Dependency-based Attacks on Node.js

Extended Abstract

Brian Pfretzschner
Technical University of Darmstadt, Germany
brian.pfretzschner@stud.tu-darmstadt.de

Lotfi ben Othmane
Fraunhofer SIT, Germany
lotfi.ben.othmane@sit.fraunhofer.de

***Abstract*—Node.js heavily relies on shared variables. Their manipulation can cause service interruption, confidential data leakage, and service behavior change. Such attacks can be performed out of third-party libraries without detection by the service. Identification of such attacks requires analysis of both, application and libraries code.**

I. Introduction

JavaScript is a popular programming language that is dynamically interpreted and has a simple syntax. It supports shared variables and function redefinition–i.e., monkey-patching. Node.js allows the use of JavaScript for server applications and supports third-party dependencies, which are linked dynamical. There is extensive use of these dependencies. For instance, there are more than 300.000 packages available in the Node Package Manager (NPM) public repository. That is more than twice the number of packages available in Maven.[1] Such modules can be easily shared, installed, and updated using the package manager NPM.

The use of Node.js in uniform and convenient cloud platforms raises security risks. Applications usually have access to potentially sensitive data stored in the cloud platform. This data can be leaked and manipulated using *Dependency-based attacks*: attacks that require implanting a malicious dependency into an application's dependency tree. Typical Node.js applications have lots of dependencies, thus, a large number of packages qualifies for the attack.

Vulnerabilities originating from third-party dependencies have been extensively studied. Dependency-based attacks are different. The malicious dependency itself is actively attacking the application when loaded. The question is: What can malicious code located in a dependency do to an application?

II. Attacks

We identified several attacks. We give here an overview of two. The precondition for the attacks is that the host application loads the malicious dependency, e.g., the attacker changes the libraries at run-time by benefiting from multitenancy.

The first flaw is due to the use of global variables in JavaScript, which could be accessed/manipulated from anywhere in the application without restriction. In the case of one cloud provider, security credentials are passed to applications as environment variables, which are mapped to global variables in Node.js. The following code snippet demonstrates example of use:

[1]http://www.modulecounts.com/

```
function leakData(data){/* leak given data */}
leakData(process.env);
```

This attack is very easy and efficient, it does not alter or interact with the host application but still leaks the credentials without the application noticing. The attacker could use the credentials to access the cloud service remotely.

The second flaw is related to Node.js's dependency loading system. Using a more sophisticated approach, the attacker can even break applications running in trusted environments. For instance, one cloud provider implements a piece of Node.js code that can only be triggered by trusted Web services. The code uses *async*, which is a popular library to process data asynchronously. At some point, the application passes the data that are being processed to `async.map`. Hence, this function can be redefined to get access to the sensitive data:

```
var async = require('async');
var map = async.map;
async.map = function (events) {
  leakData(events);
  return map.apply(this, arguments); };
```

In line 1, the *async* library is loaded. In line 2, the *map* function gets copied to a local variable for later use. The *map* function is overwritten in line 3 by a custom implementation. The new function leaks the provided data and calls the original implementation from line 2. Using this pattern, a large portion of the code executed in modern Node.js applications can be completely altered by just a single malicious dependency. Control flow can be altered, confidential data can be leaked and manipulated or back doors can be opened by weakening the security mechanisms of applications. The host application can virtually not detect that such modifications were performed.

III. Solutions

The immediate solution is code review of all dependencies (including development dependencies!), which is not practical. The flaw in the Node.js's dependency loading system might be fixable but requires some essential changes to the Node.js API. As an alternate solution, we extended the open source code analysis framework WALA[2] to support Node.js and implemented code analysis to detect these attacks.

Acknowledgments

We thank Monika Illgner-Kurz (IBM DE) and Julian Dolby (IBM US) for assistance and directions.

[2]https://github.com/wala/WALA