Detecting Suspicious Package Updates

Kalil Garrett
Georgia State University

Gabriel Ferreira, Limin Jia, Joshua Sunshine, Christian Kästner
Carnegie Mellon University

***Abstract*—With an increased level of automation provided by package managers, which sometimes allow updates to be installed automatically, malicious package updates are becoming a real threat in software ecosystems. To address this issue, we propose an approach based on anomaly detection, to identify suspicious updates based on security-relevant features that attackers could use in an attack. We evaluate our approach in the context of Node.js/npm ecosystem, to show its feasibility in terms of reduced review effort and the correct identification of a confirmed malicious update attack. Although we do not expect it to be a complete solution in isolation, we believe it is an important security building block for software ecosystems.**

***Index Terms*—malicious update attacks, anomaly detection, clustering, Node.js, npm**

I. Introduction

Malicious package updates are becoming a real threat in the *Node.js/npm* ecosystem [1] [2]. According to Adam Baldwin, npm's Head of Security, they have found at least 6 packages with simple attempts to open reverse shells on npm until 2016. These attacks, however, have received little attention from both open source and research communities. Recently, in July 2018, an attacker stole the *npm* credentials from a contributor of the *eslint-scope* package and published a malicious version of the package. The malicious version contained code that would steal the *npm* credentials of all user machines that run applications that depend directly and indirectly on the *eslint-scope* package [2]. Fortunately, an user quickly identified the attack and reported it to the community which remedied the attack after a few hours. Interestingly, the malicious version of the package contained certain suspicious characteristics that had not been observed in previous versions of the package. In Figure 1, we can observe that the update resulted in a new hookup script entry that spawn a new instance of the Node.js runtime and the use of *eval* and other potentially dangerous libraries, such as *fs* and *https*, being used in the attack. *eval* and the mentioned libraries have not been used by the package until that version. In this work, we build an anomaly detection approach for suspicious updates based on such characteristics.

In current practice, developers often rely on many untrusted packages from several third-parties to speed up their development time, prioritizing functionality and popularity over security, often trusting the reputation of package authors and trusting that the community will review updates and quickly find issues as they arise [3]. Unfortunately, these practices are not sufficient to address the security challenges faced by the Node.js/npm ecosystem:

```
...
"scripts": {
  "postinstall": "node ./lib/build.js",
},
...
```

(a) *postinstall* hookup script entry added int the malicious version.

```
try {
  var https = require("https");
  https.get({
    hostname: "pastebin.com",
    path: "/pathControlledByAttacker",
    headers: { ... }
  }, r => {
    r.on("data", c => {
    eval(c);
  });
  r.on("error", () => {});
}).on("error", () => {});
} catch(e) {}
```

(b) Malicious file (*./lib/build.js*) added by the attacker: it downloads and evaluates an script published on *pastebin.com* that is also controlled by the attacker.

Figure 1: **eslint-scope@3.7.2** attack.

- the community favors a model of small packages and use third-party packages even for simple tasks such as string manipulation (**increasing the opportunities for attacks**),
- updates are frequent (**increasing the opportunities for attacks**),
- updates are automatically installed (**facilitating the successful execution of attacks**), and
- through the use of native libraries, packages have access to powerful OS-level capabilities (**increasing the impact that attacks can cause**)

Currently, npm has over 700,000 published packages, with each package having, on average, a total of 90 direct and indirect dependencies. As the number of packages and the frequency of updates increase in the ecosystem, it becomes impractical for the community to review all package updates that are released on *npm*.

This year alone, there have been approximately 4,900 updates per week (29 per hour), making it unrealistic to assume that the community can manually review all of them. Our approach based on anomaly detection notifies developers about suspicious dependencies updates, complementing other community practices such as using automated tools to scan the package for known vulnerabilities [4] and lightweight analysis tools (e.g., linters) to search for common issues.

We propose an approach based on anomaly detection, an automated machine learning technique used to identify abnormal data being commonly used in other domains to detect credit card fraud, network intrusion, and other applications

[5]. We conjecture this technique can be successfully applied to detecting suspicious updates in *npm*.

We present a preliminary evaluation for our approach, aiming to demonstrate its feasibility. We analyze the review effort reduction that developers could get from it and test our model against a confirmed malicious update attack. Our results show that even a simple and fairly inexpensive automated approach can detect a confirmed attack and reduce the review effort by 89 percent from 701 updates per day to 77 suspicious updates only, even considering the worst case scenario where all notifications we raise are false positives, and has great potential to detect suspicious updates in real-time.

We do not expect this to be a complete solution to malicious updates in isolation, and acknowledge the limitations of our technical approach and preliminary evaluation results. However, we believe this is an important security building block for software ecosystems that need solutions for the increasing number of issues created with the current automated (and often unsecure) dependency management mechanisms.

In this paper, we make the following contributions:

- we raise awareness about malicious updates; an important (and so far ignored) issue in software ecosystems,
- we show that anomaly detection can be used in a new context (i.e., package updates) to support the identification of suspicious updates,
- we identify relevant features to be used by anomaly detection techniques on package updates, and
- we provide a preliminary evaluation showing the feasibility of our approach based on anomaly detection.

II. Background and Related work

Node.js, a runtime engine for JavaScript, has gained popularity in recent years because it allows JavaScript applications to be executed outside a browser. The non-blocking behavior of JavaScript makes it attractive to the development of robust server-side applications, making *Node.js* an incredibly powerful platform for developers. With over 700,000 packages available for developers to build their applications, *npm*, the package manager for *Node.js*), is currently the largest package manager[1]. Both *Node.js* and *npm* contribute to being the largest and most active open source ecosystem with an average of 4 contributors per package for over 700,000 packages.

There is a natural friction on the decision about updating packages or not. The inherent costs attached to updates, such as modify client code due to breaking changes, re-test your application, and review updates, causes developers to not update their dependencies, even after serious vulnerabilities are reported and patches for them are made available. There are several works that discuss the security of package managers [6] [7], but also how developers from and outside the *Node.js*/*npm* community react to updates [8] [9] [10] [11].

To reduce costs with updates, developers rely on automation, sometimes allowing updates to be installed automatically. Unfortunately, increased automation comes at a cost: applications (and its dependencies) are more susceptible to malicious update attacks. Our paper aims to raise awareness about this issue and start a discussion about update attacks.

We use anomaly detection to detect suspicious package updates, extending its use beyond known cases such as intrusion detection, fraud detection, industrial damage detection, image processing, commit reviewing prioritization, and traffic monitoring [5] [12]. Anomaly detection can be implemented with several techniques, including machine learning for classification and clustering as well as various statistical approaches, as explained in detail by Chandola et al. in their comprehensive overview of the anomaly detection [5].

Given that most of the updates on *npm* are not malicious, we use an unsupervised learning strategy based on clustering. Clustering techniques have long been studied and applied to many partitioning problems [13]. The goal of a clustering techniques is to group similar objects in a way that each cluster have objects that are more similar among themselves than when compared to other objects in other clusters. We create clusters for normal package updates data across multiple packages and detect anomalies by checking the distance of new data points to the center of each cluster. If the distance of a new data point (i.e., new package update) is greater than the cluster threshold, we tag the new data point as suspicious.

III. Anomaly Detection Approach

We propose an automated approach based on anomaly detection that can detect suspicious updates on *npm*. Our solution builds upon the assumption that normal updates occur more frequently than suspicious ones, and this normal behavior can be characterized to create a normal behavior model. New updates can then be classified as suspicious or non-suspicious using this assumed normal behavior model. Suspicious updates can then be reviewed for malicious intent.

A. Features

To characterize package updates, we extract features from packages' metadata (i.e. package.json) and from packages' source code. For each package and version, we collect features that characterize the version of the package.

We conjecture that a malicious update attacks would enhance a packages' capabilities to exploit users systems. In our process to discover features, we focused on features that could be used to attack a given packages. Adding more and more features to the model can actually be harmful (e.g., can add noise to the model or affect the performance of the detection model). Therefore, it is important to identify important features because clustering techniques are still not sufficiently fast for datasets with many features [14].

Table I shows the selected features. *Node.js* provides applications (and its dependencies) unlimited access to powerful native libraries. These libraries can be used to exploit a package users computer system, since they provide access to OS-level functionality. For instance, the selected libraries (*http*, *http2*, *https*, *net*, *fs*, *child_process*) can enable applications

[1]www.npmjs.com

Features	Description
http, http2, https	Send/receive HTTP requests.
net	Open/listen/write to sockets.
fs	Create/read/write to file system.
child_process	Spawn child processes in the OS.
eval, *Function*	Evaluate code (strings) at runtime.
new JS files added	Add new code (directly).
new package dependencies added	Add new code (indirectly).
new hookup script entries added	Run arbitrary user commands.

Table I: Description of the features selected for anomaly detection and a short description of how each can be used by attackers.

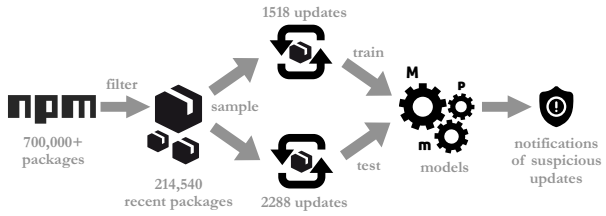

Figure 2: Overview of our data collection and anomaly detection model development process.

to access the network, the file system, and operating system processes, respectively. In addition to the *Node.js* libraries, Javascripts *eval* and Function method allow malicious code to be evaluated at runtime. While evaluating new code at runtime is not inherently malicious, it allows developers to evade our anomaly detection approach, since the features used in code evaluated at runtime would not be statically detectable.

Hence, we extract the use and emergence of the selected, exploitable libraries, *eval*, and Function by statically analyzing all package's JavaScript source files. Additionally, we identify if new files, new dependencies, and new hookup script entries are present in the package directory and manifest file, respectively. We use a binary representation of each feature, since we want to examine if a feature is present or if a change in the feature has happened from one version to another.

B. Detection Model

In order to build an anomaly detection model and establish a normal behavior model, we need to collect data from packages and package updates. In February 2018, we collected meta information for all the 703,457 packages available on *npm*, which includes information of packages, such as name, dependencies, contributors, and versions (which includes links to the zipped source code files). We build our normal behavior model based on the history of several packages, instead of tailoring one customized model for each package. Figure 2 shows an overview of our data collection and modeling process.

Package Exclusion criteria. From the entire population of packages on *npm*, we only examine packages with two or more versions, so there is update history for each package. In addition, we only collect packages that have had recent activity, excluding packages that have not been updated within a year of the collection date. After excluding the packages that do not comply with our criteria, we have a total of 214,540 remaining packages.

Clustering After collecting the features data from packages and packages updates, we need to cluster them. We use the k-means technique to cluster our data [13], since it fits our technical problem: there are few, known attacks and updates are not labeled as suspicious or normal in the *npm* ecosystem. Unsupervised learning does not require a labeled dataset, and related work has shown that clustering is an effective unsupervised learning technique [5]. It accommodates our need to detect suspicious updates without having a ground truth for a normal or suspicious update. As common practice with unsupervised learning, we need to define the number of clusters in our model. To define it, we use the *elbow method*, which uses an heuristic to identify the optimal number of clusters before running the k-means algorithm. We examine the second derivative of the elbow curve and identify the earliest point where the this derivative approaches zero. This point indicates the optimal number of clusters where the error begins to change at a constant rate. Each cluster has its own centroid. The best possible centroids, data points that represent the center of each cluster, are determined, and the clusters are then formed. We create an outlier threshold based on the distance from the centroid to the furthest cluster point, so new data points are anomalies if their distance to their assigned centroid is greater than the threshold.

In this paper, we cluster 1,518 randomly selected package updates on *npm* to establish normal behavior. We split package update data by the type of change signaled by the new version number published by a contributor and create distinct detection models for patch, minor, and major updates. Each model is assigned its optimal number of clusters: 10, 1, and 1 for patch, minor, and major releases respectively.

Implementation We utilize the *scikit-learn* python library to train and analyze our model. Before creating the model, we need to take several steps to process our data:

1) the data needs to be normalized so features with different scales are comparable.
2) the number of clusters must be defined beforehand.

We create distinct models for patch, minor, and major updates by clustering the training data with the optimal number of clusters. Every point in the training dataset is included in a cluster. An assumption is made that normal behavior occurs in compact, large clusters, so loose and small clusters are investigated to ensure they represent normal behavior.

To test the model, we use package updates from the testing dataset and assign each of them to an existing cluster. Since we assume that normal updates are the norm in the training dataset, an update is considered suspicious if it is not contained within the boundary of the cluster. Thus, the distance from the cluster center to the furthest data point is used as the threshold. If a new update from the testing dataset is assigned to a cluster

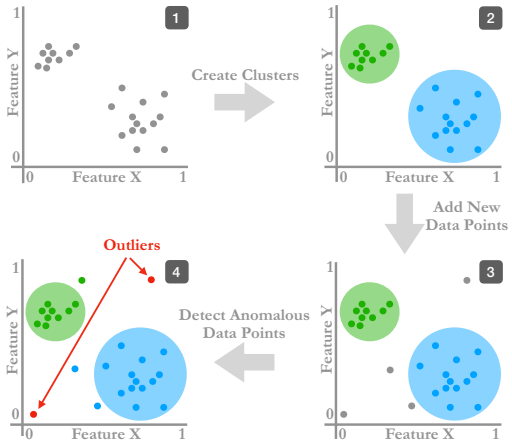

Figure 3: Our approach based on anomaly detection with unsupervised clustering.

and is further than the furthest data point, it will be considered suspicious as shown in Figure 3.

IV. Preliminary Evaluation

We now explain how we test our detection model against recent updates and against a confirmed malicious update.

A. Review Effort Reduction

To provide an estimate about precision, we have tested our model against 2,288 recent package updates. For that we only tested package updates that occurred after our training dataset collection date. We use the ratio between number of updates per week and the number of suspicious updates alarms our approach raises as a proxy for precision. The intuition behind it is to estimate the reduction in manual review effort to developers after the prioritization of suspicious updates.

Result. Our model reports 539 suspicious updates per week (almost 3 per hour). Even considering the worst case scenario where all updates are not malicious, if one trust these results, our approach could reducing the review effort by 89 percent.

B. Assessing Confirmed Malicious Update

In addition to the evaluation of recent package updates, we also examined whether our model could detect the *eslint-scope* attack. We only have this case being evaluated because the attacks using automatic updates are silently addressed. We however have anecdotal evidence that their occurrence is not rare (see Section I). To reduce biases in our evaluation, we checked that the updates from the *eslint-scope* have not been added to our training dataset. After we created detection models for patch, minor, and major updates for 1,518 randomly selected package updates on *npm* to establish normal behavior, we then tested the six *eslint-scope* updates, including the attacked version, against our model.

Result. Our results show that the patch model labeled the malicious version of *eslint-scope* as suspicious and labeled the other patch releases as normal.

V. Discussion and Conclusion

We showed in an preliminary evaluation that our approach based on anomaly detection can be useful to the *Node.js/npm* ecosystem, but more investigation is necessary. From both technical and research perspectives, there is still a need for:

- evaluate our model on the entire *npm* ecosystem, **to show usefulness on a large scale**,
- examine alternative designs for our detection model, such as examining per project, per developer, or by package size, **to improve precision**,
- create an explanation mechanism for suspicion, **to provide shortcuts for developers to make decisions about suspicious updates**,
- create a dashboard with most suspicious packages per day, **to raise the awareness about potentially malicious updates**.
- propose automated and social approaches, **to support developers in addressing issues with automated updates**,
- investigate design alternatives for package managers, aiming at **creating a secure package manager**.

VI. Acknowledgements

This work was supported through NSF Grant 1717022.

References

[1] S. Saccone, "npm hydra worm disclosure," Google, Tech. Rep., 2016.
[2] ESLint. (2018) Postmortem for Malicious Packages Published on July 12th. [Online]. Available: https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes
[3] E. S. Raymond, *The Cathedral and The Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 2001.
[4] Greenkeeper. (2018) Automated Dependency Management. [Online]. Available: https://greenkeeper.io/
[5] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, pp. 15:1–15:58, 2009.
[6] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A Look in the Mirror: Attacks on Package Managers," in *Proc. Conf. on Computer and Communications Security (CCS)*, 2008, pp. 565–574.
[7] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable Key Compromise in Software Update Systems," in *Proc. Conf. on Computer and Communications Security (CCS)*, 2010, pp. 61–72.
[8] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android," in *Proc. Conf. on Computer and Communications Security (CCS)*, 2017, pp. 2187–2200.
[9] S. Mirhosseini and C. Parnin, "Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies?" in *Proc. Int'l Conf. Automated Software Engineering (ASE)*, 2017, pp. 84–94.
[10] R. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do Developers Update Their Library Dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.
[11] A. Decan, T. Mens, and M. Claes, "An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 2–12.
[12] R. Goyal, G. Ferreira, C. K"astner, and J. Herbsleb, "Identifying unusual commits on GitHub," *Journal of Software: Evolution and Process*, vol. 30, no. 1, 2017.
[13] P. Flach, *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, 2012.
[14] L. Portnoy, E. Eskin, and S. Stolfo, "Intrusion Detection with Unlabeled Data Using Clustering," in *Proc. CSS Workshop on Data Mining Applied to Security (DMSA*, 2001, pp. 5–8.