Compromising File Systems with npm Dependencies

Jonas Zohren | TU Dortmund University

Abstract

Most popular programming languages today thrive due to the ability to seamlessly reuse third-party, open-source code. Be it Python's pip, Rust's Cargo or JavaScript's npm: All of them utilize a centralized repository to aggregate common code from web frameworks to small utilities for handling zip files.

However, during installation, JavaScript packages from the npm ecosystem can execute arbitrary code on developers' machines.

We built a custom detection pipeline to analyze ~ 250,000 installation scripts in the npm ecosystem, a simple web app to view results and found a few malicious scripts during manual inspection.

RQ1: How (and how often) do packages use scripts?

The npm database contained 1,903,676 packages at the time of the analysis. 13.06% of which had at least one published version which used install scripts.

The most common (benign) use-cases for install scripts are:

1) Setup of the development environment, e.g. installing git hooks to check formatting or editing config files

2) Compiling native code and bindings to Node.js, often via node-gyp

3) Downloading platform-specific binaries

Those uses might be valid workarounds for perceived limitations of the npm CLI, but also sidestep its security mechanisms.

RQ2: In which way are they used maliciously?

Npm install scripts are one of the most common entry points for malicious npm packages. Their execution is invisible to users by default, and is open to any package in the large dependency chain of typical JavaScript projects. In a recent incident, the package `node-ipc` began overwriting files of users with supposedly Russian IPs, as a reaction to the Russian invasion of Ukraine.

While this incident was immediately noticed, non-hacktivism malware usually attempts to go unnoticed.

One such type of malware was found during our manual analysis. We noticed a pattern of similar packages, which extracted environment variables (which can contain a plethora of sensitive data) and sent them to a remote host.

These packages used a few layers of light obfuscation:

- After releasing the malicious code, a new version without it was released, hiding it from scanners which only focussed on the newest version for each package (like our pipeline)
- The receiving server's address was split and then joined
- The payload was encoded in base64

This listing shows one of the 15 malware samples we found, reported and got GitHub (npm's owner) to remove:

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

function main() {
  var data = process.env || {};
  if (Object.keys(data).length < 10) {
    return;
  }

  req = http.request({
    host: ['964a4e924030bf1dbadda43f51807238', 'm', 'pipedream', 'net'
        ].join('.'),
    path: '/' + (process.env.npm_package_name || ''),
    method: 'POST'
  }).on('error', function (err) {
  });

  req.write(Buffer.from(JSON.stringify(process.env)).toString('base64')
      );
  req.end();
}

main();
```

Other researchers routinely found packages which stole cryptocurrency wallet keys and authentication tokens for apps like Slack and Discord.

RQ3: Is it possible to detect that?

While generally possible, malware detection warrants the existence of an entire industry and ongoing research. Due to the time limits of a bachelor thesis, we focus on trackable tampering with the file system.

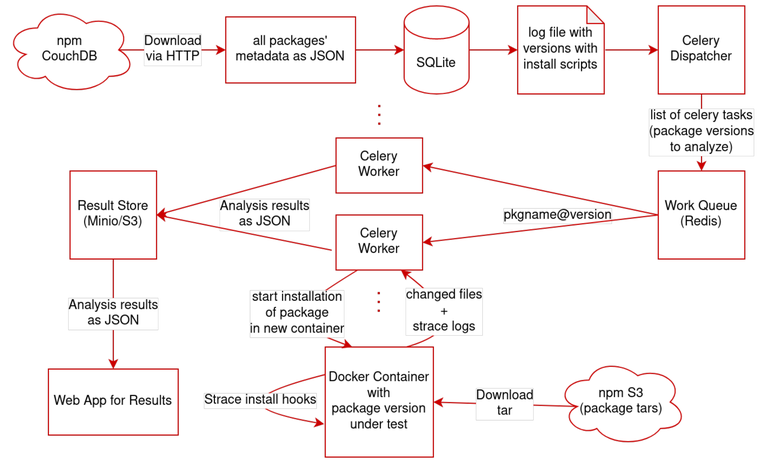
To answer this research question, we developed a new analysis pipeline, which dynamically evaluates the file system changes of package installations. Due to resource constraints, it has to run and scale out to any x86 Linux machine, given it has a network connection.

The core of this pipeline is the installation of an npm package inside a Docker container. The installation's file related system calls are logged via the 'strace' tool. The overall changes to the file system are obtained via docker's 'diff' functionality.

Scaling is enabled through the use of the python work queue 'Celery', which takes a list of npm packages to analyze and spreads them out to worker machines. Workers start a new container, run the installation, aggregate file system access and send the results as JSON files to 'Minio', an S3 compatible object storage server, from which they can later be retrieved for analysis.

A metric we call 'risk factor' is also calculated, based on a weighted logarithmic product of different file access types (read, write, sensitive paths).

The following graphic provides an overview over the whole pipeline the way it was used for this thesis



To aid analysis, we created a simple web app, which accesses results in Minio. Loading the dependency file of an npm project shows a ranked list of the dependencies' riskiness and further details.

For this thesis, we ran this pipeline on the approximately 250,000 newest versions of packages (with install scripts) over the course of a few days. To our surprise, none of the analyzed packages triggered our maliciousness threshold.

We attribute this lack of findings to two possible reasons:

1) npm security is an active field of research and improvement. During our work on this thesis, multiple researchers used similar techniques to find, report and remove malicious packages, which we now could not find. Companies like socket.dev began to run extensive static analysis on many packages.

2) Due to resource constraints, we only analyzed the newest package version, which (as shown before) does not always contain the malware code.

Nevertheless, we have proven that one can implement an analysis framework, capable of analyzing a large number of packages on commodity hardware. The analysis step is extensible and could be extended to analyze network traffic, other system calls, as proven by the works of the 'Open Source Security Foundation'.

RQ4: How can install script security be improved?

- If possible, do not use install scripts at all. Instruct npm to ignore scripts with the argument '--ignore-scripts'
- To protect a developer's machine, remote development tools like 'GitHub Codespaces' and 'GitPod.io' spin up an ephemeral Linux environment in the cloud, containing a potential breach to a single code base.

dortmund university

zohren.xyz/ba-thesis.pdf
jonas.zohren@tu-dortmund.de