parametrizations of these first-party nodes which we search for in flow definitions, where our

search yields only instances guaranteed to use the shared context. We do not give third-party nodes this level of specificity in our analysis - it is infeasible for us to collect the relevant parametrization rules that result in context being used for these many thousands of nodes. Instead, we interpret any instance of context use within a given node's code as a Boolean variable signifying that this node uses the shared context. Using these first-party node rules for detecting context along with the constructed set of third-party nodes that use context, we generate a value for the total number of occurrences for both within any particular flow, at the flow and global scopes separately.

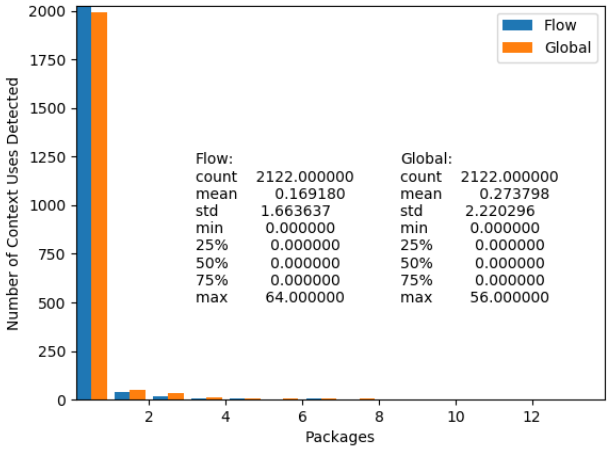

Figure 6.1: Shared context usage in Node-RED packages: Usage of the shared context, both at the flow and global levels, is not very prevalent in Node-RED packages.

6.2 Context Vulnerabilities

As can be seen in Figures 6.1 and 6.2, most nodes (defined in these packages) and most flows do not use shared contexts. However, there are those that use the shared context heavily, and even this small minority can have instances of security flaws. Manual analysis of the top 25 results for nodes and flows in terms of context use yields examples of such flaws, at both the scopes of flow and global. We have found these vulnerabilities within a small portion of the available nodes and flows; the prevalence of these issues within the dataset as a whole to a larger degree can be extrapolated.

6.2.1 Inter-node Communication

One common usage of context is for communication between nodes. This presents issues with both integrity and availability, as the shared data can be modified or

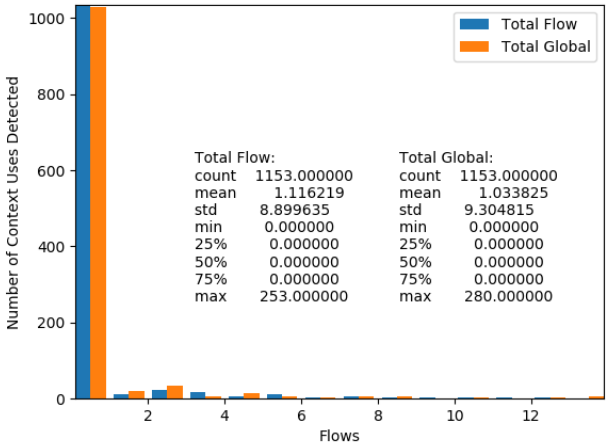

Figure 6.2: Shared context usage in flows: Similarly, the majority of sampled flows do not utilize the shared flow or global contexts. However, a slightly larger portion of flows use this capability when compared to nodes.

erased, changing or entirely disrupting functionality.

An example of this in a real-world flow is "Water Utility Complete Example" [37]. This flow manages two tanks and two pumps. Water is pumped from a well into the first tank, and can be pumped between the tanks.

Data managing tank state is first read from the physical tanks, and then stored in the global context.

```
global.set("tank1Level",tank1Level);
global.set("tank2Level",tank2Level);
global.set("tank1Start",tank1Start);
global.set("tank1Stop",tank1Stop);
global.set("tank2Start",tank2Start);
global.set("tank2Stop",tank2Stop);
```

Later, it is retrieved from the global context to adjust when a pump should start or stop.

```
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)
{
     msg.payload = { value: true,
     'fc': 5,
     'unitid': 1,
     'address': 2,
     'quantity': 1 }
     return msg
}
```

```
else if (pumpMode === true && pumpStatus === true && tankLevel >=
    tankStop)
{
     msg.payload = { value: false,
     'fc': 5,
     'unitid': 1,
     'address': 2,
     'quantity': 1 }
     return msg
}
```

In this example, an attacker could set the global variables relating to the tank's reading to either exhaust water flow (always stop), or attempt to cause physical damage through continuous pumping (never stop).

Another example of a possible physical disruption is a flow that controls sprinklers with program logic dependent on variables in the global context [35].
Depending on the sensitivity of the information provided in the flow, a malicious flow or node having access to the shared context can also lead to privacy issues such as exfiltration of sensitive data.

6.2.2 Shared Resources

Another usage we have noted is putting resources such as libraries into the shared context. In addition to concerns with integrity and availability, this enables a new method of extracting private data. An attacker can encapsulate the library such that it collects sensitive information sent to the library.

An example of this at the flow level is "btsimonh's node-opencv motion detection (2017-11-02)" [21]. This flow is meant to be deployed on a Raspberry Pi, using a video stream for motion detection. As such, image frames are fed into a computer vision library, opencv. This library is imported in the code snippet below.

```
var require = global.get('require');
if (!require){
     node.error("require not found in globals - see https://btsimonh
        .wordpress.com/node-opencv-with-node-red/ for installation
        notes")
     return;
}

// look for globally installed opencv
var cv = require.main.require('opencv');
if (!cv){
     // look for locally installed opencv
     cv = require('opencv');
}
if (!cv){
     node.error("opencv not found - see https://btsimonh.wordpress.
        com/node-opencv-with-node-red/ for installation notes")
     return;
```

```
}

var cvdesc = Object.keys(cv);
node.send([null, {payload:cvdesc}]);
flow.set('cv', cv);
```

This has two instances of disruptable libraries - require and opencv. Other examples are common [25, 36].

Nodes also display similar vulnerabilities. An instance of this is found in "node-red-contrib-openjtalk" [29]. This node provides a Japanese text to speech system. Again, a library is exposed in the node's code.

```
sandbox.global.set("openjtalk", require('openjtalk'));
```

While context is not used in most nodes or flows in Node-RED, our sampling of both nodes and flows that do use this functionality show that their usage still does offer possibilities for a motivated attacker to affect the privacy, integrity, and availability of these applications.

6.3 Threats to Validity

While the monitor we have defined is dynamic, the empirical studies we conduct are mostly static in nature. This is due to the difficulty of configuring the environment with necessary software, hardware, and input/output to get results from running the gathered nodes and flows.

7

Empirical Study - Security Classification

In previous chapters, we have studied the attacks and vulnerabilities possible in the Node-RED platform, only concerning this programming tool in relative isolation. However, Node-RED is only one of many ways to develop IoT applications. In this chapter, we give additional context in the comparison this platform to another, If This Then That (IFTTT).

7.1 Methodology

This empirical study is inspired by the analysis of IFTTT conducted by Bastys et al. (2018), wherein the authors undertake a study classifying information sources (triggers) and sinks (actions) by designating appropriate security levels [4]. A short summary of the classification as performed in that work, and how we have modified it to fit the Node-RED platform, follows.

We label triggers as either private, public, or available. The "public" label represents public information, while the "private" label likewise represents private information. "Available" labelled triggers generate public information, but whose presence (availability) is also valuable to the user. We label actions as either public, untrusted, or available. "Available" actions are those that their absence would be missed. "Untrusted" actions are those that can affect integrity. "Public" actions can communicate with the public, and can affect privacy. Classification of actions is cumulative - "public" actions are "untrusted" and "available", and "untrusted" actions are also "available". This cumulative relation is not present for labels of triggers.

In addition, we also label triggers and actions with categories, with these categories present in the original IFTTT dataset. These categories amount to broad descriptions of use such as "Pet Trackers". However, these category groupings are not present in our node dataset, so we label nodes with what we judge to be their most applicable description, using this same set of possible categories. We arrive at this description by a series of steps, where if a conclusion cannot be reached in the current step, we proceed to the next - 1) reading the documentation of the node 2) running the node in a flow 3) manually reading the code defining the node. This is a departure from IFTTT data, where categories are part of the ground truth, and

are instead derived here in a rather ad-hoc manner. However, the impact of this is minimal - categories are only used to illustrate the broad kinds of functionality nodes provide in Figure 7.1. The security classification of nodes is independent of this categorization, and does not affect the statistics arrived at about possible security violations.

Some modification is also necessary when applying the labels of "trigger" and "action" to Node-RED. Nodes in this platform are more general, and do not necessarily fit into the binary distinction between trigger and action. Nodes can be any combination of the two - triggers, actions, both, or none. We give nodes with no communication (configuration nodes generally) the new category of "Meta". Nodes with both trigger and action function are given entries in both. Figure 7.1 provides a visual summary of this labelling and categorization.

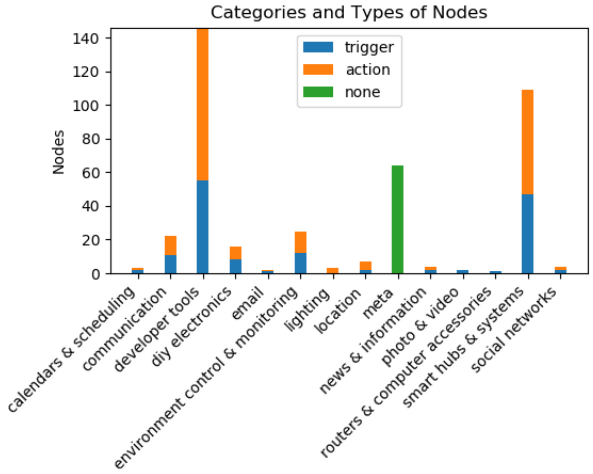

Figure 7.1: Node-RED nodes labelled with categories and trigger/action

7.2 Data

We find only a subset of categories present from the original 38 in the IFTTT dataset when labelling trigger/action nodes in Node-RED (see Figures 7.2 and 7.3). The difference in the Node-RED and IFTTT platforms can be illustrated by this - Node-RED is more oriented to custom-built flows, constructed from nodes with low-level functionality. In contrast, IFTTT provides applets (analogous to Node-RED nodes), preconfigured with a trigger and action (as well as filter code, which can be seen as similar to Node-RED parameters). This focus of Node-RED on more extensible usages manifests itself in the large number of nodes in the "developer tools" and "smart hubs & systems" categories.

We gather data from a manual examination of the top 100 node packages, as ranked by popularity in the Node-RED catalog. This yielded us 408 node definitions within

those packages. Since previous work does not present a rigorous labelling methodology, we propose a set of heuristics for attaching labels to nodes within Node-RED. In essence, a conservative approach is taken when labelling the impact of violations. For example, a node that provides output to Raspberry Pi output pins might usually be used for driving electronics like LEDs and motors. Using this view of the node, we can judge an untrusted action label as appropriate. However, these output pins might instead be used for communicating with some other electronic apparatus with internet access, and then a public action label is necessary. Therefore, taking the conservative option will yield us a labelling as public. We use some general guidelines, such as that output to local is available, input from local is private, and databases are private triggers and untrusted actions. Figures 7.2 and 7.3 give an overview of the classifications we apply to trigger and action nodes respectively.

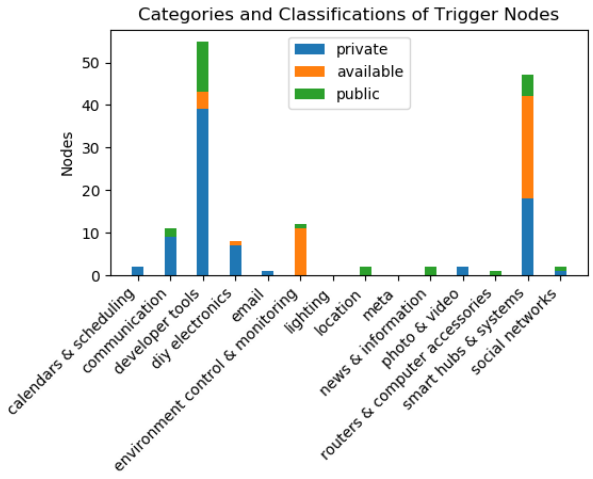

Figure 7.2: Security classification for Node-RED trigger nodes

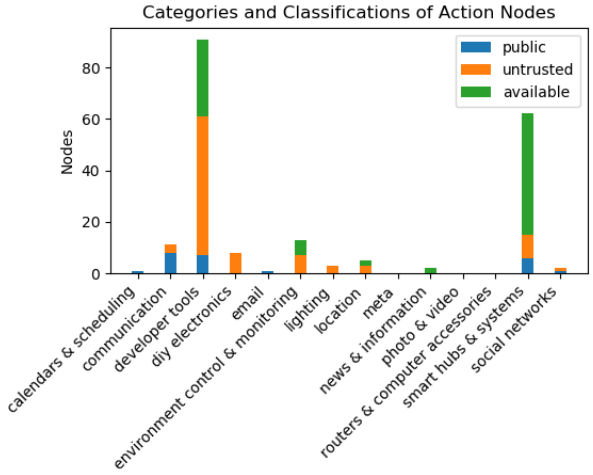

Figure 7.3: Security classification for Node-RED action nodes

7.3 Violations

The JSON specifying a flow specifies a graph(see Figure 2.3 for how this is parsed). After trimming disconnected nodes (generally configuration nodes) in the graph, we reconstruct the information flow between nodes. We cover the top 100 packages, representing only 4.71% of all Node-RED packages and 7.67% percent of all node definitions (from 5316 total). This classification completely covers 642 flows (54.36% of the 1181 total flows after pruning invalid), and only these are considered in this experiment. We find possible violations of security by tracing the graph for the descendants of trigger nodes, and looking for the classes of these action node descendants.

These possible security violations take the form of privacy violations (private triggers to public actions), integrity violations (any trigger to available actions), and availability violations (available triggers to available actions). Possible privacy violations occur in 70.40% of flows, integrity violations occur in 76.46%, and availability violations occur in 1.71%. We can form some tentative conclusions about the nature of Node-RED flows from comparison to the 30%, 98%, and 0.5% obtained for privacy, integrity, and availability violations in the IFTTT dataset. The more than doubling of privacy violations might be a symptom of the decreased granularity of many functions in Node-RED when compared to IFTTT. The lesser amount of possible integrity violations indicates that more actions taken in Node-RED are only available and not untrusted; an instance where available actions often occur can be seen in the common use-case of Node-RED as a visual dashboard.

7.4 Threats to Validity

This empirical study can be considered flawed, in that classification of nodes into fuzzy labels such as “Home Automation” is not defined directly in data, and has been interpreted by us in a rather ad-hoc manner instead. While this differs from the labels concretely defined as base truth in the dataset in the IFTTT-focused work by Bastys et al. (2018), we argue this has little impact as it does not influence the security labels of these nodes, from which we derive our statistics about possible security violations in the Node-RED platform [4].

8

Mitigations

Referring back to the attacker model developed in Chapter 3 and an attack misusing sensitive APIs in Chapter 4, we develop a mitigation strategy to prevent these violations. We also address how various actors in the Node-RED ecosystem can avoid using the vulnerable shared context feature, which we demonstrate in Chapter 6.

8.1 API Misuse

Bastys et al. (2018) use Information Flow Control (IFC) to guard against similar attacks, though in the IFTTT platform [4]. However, Node-RED uses general Node.js code instead of a centralized API where all actions are described. This lack of easily categorizable APIs leads to IFC being unsuitable to control API usage in the wider world of Node.js code. We fall back to access control as the basis of our security policy. Access control is motivated by packages containing sensitive functions, such as 'child_process' containing variants of 'exec'. An access control policy that entirely forbids the use of such functions could be valid if we want to form some security guarantee in Node.js code generally. However, Node-RED nodes need access to some sensitive functions with very general usage to perform their basic functionality - one example being the request library for communicating over HTTP. Therefore, a more fine-grained access control is necessary in these instances, where the system also distinguishes legal usage based on the arguments provided to sensitive functions.

Applying this access control to a singular node has little subtlety - the code will be run from top to bottom, with sensitive information provided via the message passed in. As long as the whitelist defining what sensitive functions are allowed with what parameters is particular enough, a security guarantee can be formed on that node's execution.

We also argue these node-level security guarantees combine to form a corresponding guarantee about the entire flow of which those nodes are members. The proof of this statement is left as future work. However, this guarantee only holds if the enforcement is able to distinguish between nodes. Otherwise, a security policy for a flow would have to be defined as a sum of all security policies for nodes within. Either this sum policy will be too restrictive, or not restrictive enough. For example, a flow with a filesystem access node piping its contents to an email node could either deny both their respective sensitive privileges (breaking the flow, but ensuring security),

or grant them both of their privileges (not ensuring security).

Finally, static or dynamic techniques can enforce this access control. Given that the whitelists specifying access control are being defined at the node level, and a node can change its behavior based on the message received at runtime, dynamic enforcement seems necessary. This decision to focus on a dynamic monitor is reinforced by descriptions in other work of JavaScript as a "highly dynamic" language [3].

8.1.1 Implementation - Monitor choices

We have outlined our access control method as needing a dynamic monitor with the ability to inspect which functions are called, and with what arguments. We implement a solution using Sinon to set up proxies around relevant objects, namely instances of libraries [47]. Sinon introduces the ability to add a "Spy" (proxy) around a function that intercepts calls and parameters. The concept of proxies was first introduced in ECMAScript6, where these wrapper objects attach traps to a wrapped value. These traps are triggered when operations are applied to the proxy, and the behavior of the wrapped value can be modified. As mentioned, this modification of behavior can include observations such as function calls and associated parameters, but it can also extend to more drastic changes such as not applying the operation to the wrapped value. Assuming the library is exposed as an object with prototype methods, these proxies can be set around these prototypes, ensuring that the monitor is able to observe all future instances of the library, such as any that might occur while running a particular flow containing nodes of interest.

However, this monitoring relies on the library indeed being an object with prototype methods defined. This assertion is not always true - many libraries simply export functions, which are used without an object providing them. A solution is to wrap these libraries in a manner that is conducive to our monitoring. Importing the wrapped library directly, avoiding the monitor, can be prevented by examining the ordering of libraries required - if the wrapper is not imported directly before the wrapped library, we know that the monitored version is not being used, and the program can then abort and a warning given about the requirement of using the wrapped version to ensure security. This ordering of library imports can be obtained by additionally setting a proxy around the method for importing libraries in Node.js, "require".

With this monitoring solution, the described implementation can be used as a security testing environment for a given node in question. The "node-red-test-helper" library allows a Node-RED flow can be run with input and events defined in a script, instead of requiring manual testing in the web interface [32]. With this and the monitor properly configured, one can test the usage of sensitive APIs by a particular node.

When looking at the security of an entire flow instead of a singular node, we also need some context to these function calls - what is allowed or not changes depending

on which node instance is calling the sensitive function in question. As previously mentioned, an additional detail that the monitor requires is the context in which the wrapped library is used. This is not guaranteed to be present in the immediate details that can be extracted from a proxy; in any given proxy we only have access to the context in terms of the calling environment. There is no way to chain back to which Node-RED node called the function if it is called while encapsulated in some intermediary object (such as a library or other object), or some other form of indirection used. Therefore, we instead rely on the nodes sending details of the events that start or stop the execution of their code, which is (normally) sufficient to identify in what context (which node) a sensitive function is being called in.

However, this implementation runs into problems when flows become more complex. Node-RED is built on Node.js - JavaScript itself is single-threaded, but Node.js registers callbacks to the underlying multi-threaded operating system to perform otherwise blocking actions. These callbacks are not guaranteed to be run in any defined order. This is a problem when facing flows that have callbacks being registered from multiple nodes - we can no longer say that a node's code will execute from start to end uninterrupted. Instead, the execution of these callbacks can be scheduled in such a way that the execution of various nodes become interleaved. This can occur either from there being disconnected graphs within the same flow, or a branch within any single graph. Figure 8.1 shows an example that runs in such an unspecified way. Interleaved execution makes the start and stop event signals, being used to determine the context of which node is calling a function, no longer sufficient. We find that from the flows published in the Node-RED catalogue, 835/1181 (70.7%) of flows have branches and 748/1181 (63.4%) have multiple graphs. This only leaves 178/1181 (15.1%) of these published flows as being covered with this implementation.

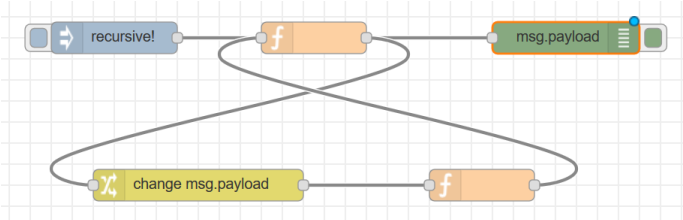

Figure 8.1: Flow resulting in interleaved output from Function nodes

8.2 Context

The vulnerability of shared context can be addressed in a number of ways by different actors within the Node-RED ecosystem.

8.2.1 Framework

The simplest solution we propose is that the framework provider remove the shared context functionality. If this direct mitigation is not an option, context could instead have more specificity attached to its API, with a policy around access instead of the coarse-grained levels of flow and global. This could take the form of giving per-node whitelists of variables in the shared context that a given node instance can read and/or write to.

8.2.2 Developer

We find that the developer can address context by avoiding using this functionality. This can be achieved by rewriting the program in question, where this rewriting varies based on the way that context is used. We construct these use-cases by generalizing how context is used into four separate categories, after manually examining the top 25 node and flows that our experiment finds using global and flow context.

Setting up debug flags and other environment configuration is a common use of context. This can be rewritten to use the already built-in Node-RED concept of configuration nodes.
Communication between nodes on the same flow through context can be rewritten to use explicit wiring.
Communication between nodes on different flows can be addressed by moving them into the same flow, and then explicitly wiring these nodes similarly to the previous use-case.
Finally, usage of context to provide shared resources can be addressed by moving nodes into the same flow if necessary, and then introducing configuration nodes and explicit wiring between them as required.

This rewriting is not without its faults. Combining flows and adding more wires and configuration nodes increases complexity for both the developer and the user. The developer might have to rewrite their nodes to also communicate information (that would otherwise be simply put into context) as additional messages, requiring additional wires. The user also has to contend with a far messier visual representation, additional necessary configuration, and a loss of modularity as nodes in the flow must be united.

8.2.3 User

We do not find as many instances where the user has a corresponding amount of control over context usage as the framework or developer. The user could always choose to inspect their nodes and flow configuration to selectively remove those that use context while preserving functionality, but achieving this task is not possible without creating modified versions of those nodes, thereby having the user assume

the role of developer. At the very least, the user can be presented with a notification when context is used, as implemented by the framework. They can then use their own discretion on whether the lack of security guarantee is acceptable with the sensitivity of the information provided to the flow.

8.3 Threats to Validity

As explained in Section 8.1.1, our implementation of the dynamic monitor to enforce the described security policy on flows only works for a small subset of the published flows; the implemented monitor only functions on linear non-branching flows containing no disconnected graphs. Our discussion of related work in Section 9.3 explains why this tradeoff is acceptable given the more dramatic flaws when considering other implementation choices. Theoretically, knowledge of the behavior of the Node.js scheduler could also enhance implementation viability, as the monitor will then be able to know which branch of code is running by shadowing the choices made in the scheduler. In practice, this still would be insufficient as the timing of any callback's execution is another factor which must be known to properly shadow the runtime execution.

In addition to only working on flows, our solution also comes with the burden of security policy generation. Developers of nodes will have to understand what sensitive APIs they are using, or the monitor will not allow their code to run. These function calls might be hidden to the application developer by being encapsulated in a package they import, leading us to suspect that an unaided developer might be unable to properly construct a security policy for their node. This knowledge requirement can be mitigated by the developers also having access to the security monitor - the monitor can throw errors specifying what sensitive functions are used against an initial policy, and the developer can take steps to add these functions to the whitelist for their application. However, this procedure does not guarantee that the developer adds all necessary entries to the whitelist, as all code paths are not necessarily reached in testing during development. In a sense, perfect knowledge of the usage of APIs is necessary. This burden could be shifted to the framework level, where additional guidance can be given to developers by the Node-RED platform, perhaps by having an index of what sensitive functions can be accessed through other commonly used packages and functions.

The user also has the burden of verifying a security policy is acceptable. For example, a node for sending emails should only be able to send said emails to specific configured addresses (which can be automatically derived from the parametrization of the node). However, a legal policy also could instead give the node the ability to send to all addresses, including that of an attacker interested in obtaining the exfiltrated information. The task of identifying these subtle differences could be too much for a non-technical audience. As before, the burden can also be shifted to the framework - the Node-RED platform can introduce additional steps to the publishing of nodes, where instead of a new node package being automatically added to the

index, some verification of the acceptability of a node's security policy is performed before the package is officially listed. While this does include some manual labor in evaluating whitelists, this effort involved in this task is far diminished in magnitude from having to read and understand the node's actual JavaScript code, which is the only way to form a similar guarantee about security in the current system.

9
Related Work

9.1 IoT Security

A large number of works study the challenges facing the IoT regarding security. Pawar and Ghumbre (2016) are one instance of this, when they demonstrate the usage of IoT to a handful of applications, mostly medical. This survey identifies security challenges and demonstrates the uses of cryptographic algorithms as countermeasures [40].

Our survey of the security of IoT apps in Node-RED is inspired by the work of Bastys et al. (2018) analyzing the security of the competing If This Then That IoT platform [4]. Information Flow Control (IFC) is suggested in that work as a suitable for forming security guarantees, and a prototype is demonstrated. This work also includes a security classification to get some indication of the prevalence of security violations, which we replicate with Node-RED in Section 7.

9.2 Node.js and NPM Security

As a shifting foundation affects the house built on it, so does the security of the underlying Node.js runtime and Node Package Manager (NPM) ecosystem affect Node-RED. Various works, including that by Zimmermann et al. (2019) identify security flaws in Node.js and the NPM [55]. This study identifies the issues that come with package dependencies, namely inclusion of vulnerable or malicious code, ease of inserting this code, and difficulty in fixing these flaws. Pfretzschner and ben Othmane (2017) also study Node.js dependencies, but also identify a specific attack using shared variables to manipulate application behaviour from library code [41]. Decan et al. (2018) provide data through an empirical study on vulnerabilities in the NPM dependency network [11]. Vaidya et al. (2019) also identify security issues in Node.js, but also do this more generally in other language-based ecosystems such as PyPI in Python [53]. Kula et al. (2017) provide data around the occurrences of micro-packages in the NPM ecosystem, which can exacerbate the issues that occur with package dependencies [15].

9.3 Dynamic Monitoring of Node.js

The dynamic nature of JavaScript requires that the sort of detailed analysis required by the security solutions we propose also be dynamic. Andreasen et al. (2017) survey available methods for dynamic analysis for JavaScript, and outlines three general categories: runtime instrumentation, source code instrumentation, and metacircular interpreters [3]. In addition, this survey also mentions that other lightweight methods are a possibility for achieving dynamic monitoring of JS code.

9.3.1 Runtime Instrumentation

Two leading runtime instrumentation tools are DProf and NodeProf, described by Gregg and Mauro (2011) and Haiyang et al. (2018) respectively [12, 50]. DProf functions at the lowest level of these options, instrumenting a program at the instruction level. This allows the tool to be used on a variety of languages, including JavaScript, but it is not oriented to that use-case in particular. NodeProf instead instruments at the abstract syntax tree (AST) level, and is specifically made as a dynamic analysis framework for Node.js.

However, we find in testing that despite this stated support, some Node.js language features, such as "module.exports", which are heavily used in Node-RED code are not implemented in NodeProf yet. In addition, by functioning at this lower-than-code level, we must run the entire system, including the Node-RED server, through a runtime instrumentation system to obtain results. Therefore, we choose to use the simpler method outlined in Section 8.1.1. NodeMOP is a more recent Monitor-Oriented-Programming (MOP) tool built on top of NodeProf that also looks promising in our application [44]. However we were unable to obtain running code for this project, and furthermore also functions at the AST level, leading to it probably not having fit our use-case similarly to NodeProf.

9.3.2 Source Code Instrumentation

The particular method of source-code level instrumentation of Node.js code we find continually referred to is Jalangi [46]. Linvail, a similar shadow execution platform also promises Node.js support [6]. Ancona et al. (2017) use extensions to Jalangi to record trace-expressions to verify correct usage of API functions [2].

We find that we can successfully instrument a given Node-RED node's code with Jalangi. However, this instrumented code is not able to be simply run in Node.js again. Instead, it must be run by a Jalangi-provided executable. This detail again brings back the undesirable necessity of running the entire Node-RED system through this tool, and led to us not using this method in our implementation.

9.3.3 Metacircular Interpreter

Photon is mentioned as a notable metacircular interpreter for dynamic analysis [16]. Jalangi, through its support for shadow values, can also be classified as such. However, we cannot find substantial reference to practical usage of Photon, and Jalangi has already been eliminated from the running for possible implementations of our monitor for the previously mentioned reasons.

9.3.4 Lightweight Methods

With the elimination of the previous three methods, that leaves us only the vaguely defined lightweight methods. The approach we and Laurent et al. (2015) use is based on the ability to dynamically overwrite the prototypes of objects in JavaScript, and set up proxies around them [7].

Membrane-based analysis also bears some resemblance to our use of proxies. First described by Miller (2006), membranes are a "defensive programming pattern used to intermediate between sub-components of an application" [17, 51]. While not specific to JavaScript or Node.js, this pattern is implemented in Node.js by recursively wrapping an object in a proxy in such a way that this wrapped object also only can return wrapped objects. This has some similarity to our wrapping of "require", necessary for us to detect ordering of imports. Staicu et al. (2020) provide an example of this technique applied to Node.js, where they isolate libraries to automatically extract taint specifications [49]. While this isolation of libraries is also what we aim to achieve with our monitor, the security that a membrane achieves is not necessary for our work. We only care about monitoring specific sensitive libraries, instead of the complete isolation recursively built from a single component, which is a membrane's main purpose.

9.4 Node-RED

Node-RED has been the subject of some academic interest. Clerissi et al. (2018) model Node-RED and perform checks over system properties to provide early system validation [8]. However, the executable Mocha tests used in this work bring us no closer to analyzing the properties of functions and their usage that we require to enforce a security policy.

Nguyen et al. (2018) use the ThingML modeling language to generate "gatekeepers" that can be deployed in edge servers to control IoT tenant applications based on an execution policy [18]. Node-RED is mentioned as one possible execution engine that a tenant app can run in, though they do not go into any detail beyond this.

Toma et al. (2019) outline an IoT solution to closely monitor pollution using Node-RED [52]. Security is analyzed in an extensive section of this paper, but only in

the context of the MQTT protocol, and not the specifics of the Node-RED platform.

Bröring et al. (2019) collect a Node-RED dataset with their NOde-red library eVAluation (NOVA) approach [5]. Similar to our data-gathering, this approach also collects node packages and flow definitions. These authors also note the ambiguous references to nodes in flows, where member nodes are only identified by a simple string, which can refer to any one of multiple nodes published in different packages. However, the implications that this property has on security, such as squatting attacks, are not considered (refer to Chapter 3).

These examples are mostly indicative of what security research involving Node-RED exists, even to a more tangential degree. We were unable to find research analyzing Node-RED from the specific language security perspective while developing this work.

9.4.1 COMPOSE

However, we have become aware of at one project that has superficial similarity to the work done in this thesis. Titled the Collaborative Open Market to Place Objects at your Service (COMPOSE), this project, conducted from 2012 to 2015, bears more than a passing resemblance to our own work. Schreckling et al. (2016) published a report titled "Data-Centric Security for the IoT" describing entire COMPOSE framework, wherein they explain how the platform is built on top of glue.things (in turn an extension of Node-RED), integrating JSFlow into the execution environment to enforce security policies [14, 45]. This project was in turn used in the development of a SEcure DAta ContRol for the IoT (SEDARI) [13]. However, this larger-scale smart-city platform also seems to have ended its development in 2017, and does not deviate from COMPOSE in terms of how its security framework is built.

We agree with the characterization that the authors of COMPOSE form of some basic difficulties in attempting to form security policies for IoT applications. Existing security architectures are not directly applicable; IoT platforms instead have "unpredictable contexts in which data is processed and applications are executed" [45]. COMPOSE's authors also recognize the necessity of making security policies understandable to non-experts.

However, we address these fundamental difficulties in different ways. Moreoever, COMPOSE was targeted at the "development of a full end-to-end solution for developing Internet of Things (IoT) applications and services" - the scope of this project is much greater than what is considered within this thesis. Their main contributions include three separate items: "COMPOSE Marketplace, the runtime engine, and the Ingestion layer"; however, the only results that are applicable to our security interests are within their runtime engine. As we have previously mentioned, their runtime engine includes a security framework using JSFlow to enforce policies during execution. This dynamic monitoring differs from our own in that

the policy it enforces is granular to the data instead of API, as one expects from an application of IFC. Besides necessarily being concerned with provenance of data, COMPOSE also introduces reputation as another dimension of these security policies, where reputation is derived for an entity by factors such as popularity, activity, and feedback. In addition, COMPOSE does not consider any particular vulnerabilities within the Node-RED platform, such as the context feature that we highlight. Finally, COMPOSE only implicitly defines their threat modeling by providing security mechanisms, and neglects to provide a concrete attacker or attacks.

In comparison to this previous work, we also focus more effort on providing empirical studies. Instead of this practical direction, COMPOSE instead ran a pilot program, described as an "in-store analytics solution for monitoring the behaviour of customers in a grocery store" [10]. While this did successfully display the ability to build a full-fledged IoT application using their end-to-end solution, this practical display was not focused on evaluating the security properties of their system. Security is only mentioned as a footnote: "Finally, security was used to control the access to data. In the specific pilot setting this feature was not used, but considered as extremely relevant by store executives".

10
Conclusion

In summary, this paper provides a novel analysis of the security of the Node-RED platform. We form a picture of the architecture of the platform, describing the publishing system and possible attacker models. Finally, we collect and analyze data to give a more concrete picture of how the Node-RED platform is used, while also revealing the prevalence of the opportunities for attack and instances of vulnerabilities.

This paper used the following three research questions to guide us during our analysis of the security of the Node-RED platform. We have answered each of these in their respective sections in this work as specified in Section 1.4, but will reiterate here our main conclusions regarding these questions.

1. *What attacker models and associated attacks are present in Node-RED? Are there specific vulnerabilities in this platform as well?*

   The application developer, in the form of publishing nodes and flows, is an attacker able to influence privacy, integrity, and availability.

   We demonstrate attacks that necessitate an API access control policy. This policy forms security guarantees about the execution of nodes that is currently lacking in the Node-RED system. We also identify a shared context feature as a separate vulnerability.

   Additionally, Node-RED is also vulnerable to issues found more generally in Node.js and NPM. We highlight the design decision of having flows identify what node's code to run, based on a string, as a factor that eases possible typo-squatting attacks.

2. *How prevalent are these issues in the Node-RED ecosystem?*

   Based on the results of our empirical studies, Node-RED is vulnerable to both attacks derived from its building upon the NPM ecosystem, as well as the API misuse attacks and context vulnerabilities identified in this paper.

3. *How can language-based security techniques be used to address these attacks and vulnerabilities?*

   We implement a mitigation to attacks misusing APIs, by using proxies to en-

force an access control whitelist in a dynamic monitor. This external monitor proxies prototypes of sensitive libraries, so that any usage by nodes can be controlled. We suggest the use of this mitigation technique as a security tester for a Node-RED node, though it can also apply as a security monitor for a subset of flows that satisfy certain conditions. Finally, we propose solutions for the actors within the Node-RED ecosystem to reduce the vulnerable usage of shared context in the platform.

10.1 Future Work

Following from the threats to validity we have previously detailed in their respective chapters, there are at least two clear directions for future work - improving the collected dataset, and improving the implementation of our security monitor.

The dataset we have gathered of flows is small considering the amount of published nodes. Flows are just JSON files with a specific formatting, and need not necessarily be only found in the official catalog. Instead, a scraper can go through some other public index to gather all publicly visible flow specifications. This hopefully will yield a more representative dataset, with more unique nodes being used in these flows. For example, Arlo, a manufacturer of web-connected cameras has a relevant Node-RED node published [27]. However, there are no flows in the official catalog that use this node definition. Despite this, a simple Google search yields a blog post containing a Node-RED flow dealing with Arlo cameras as the fifth result [1].

Our implementation of a security monitor is also lacking in that it can only reliably run on a fraction of the total flows, as detailed in Section 8.1.1. In Section 9.3, we examine other choices for implementing this dynamic monitor and how they are unsuitable to the enforcement of the security policy we have defined in this work. However, further investigation may yield a more comprehensive solution.

Bibliography

[1] 2RS2NS. Setting custom arlo modes with nodered. `http://2rs2ns.com/setting-custom-arlo-mode-with-node-red/`, 2018. [Online; accessed 7-January-2020].

[2] Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca. Towards runtime monitoring of node.js and its application to the internet of things. In Danilo Pianini and Guido Salvaneschi, editors, *Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Turin, Italy, September 18, 2017*, volume 264 of *EPTCS*, pages 27–42, 2017.

[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, 50(5), September 2017.

[4] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If this then what?: Controlling flows in iot apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1102–1119, New York, NY, USA, 2018. ACM.

[5] Arne Bröring, Victor Charpenay, Darko Anicic, and Sebastien Püech. Nova: A knowledge base for the node-red iot ecosystem. In Pascal Hitzler, Sabrina Kirrane, Olaf Hartig, Victor de Boer, Maria-Esther Vidal, Maria Maleshkova, Stefan Schlobach, Karl Hammar, Nelia Lasierra, Steffen Stadtmüller, Katja Hose, and Ruben Verborgh, editors, *The Semantic Web: ESWC 2019 Satellite Events*, pages 257–261, Cham, 2019. Springer International Publishing.

[6] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail: A general-purpose platform for shadow execution of javascript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 260–270. IEEE, 2016.

[7] Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. Dynamic analysis using javascript proxies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 813–814, Piscataway, NJ, USA, 2015. IEEE Press.

[8] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. Towards an approach for developing and testing node-red iot systems. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Ensemble-Based Software Engineering*, EnSEmble 2018, pages 1–8, New York, NY, USA, 2018. ACM.

[9] Roger Collier. Nhs ransomware attack spreads worldwide. *CMAJ*, 189(22):E786–E787, 2017.

[10] Community Research and Development Information Service (CORDIS). Collaborative open market to place objects at your service. `https://cordis.europa.eu/project/id/317862`, 22 April 2017. [Online; accessed 7-January-2020].

[11] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 181–191, New York, NY, USA, 2018. ACM.

[12] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[13] Rolf Schillinger Günther Pernul, Guido Schryen. *Security in Highly Connected IT Systems*. Bavarian Research Alliance FORSEC, 2017.

[14] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. Glue.things: A mashup platform for wiring the internet of things with the internet of services. In *Proceedings of the 5th International Workshop on Web of Things*, WoT '14, page 16–21, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *CoRR*, abs/1709.04638, 2017.

[16] Erick Lavoie, Bruno Dufour, and Marc Feeley. Harnessing performance for flexibility in instrumenting a virtual machine for javascript through metacircularity. 2012.

[17] Mark Miller. *Robust Composition: Towards a Uni ed Approach to Access Control and Concurrency Control*. Johns Hopkins University, 2006.

[18] Phu H. Nguyen, Phu H. Phung, and Hong-Linh Truong. A security policy enforcement framework for controlling iot tenant applications in the edge. In *Proceedings of the 8th International Conference on the Internet of Things*, IOT '18, pages 4:1–4:8, New York, NY, USA, 2018. ACM.

[19] Node-RED. Handling errors. `https://nodered.org/docs/user-guide/handling-errors`, 2020. [Online; accessed 7-January-2020].

[20] Node.RED. Design: Flow file format v2. `https://github.com/node-red/node-red/wiki/Design:-Flow-file-format-v2`, 2015. [Online; accessed 7-January-2020].

[21] Node.RED. btsimonh's node-opencv motion detection (2017-11-02). `https://flows.nodered.org/flow/33a93ac5418009993d38c00009ef453e`, 2020. [Online; accessed 7-January-2020].

[22] Node.RED. Community node module catalogue. `https://github.com/node-red/catalogue.nodered.org`, 2020. [Online; accessed 7-January-2020].

[23] Node.RED. Creating your second flow. `https://nodered.org/docs/tutorials/second-flow`, 2020. [Online; accessed 7-January-2020].

[24] Node.RED. Library - node-red. `https://flows.nodered.org/`, 2020. [Online; accessed 7-January-2020].

[25] Node.RED. node-opencv motion to video. https://flows.nodered.org/flow/c172899be094e2cf37a92f32b7c47635, 2020. [Online; accessed 7-January-2020].

[26] Node.RED. Node-red. https://nodered.org/, 2020. [Online; accessed 7-January-2020].

[27] Node.RED. node-red-contrib-arlo. https://flows.nodered.org/node/node-red-contrib-arlo, 2020. [Online; accessed 7-January-2020].

[28] Node.RED. node-red-contrib-func-exec. https://flows.nodered.org/node/node-red-contrib-func-exec, 2020. [Online; accessed 7-January-2020].

[29] Node.RED. node-red-contrib-openjtalk. https://flows.nodered.org/node/node-red-contrib-openjtalk, 2020. [Online; accessed 7-January-2020].

[30] Node.RED. node-red-node-dropbox. https://flows.nodered.org/node/node-red-node-dropbox, 2020. [Online; accessed 7-January-2020].

[31] Node.RED. node-red-nodes/social/email/61-email.js. https://github.com/node-red/node-red-nodes/blob/master/social/email/61-email.js, 2020. [Online; accessed 7-January-2020].

[32] Node.RED. node-red/node-red-node-test-helper. https://github.com/node-red/node-red-node-test-helper, 2020. [Online; accessed 7-January-2020].

[33] Node.RED. Packaging. https://nodered.org/docs/creating-nodes/packaging, 2020. [Online; accessed 7-January-2020].

[34] Node.RED. Running on raspberry pi. https://nodered.org/docs/getting-started/raspberrypi, 2020. [Online; accessed 7-January-2020].

[35] Node.RED. Sprinkler control example. https://flows.nodered.org/flow/60867ba2acfc317c5710b0c07cc071da, 2020. [Online; accessed 7-January-2020].

[36] Node.RED. Test node-opencv flow. https://flows.nodered.org/flow/b18e4eed8317d721db9c0b7c65755dc4, 2020. [Online; accessed 7-January-2020].

[37] Node.RED. Water utility complete example. https://flows.nodered.org/flow/b1d00d13f1db357ac686f9379731060c, 2020. [Online; accessed 7-January-2020].

[38] Node.RED. Working with context. https://nodered.org/docs/user-guide/context, 2020. [Online; accessed 7-January-2020].

[39] Gavin O'Gorman and Geoff McDonald. *Ransomware: A growing menace*. Symantec Corporation, 2012.

[40] A. B. Pawar and S. Ghumbre. A survey on iot applications, security challenges and counter measures. In *2016 International Conference on Computing, Analytics and Security Trends (CAST)*, pages 294–299, Dec 2016.

[41] Brian Pfretzschner and Lotfi ben Othmane. Identification of dependency-based attacks on node.js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, pages 68:1–68:6, New York, NY, USA, 2017. ACM.

[42] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[43] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. Cryptolock (and drop it): stopping ransomware attacks on user data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 303–312. IEEE, 2016.

[44] Filippo Schiavio, Haiyang Sun, Daniele Bonetta, Andrea Rosà, and Walter Binder. Nodemop: Runtime verification for node.js applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1794–1801, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Daniel Schreckling, Juan David Parra, Charalampos Doukas, and Joachim Posegga. Data-centric security for the iot. In Benny Mandler, Johann Marquez-Barja, Miguel Elias Mitre Campista, Dagmar Cagáňová, Hakima Chaouchi, Sherali Zeadally, Mohamad Badra, Stefano Giordano, Maria Fazio, Andrey Somov, and Radu-Laurentiu Vieriu, editors, *Internet of Things. IoT Infrastructures*, pages 77–86, Cham, 2016. Springer International Publishing.

[46] 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*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[47] Sinon.JS. Sinon.js - standalone test spies, stubs and mocks for javascript. works with any unit testing framework. `https://sinonjs.org/`, 2020. [Online; accessed 7-January-2020].

[48] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: understanding and automatically preventing injection attacks on NODE.JS. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[49] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting taint specifications for JavaScript libraries. In *Proc. 42nd International Conference on Software Engineering (ICSE)*, May 2020.

[50] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 196–206, New York, NY, USA, 2018. ACM.

[51] Tom Van Cutsem. Isolating application sub-components with membranes. `https://tvcutsem.github.io/membranes`, Jul 22, 2018. [Online; accessed 7-January-2020].

[52] Cristian Toma, Andrei Alexandru, Marius Popa, and Alin Zamfiroiu. Iot solution for smart cities' pollution monitoring and the security challenges. *Sensors*, 19(15):3401, 2019.

[53] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. Security issues in language-based sofware ecosystems. *CoRR*, abs/1903.02613, 2019.

[54] I. Yaqoob, E. Ahmed, I. A. T. Hashem, A. I. A. Ahmed, A. Gani, M. Imran, and M. Guizani. Internet of things architecture: Recent advances, taxonomy, requirements, and open challenges. *IEEE Wireless Communications*, 24(3):10–16, June 2017.

[55] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. *CoRR*, abs/1902.09217, 2019.