npm Source Planting

```
{
    "name":  "Ave Ottens",
    "email": "j.m.a.ottens@student.tue.nl",
    "id":    1312405,
    "date":  [2020, 04, 12]
}
```

Introduction

In this report I discuss my malicious JavaScript code that targets Node.js developers using the npm package manager. The code modifies the package manager and takes over its `pack` command. It then silently attaches itself to unrelated packages that get published on that computer.

Exploration

I started digging through the npm/cli repository with big ambitions. I wanted to obfuscate my malware, steal authentication tokens and use them to publish packages without user input, maybe host my own repository for testing, etc. This turned out to be much harder than expected, and I had to prune features that felt out of scope.

This led to the following objectives:

- Plant arbitrary code into the npm source code when installing a package.
- Sneak arbitrary code into a package before its packed to a tarball.

With this in mind I dove into the source code again, armed with a notepad. This project made me appreciate the importance of documenting your steps.

First, I wanted to know a little bit more about the structure of the code. Running `where npm` in a Windows command prompt returns the location of the script. In my case this points to a batch file, where we find the following hint:

```
SET "NODE_EXE=%~dp0\node.exe"
...
SET "NPM_CLI_JS=%~dp0\node_modules\npm\bin\npm-cli.js"
...
"%NODE_EXE%" "%NPM_CLI_JS%" %*
```

Aha! The entry point for npm seems to be `npm-cli.js`, in the `/bin` folder. I wanted the malware to execute only on specific commands, so my next task was finding out how those got parsed. JavaScript is a dynamic language, and you can get away with lots of variable gymnastics. `npm-cli.js` was no different, and I had a hard time figuring out what the variables were supposed to mean.

At this point I decided to try adding some breakpoints. I cloned the repository and opened it in Visual Studio Code. After some trial and error I managed to make a debug target that looks as follows:

```
{
    // ...
    "program": "${workspaceFolder}\\bin\\npm-cli.js",
    "args": ["--verbose", "pack"],
    "cwd": "C:\\Users\\ave\\my-package"
    "console": "integratedTerminal"
}
```

Now we can simulate npm launches from other folders, using whatever arguments we want! The integrated terminal makes it possible to execute expressions while the program is paused, and we can use this to experiment with code injections and whether they would work.

Initially I wanted to hook code into the `publish` command, which is responsible for uploading packages to the servers. Managing credentials was a bit of a pain however, and I didn't want to break something on the server. Therefore I decided to target the `pack` command instead. This puts the package into a tarball, which could be uploaded to the server at a later time. It was a great decision, because this made debugging and testing much easier.

There was one noteworthy moment while analysing the `publish` command. npm kept throwing an E401 error, since I wasn't logged in and not authenticated to access the server. I saw this as an opportunity to find the authentication code, and wanted to find out where this error was thrown. If I could predict whether a call to `publish` would fail, I could make the malware more sneaky. No point in injecting code if the package would never make it on the repository!

I did a search for "401" in the source code, and the hits were extremely cluttered. One thing caught my eye though, an entry in CHANGELOG.md:

```
* [`857c2138d`](...)
  [#20032](https://github.com/npm/npm/pull/20032)
  Fix bug where unauthenticated errors would get reported as both 404s and
  401s, i.e. `npm ERR!  404 Registry returned 401`.  In these cases the error
  message will now be much more informative.
  ([@iarna](https://github.com/iarna))
```

I used the link to the issue (#20032) to find out what file was edited, hoping this would point me towards the error handler. I had no luck however, and I abandoned the idea in favour of the `pack` command.

After scattering breakpoints far and wide, I found out the file was running `require('../lib/pack.js')`. This is done dynamically, and not immediately obvious when looking at the source code.

I wanted the packing process to work as follows:

1. Malicious code is dropped into a package.
2. The package is packed into a tarball.
3. The code is removed again, so the user can't tell anything fishy went on.

Therefore I needed to find two places in the `pack.js` file. One right before the packing happens, and one right after it is finished. After carefully stepping through the code I managed to find two candidates. I already sketched out some implementation, but with this new understanding I could finally make it robust!

Strategy

The npm interface is a Node.js package itself - which is just a folder with a valid `package.json`. Because JavaScript is interpreted, all of the source can be read, but I was horrified to find out you can actually edit the code! This only works on some systems and configurations. Sometimes the user account doesn't have permission to write to these folders. I found that using nvm, the node version manager, made editing the code possible. When npm is installed using the tool, the folders belong to the user, and not the system.

Node.js has no sandbox whatsoever. Any syscall can be made, as long as the operating system allows it. Any package manager is a target for abuse, but this makes JavaScript packages especially risky. The ecosystem also sees a large amount of code reuse, and installing one package might actually reel in dozens of dependencies.

There have been plenty of targeted attacks, where bad actors somehow gain access to a popular package. They can then drop malware to steal credentials, which gives them access to more packages, and the cycle of life continues. Is there a way to automate this breach of trust? Yes! What if a developer was unknowingly responsible for attaching malware to their package? We wouldn't need to steal anything at all, and the developer would not be alerted to the malicious code.

To accomplish this, I wrote some utility functions and two scripts which are to be attached to the malicious package. They are all JavaScript files, and these may get executed automatically once the user installs the package. Their exact function and role in the bigger picture are explained below.

./lib/process.js

A lot of useful information can be extracted from the command line. This file provides a wrapper for spawning child processes. It waits until the process is done, and returns the output as a string. On Windows, the `cmd.exe` shell is used. On Unix, the `bash` shell is used. We can detect the operating system quite easily by inspecting `process.platform`. It is set to `"win32"` on all flavours of windows. I did not consider MacOS for this project, since I didn't have a way to test it.

./lib/config.js

We can use `npm config list` to show the entire npm configuration. When we pass the `--json` flag, the config is returned as JSON, and it can be deserialized into a JavaScript object. `config.js` returns that object, and it is used in other scripts.

./lib/rewrite.js

This is the real core of the malware. It allows us to prepend, append, and inject arbitrary lines in a text file. Normally, we can only overwrite or append to a file. To do this, it takes the following steps:

1. Rename the original file to file.temp
2. Read the entire file into memory.
3. Split the resulting string into an array of lines.
4. For each line, call a function that modifies the line based on some logic.
5. Prepend and append lines if they were given.
6. Join the lines back together into a string.
7. Write the string to the new file.
8. Delete the temporary file.

This sequence is not the most memory- or time-friendly, but for small files it will do. It also blocks Node.js until it is done, wasting some time while reading from disk.

The original implementation used asynchronous filesystem functions. This allowed us to parse the text while it was being read from the disk. While this is much faster, and only requires enough memory for one line at a time, it did have a major drawback. An asynchronous function needs to be passed a callback, which specifies what to do after the function has finished. This isn't normally a problem, but a call to this function had to be (indirectly) injected into the npm source code. For a synchronous function this is quite easy, you put it on a new line and you're done. However, for a callback you need to wrap all of the other code in braces somehow, without breaking the existing code. I was a bit stumped by the npm source code to begin with, so I decided it was much better to just take the performance hit and make the function synchronous.

./scripts/install.js

Modifying the npm source code happens in this file. First, it finds the location of the npm install. On Windows, this can be found in the npm config. The prefix property points to the Node.js folder, and on appending `node_modules/npm` we have a path to the npm folder. This doesn't work for Unix machines. Here, we execute the following command on the shell:

```
$ dirname $(readlink -f $(which npm))
```

`which npm` returns the path to the npm script, which is a symbolic link. Using `readlink -f` we can resolve it to its actual location, and `dirname` then returns the path to the folder containing the script: `npm/bin`. We append `..` and normalize the path, which gives us the npm root folder.

Now, all these utility functions and scripts need to be copied into the npm folder, so they are accessible at a later time. We can copy folders recursively using a 3rd party package, but in this case we can't download any dependencies. Therefore we just create and copy the right files and folders explicitly.

After this step, the npm root folder contains a `malware` folder, which houses all of our code. This could be obfuscated and hidden more deliberately, but that wasn't in the scope of this project. In the event that the install script is run twice, the malware folder already exists. The script then throws and error, but we can catch it and hide it from the user if needed. That way they won't be alerted to any suspicious activity.

Now we use the rewrite function to inject code into `npm/lib/pack.js`. We make the following changes:

```
// before
BB.all(
    args.map((arg) => pack_(arg, cwd))
).then((tarballs) => {...})
```

```
// after
BB.all(
    args.map((arg) => {
        require('../malware/scripts/package.js').dirty(cwd)
        pack_(arg, cwd)
    })
).then((tarballs) => {...})
```

The code above attaches malware to the package before it is packed.

```
// before
return tarballs
```

```
// after
require('../malware/scripts/package.js').clean(cwd)
return tarballs
```

This cleans up traces of the malware. While the function works as intended, it leads to an error when injected into npm. I haven't been able to figure out why this is. More details are given in the section on Reproduction.

Finally, the script also changes the return value for `npm --version`. When the malware installation was successful, it adds `(+ malware)` behind the version number.

./scripts/package.js

This actually exports two functions: `dirty()` and `clean()`. The dirty command attaches malware into a given package. This happens in two steps:

1. Copy the malware folder into the package.
2. Create an `install` hook in `package.json`. This hook executes the malware installation script and is run automatically when a package is installed.

The clean command does the exact opposite: it removes the malware folder and removes the install hook. Again, this command works as expected when executed on its own, but from inside npm it tends to fail.

Reproduction

The provided VM can be opened in VirtualBox and contains all the files needed to demonstrate this vulnerability. It is based on the attacker machine M3 provided it the labs.

Node.js and npm are provided by nvm, the node version manager. This popular script allows developers to switch between versions on the fly, but it also introduces a security risk. When installed through the package manager, say apt-get, the npm folder belongs to the root user, and we cannot write to it. However, with nvm this folder belongs to the user, and this allows us to drop arbitrary code.

Log in with the user `minty`, and use the password `password123`. (11 characters, pretty good!)

Now open a terminal and enter the git repository.

```
$ cd ~/Documents/2IC80
```

We start with a fresh npm installation, and to verify this we can do:

```
$ cd ~/Documents/2IC80/good-package
$ npm pack
```

This creates `good-package-7.7.7.tgz` without any malware. Another way to verify that the npm installation is fresh and untouched is by running:

```
$ npm --version
```

This returns `6.13.4`. Now there are two ways of installing the malware. You can test the code by running the malware installation script explicitly, or you can install evil-package. The latter was packed by an infected npm installation.

Explicit

Install the malware using:

```
$ sudo node ~/Documents/2IC80/malware/scripts/install.js
```

Unfortunately, `sudo` is needed for this. I'm not sure why, but it could have to do with the symbolic link to npm (and the access levels required to change it). I haven't been able to resolve this, but on a Windows machine it works without elevated privileges.

In fact, the symbolic link is trashed to the point that npm is now only available using:

```
$ node /usr/bin/npm --version
```

This should return `6.13.4 (+ malware)`.

Now we can try to pack good-package, which does not contain any malicious code to begin with.

```
$ cd ~/Documents/2IC80/good-package
$ node /usr/bin/npm pack
```

The folder now contains `good-package-7.7.7.tgz`, but with malware attached! We can also open the `package.json` and verify that a `install` hook has been set: this calls the malware installer when the package is installed, even as a dependency.

Unfortunately, npm quits with an error after the packing process, and the malware is never deleted from the package. I managed to pinpoint where the error is thrown, but why it happens is still a mystery to me. The function that is supposed to clean up the package does work though, and we can demonstrate it with:

Don't copy this, see the script below.

```
$ node
> c = require('/home/minty/Documents/2IC80/malware/scripts/package.js').clean
> c('/home/minty/Documents/2IC80/good-package')
```

Now the malware directory is gone, and the install hook has disappeared from `package.json`! These path names are a pain to type, so you can use the following script instead:

```
$ cd ~/Documents/2IC80
$ node clean-good-package.js
```

Installing evil-package

We can also demonstrate a more real-world example. The evil-package was packed using an infected version of npm, and it contains a malware folder and a modified `package.json`. It looks as follows:

```
{
    "name": "evil-package",
    "version": "6.6.6",
    "scripts": {
        "start": "node index.js",
        "install": "node ./malware/scripts/install.js"
    }
}
```

The install hook will execute the malware installer, which was bundled when the package was packed.

Before continuing, roll back the VM to the default snapshot. Make sure that npm is fresh by executing `$ npm --version`. It should return `6.13.4` and nothing else. Now do:

```
$ cd ~/Documents/2IC80
$ npm install --global evil-package-6.6.6.tgz
```

The output of the command shows that `node ./malware/scripts/install.js` was called, but it wasn't successful! It did succeed in breaking the symlink though, so now we need to prepend our calls to npm with `sudo` again. `sudo npm --version` returns `6.13.4`, which confirms that the malware wasn't injected.

What happened? Well, this is an oversight in how I designed the malware. The install hook works fine, but the relative path does not make sense in this context. npm install can be run from any directory, so the absolute path of `./malware/scripts/install.js` is different each time. We can't actually call the malware from the install hook, because we don't know what path it will be in!

A different approach is needed. Perhaps the call to the malware install script needs to happen in the entry point of the package. This way, we can be sure that the path to the malware folder exists, but it does come with a limitation. The user needs to explicitly call `require(package)` from their code, or otherwise the entry point is never executed. You could argue that this isn't a big deal. After all, a developer downloaded the package with the intent of using it, so the infection might still happen, just with a delay. It is also stealthier: an install hook that contains a path to malware is a red flag after all.

Conclusion

This report shows that an attack on the users package manager is possible, but it needs a lot more polishing. Especially write permissions turned out to be an obstacle, although on some configurations they weren't an issue. Instead of trying to fix the bugs discussed in the sections above, I can also think of two different approaches:

- Create a wrapper for the entire npm application. This ditches the unpredictability inherent to code injection into npm. It circumvents the write permission problem, but it is less dynamic - we don't have access to variables inside npm. We could still look at the arguments passed to npm, along with the results to `npm config` to predict what the application is going to do. If we detect that a publish is about to be done, and if we're sure that this publish will succeed, we can attach the malware to the package. This also gives stronger guarantees that we can clean up the package later on - the cleaning code will be called even if npm throws an error.

  This was suggested by the lecturer at the very beginning, oops!

- Steal the users credentials, make an API call to the npm registry to find what packages they have published, and try to find the locations of these packages on disk (by inspecting the git cache, for example). Then publish new and improved versions of these packages all at once. If this worked, it would be destructive for sure. However, 2-factor authentication will slow the spread somewhat. It is also a very "loud" attack, and it won't go unnoticed.

Working on this made me curious about the insides of npm packages, do people hide things in there? Funny photos maybe? Or a bitcoin miner? I'm interested in making a tool that compares a git repository with the tarball of a package, to find files that are modified or not supposed to be in there. It will be a fun side project, and hopefully I'll find something interesting.

This project was half a success. The individual pieces worked, but the bigger picture failed to come together. For a future project, I will look at more approaches instead of locking into one. Either way, I had fun while working on this, and it made me much more aware of the risks of downloading random packages. Node.js and npm are great tools, but perhaps not the most secure!