Detecting Vulnerabilities in NPM: A Scalable Model

Samuel Steinberg[1], Jonathan Pace[2], Tapajit Dey[3]and Jeffrey Dong[4]

Abstract— **This document details the findings of creating a model for detecting malicious NPM packages. The aim of this application is to create a reliable framework to isolate packages of interest and classify suspicious patterns based off their change in weekly downloads. The components of this paper detail the process and findings for detecting vulnerable and potentially harmful NPM packages in addition to offering insight on how this model might be scaled by future developers.**

I. INTRODUCTION AND MOTIVATION

The Node Package Manager, or NPM, is an open-source default package manager for the Node.js run-time environment. It is primarily used to install and distribute code. There are, however, bad actors that seek to distribute malicious packages that can harm users who un-wittingly download them. In our research, we sought to develop an algorithm to find patterns between download numbers on a week-to-week basis for any given NPM package that our model flagged as suspicious and compared the time frame with events taking place in their dependent packages. We then sought to analyze these events and classify them and display the most common causes of potential harmful packages.

II. ARCHITECTURE AND METHODS

A. Architecture

Our application is structured to be scaled up to include any number of packages. First, we read from NPMPackages.json.gz, which contains the version history for all NPM packages into a generator object. Using this data, we build a collection of package download data in our Mongo database. Each item in the collection has a 'name' element and a 'download' element which contains each packages weekly download data. We then clean the data, removing all weeks which contained downtime in the NPM website, ensuring that we do not interpret these falls in downloads as a suspicious event. After this begin suspicious rise/fall detection by building a series of lists which we later use to compare, iterating through our database of packages. Next, we build the packages dependent list with its version history – including its release dates for different versions. This is used to build our add/drop date lists. These lists are then used when we compare the rise/fall lists to the add/drop lists to locate a potentially suspicious correlation.

B. Methods

1) USING A DATABASE AND ORGANIZATION: We created a Mongo database for weekly download data per each individual package. For each package, an empty download list is created. The NPMjs API is called to retrieve one week's worth of download data, which we then insert into the download list. After the iteration is completed for each week in our date range, an association is built between the name of a package and its associated weekly download data. This association is what we insert into the database collection.

2) RISE AND FALL LISTS: After extracting the package and its data from the database, read the download data and build a dictionary for both rises and falls. We then iterate through the download data, and filter out weeks with less than 300 downloads and check that both the current week and previous week do not have zero downloads. Then, for both lists a percentage difference is taken between the current week and previous week. For a rise, if a percentage difference is greater than 30% and the current week's downloads is greater than 30% of the package's highest download week, it is categorized as a suspiciously high rise and is added to the rise list. For a fall, if a percentage difference is less than -30% and if the previous week's downloads is at least 30% of the package's highest download week, it is categorized as a suspicious fall and is added to the fall list.

3) ADD AND DROP LISTS: After building the dependent list with version history, we iterate through it. For each package, we build two lists: One of all versions with dependents that the package did not have the previous week (an add), and another built with all versions without dependents that the package had the previous week (a drop).

4) LIST COMPARISONS TO DETERMINE EVENTS: After building the lists of rises, falls, adds, and drops, we compare the rise list to the add list, and the fall list to the drop list. If there is an add of a popular package in a similar date range to a date in the rise list, we can conclude that the adding of that package caused the rise. The same can be said for comparing the fall list to the drop list.

III. PATTERNS AND POSSIBLE EVENT EXPLANATIONS

A. PACKAGE ADD OR DROP

If a package was added as a dependency to a popular project then whenever that project is downloaded, so is the package. This will lead to a rise in downloads for the

*This work was not supported by any organization

[1]S. Steinberg is an undergraduate student of Computer Science at the Tickle College of Engineering, University of Tennessee, Knoxville, TN 37996 ssteinb2@vols.utk.edu

[2]J. Pace is an undergraduate student of Computer Engineering at the Tickle College of Engineering, University of Tennessee, Knoxville, TN 37996 jpace7@vols.utk.edu

[3]T. Dey is a graduate student in the Tickle College of Engineering, University of Tennessee, Knoxville, TN 37996 tdey2@vols.utk.edu

[4]J. Dong is an undergraduate student of Computer Science at the Tickle College of Engineering, University of Tennessee, Knoxville, TN 37996 jdong6@vols.utk.edu

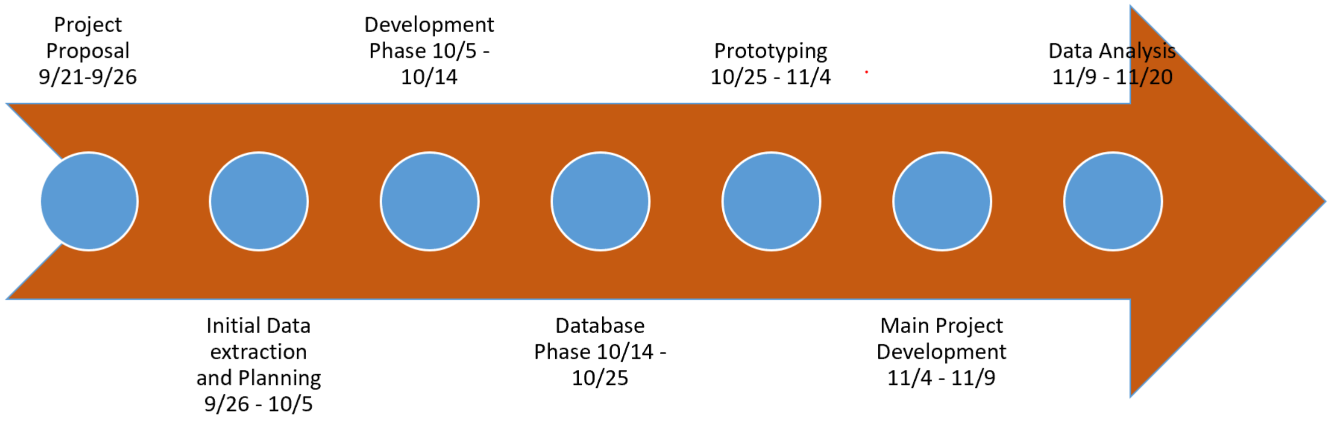

Project Proposal 9/21-9/26
Initial Data extraction and Planning 9/26 - 10/5
Development Phase 10/5 - 10/14
Database Phase 10/14 - 10/25
Prototyping 10/25 - 11/4
Main Project Development 11/4 - 11/9
Data Analysis 11/9 - 11/20

Fig. 1.

package. When a package is dropped as a dependency, then the downloads will drop by the same amount as before it was added. This will lead to a fall. This category is considered the most suspicious, next the 'Unexplained' category. An example can be seen in the package exec-file-sync, which was added as a dependency to the popular package os-locale on 8/14/2015. This led to an unprecedented rise in downloads in exec-file-sync, but after it was removed on 9/3/2015 its weekly downloads crashed down to its previous numbers.

B. SIMILAR DOWNLOAD DELTA IN DEPENDENT PACKAGE

If a popular project experiences a significant rise or fall in downloads then so does its dependencies. This means that if we observe a similar rise or fall in a similar period of time between a package and dependent, we can conclude that the change in package downloads was caused by a change in dependent downloads. This can be seen in the package htmlnano, where its dependent parcel-bundler experienced a rise from 4000 to 293,000 weekly downloads and htmlnano experienced a rise from 4000 to 292,000. They then both fell to approximately the same download numbers as before the rise. These in-sync rises and falls also took place at the same approximate time.

C. NEW VERSION RELEASE

There is also the potential of a package experiencing a rise or fall due to a new version release. A rise could occur if users using a particular package want to download the new version (of the same package). Once current users download the new version, downloads will fall back down to a normal level. This can be seen in the package doofinder, where a new version was released 1/26/2018. After the release, downloads experienced a sharp rise for the following week.

D. HOLIDAYS

Since most people take holidays off, it is expected that the weekly downloads will experience a fall during this period. This then leads to a rise the following week when people go back to work and developing software.

E. UNEXPLAINED EVENTS

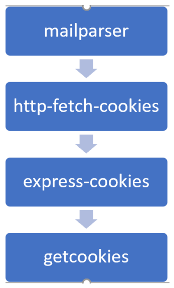
There are, however, events that cannot be explained by the patterns we discovered. These include events where there was not a new version release, there was no significant change in dependent downloads, there was no add/drop of packages of significance, and did not occur over dates worth mentioning. We categorize these as unexplained events, and are events that we would like to flag to take a closer look at for being the most likely candidates for potential vulnerabilities.

IV. FINDINGS

We detected 71 rises and falls combined in our list of packages. Of these 71, six were caused by add/drops, two were caused by holidays, thirty-six were attributed to a similar delta in downloads for a dependent package, three due to a new version release, and twenty-four were unexplained. We graph our results in Fig 2.

V. EXAMPLE OF A NON-TRIVIAL PACKAGE

An example of a proven malicious package attached to a popular one would be: getcookies. The package getcookies was added as a dependency to express-cookies, which was a dependency of http-fetch-cookies, which was a dependency of the popular package mailparser.



Similar to a hacker taking over a chain of computers in order to cover up their tracks, getcookies performed the same type of malicious behavior in NPM. How would our project detect this? First, once the packages are loaded in, all of mailparser's dependencies would be added into our dependency database. Our program would detect a rise in getcookies, which would be paired with a similar add/drop

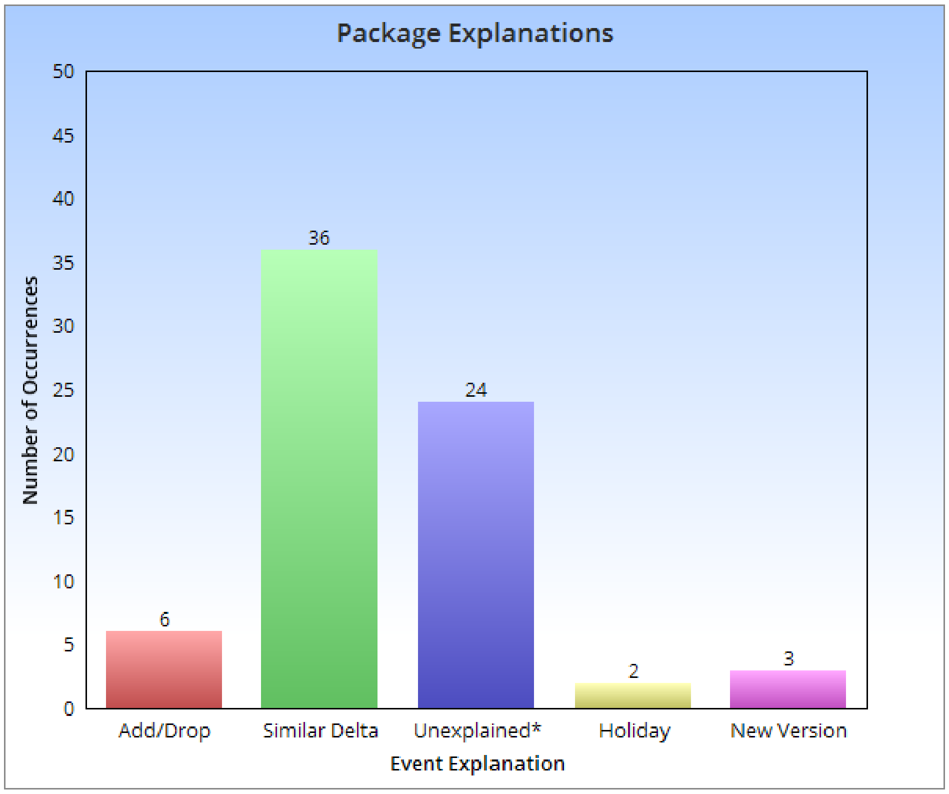


Fig. 2. *pino-toke is attributed to twelve occurrences

date on express-cookies. This would be categorized as an add/drop event. This is one of the most suspicious categories, which would flag getcookies as a package with a potential vulnerability. The subsequent manual checking could have discovered the malicious code in getcookies.

VI. DATA ANALYSIS AND EVENT DETECTION

We started analyzing our data by retrieving packages one-by-one from the database, which contains its weekly download data. We then compare each weeks downloads to the previous week by taking the percentage difference. If the percentage change is greater than 30% and the absolute value was at least 30% of the highest ever number of downloads for the package: We have a significant rise event. If the percentage change is less than -30% and the absolute value of the previous weeks downloads is at least 30% of the highest ever downloads, then we have a significant fall event. To determine a correlation a rise/fall and an add/drop of dependents, we iterate through each rise/fall in the list and compare that to the dates of all add/drops of dependents of that package. When this occurs within a week of one another, we generate the package correlation and the time stamp. For example the package exec-file-sync will generate:
Rise of exec-file-sync on 2015-08-20 which coresponds to add of os-locale on 2015-08-14 08:00:04 with a time difference of 5 days, 15:59:56
Fall of exec-file-sync on 2015-09-03 which cooresponds to removal of os-locale on 2015-09-03 10:31:39 with a time difference of -1 day, 13:28:21

VII. PROJECT MANAGEMENT

A. WORK FLOW AND TIMELINE

1) GETTING STARTED AND PROPOSAL: To begin the project, each of our group members began by gathering information about the NPM site, the relationship between packages and their dependents/dependencies, and analyzed examples of packages previously found to contain malicious contents. Gathering a strong knowledge of our topic was essential to our group and our success moving forward. We used this information to develop our proposal and to begin reading in packages to our application.

2) PARTITION OF WORK AND GROUP MEMBER RESPONSIBILITIES: Before actually beginning our project, work was divided between group members. Samuel Steinberg was assigned the task of creating the mathematical functions and data structures used in partitioning the data and finding patterns between suspicious packages and their dependents. He was also assigned responsibility for documentation. Jonathan Pace was responsible for reading in the packages and helping to develop the algorithms used in determining the cause of suspicious events. He was responsible for handling our database: He set it up in Mongo, and was responsible for inserting and extracting the data efficiently. He was also responsible for creating our add/drop and rise/fall correlation function. Tapajit Dey was responsible for overseeing the project and assisting in reading in data. Jeffrey Dong assisted in data analysis. This partition of work ensured that each member knew their role and became proficient in that regard as well. The combination of our roles and coordination with one another would come together to complete our algorithm and project.

3) TIMELINE:

a) INITIAL DATA EXTRACTION AND PLANNING: The initial extraction of our data consisted of opening and reading from a JSON file compressed into a .gz file. This would read in all of the objects/packages we would be analyzing. Once this was parsed, we now had access to all packages on NPM. This was completed in a little over a week, consistent with our goals for the initial stage.

b) DEVELOPMENT PHASE: The development phase refers to the labeling and parsing of our data. We developed a function that swept through each listed package and stored their version number, time stamp, release date, name, and dependencies. We designed this stage to form the backbone of our database insertion and was designed for easy labeling an insertion. In addition, this is where the initial mathematical equations were developed and data structures were implemented. The tweaking and changing of structure would go on for the duration of the database and prototyping phase as well. This took around nine days to complete.

c) DATABASE PHASE: The database phase was where we inserted into the Mongo database. We inserted in a table where all the information for each package and its dependents were organized and labeled accordingly. The information inserted included consisted of all the information extracted from the Development Phase. Two distinct database entries were maintained: one consisting of all suspicious packages and another made up of all of their dependent packages. Additionally, this stage is where a major overhaul of our mathematical functions and data structures took place. They had to be re-designed to handle larger amounts of data, adapt to the change in syntax for the database, and be refined for further accuracy. This stage took nearly two weeks to complete.

d) PROTOTYPING: The prototyping stage was our most work-intensive and thought-provoking. This stage focused mainly on our math and methods of detecting a dependent of interest. All group members would engage in changing our data structures, logic, and expectations for determining the reasons for a suspicious event taking place. While we first focused on the downloads of dependents and developing a logarithmic function to determine rises and falls in dependents, this proved to give too many false positives on steps and missed too many cases for falls. Hence, we developed an improved algorithm analyzing similarities of add/drop dates in dependent packages and dates of rises/falls in the suspicious package. This proved to be the most efficient means of detection. This stage took ten days to complete.

e) MAIN PROJECT DEVELOPMENT: The development stage for our main project took the least amount of time, tallying in at around five days. Due to designing our application and algorithms with respect to the future build-up, this stage only consisted of scaling up to allow more packages to be loaded and analyzed.

f) DATA ANALYSIS: Data analysis is the core of our project: After loading all the data in for each package and running our application, we could then see the most likely causes of a package being labeled as suspicious and gather the results. In addition to our generated results, manual checking was also performed here to ensure data reliability.

B. COMMUNICATION

An integral part of our project was communication and maintaining the steady work flow defined in Section A. Communication took place in person (during assigned class-time, along with scheduled meetings outside of class time) and electronically over user-friendly sites such as Slack and GitHub. Initially, most of our work was completed in person. This was by design, as to successfully and efficiently complete this project knowing one-another and starting on the same page was key. As the project went on, communication over Slack and setting goals over GitHub became of increasing importance. This allowed us to know what we had to complete, and when we had to complete it. Our efficient and high-level communication was essential to developing our algorithm and completing our project on time and ahead of schedule.

VIII. OVERALL EXPERIENCE

The overall experience in creating our application was extremely worthwhile. Each group member gained valuable experience throughout the duration of the project.

A. WORKING IN A TEAM SETTING

Working as a cohesive team is a very valuable and more importantly essential skill in the workplace. Working and communicating as a team for the duration of our two-month long project allowed us to hone our skills and develop them. Our project contained many moving parts, and the efficiency working alongside one another got improved exponentially as the project developed.

B. MODERN SOFTWARE DEVELOPMENT EXPERIENCE

Access to sites such as GitHub and merging individual segments of code was new to each member of our group, and challenging at first. Since industry standard utilizes heavily individualizes roles which must come together to push a product in the end, this proved to be a very important experience. As our project progressed, each group member became more and more ”professional” when it came to completing individual work and pushing it to GitHub. This made it seem as though our project was more of a product.

C. RECURRING ISSUES

Though minor issues were promptly fixed, we did face two major recurring issues. These were the constant re-writing of certain code segments, and syntax adjustments. When it came to developing our algorithm for flagging rises/falls in packages, for example, we had to completely re-write it for increased accuracy. This was due to at first using a logarithmic function to compare downloads between a given week and either its following or previous week. A prime instance of this would be when a package incurs a sharp rise, and a sudden fall that would fall halfway back down to the original download number. A logarithmic function would detect the rise, but would not flag a fall due to it not meeting our standard for a fall (due to using this function). We fixed this by setting a fixed percentage in download difference, 30%, a number which we hope future developers can use and base their standards around. Syntax issues were recurring mainly because none of our three members had ever used the language before. This was both rewarding as a learning experience along with it being a recurring issue that often took significant time to figure out. This was especially challenging in our prototyping phase when we needed to compare extracted data in nested structures.

D. *CHANGING METHODS AS PROJECT PROGRESSED*

When writing our algorithms and planning for the future, there were times when our methods and target data changed. This occurred in regards to what would become our Mongo database and algorithm for determining the cause of rises/falls in a suspicious package. Before use of a Mongo database, we had planned to store all of our data into a .csv file. We decided to use the Mongo because .csv files do not distinguish between text and numeric values, and with our nested data structures extraction would be exponentially more difficult. Mongo allows much easier extraction and labeling is trivial. When it came to determining the cause of rises and falls, we had to scrape an entire function that we had hypothesized would give us the most accurate results. This involved keying rises and falls of the package with those of its dependents. We used this until analyzing manually whether or not we were accurate in our math, and realized that it was only one of a multitude of reasons for a suspicious event and an uncommon one. This was refined and eventually led us to categorize our distinct patterns.

IX. LIMITATIONS

A. *MULTIPLE RISE OR FALL WEEKS*

In our application, if there is a significant rise or fall encompassing more than one week, it is flagged multiple times.

B. *EARLY RISE*

If there is an early spike in downloads in a package's history, and then the package gains popularity to the extent that it is more popular than it was during the spike: We do not detect the early rise.

C. *MULTIPLE RISES OF VARYING SIZES*

If there are two spikes, but one is significantly larger than the other only the larger one is detected. Specifically, if the smaller spike is less than 30% of the size of the larger spike it will not be detected.

D. *DOWNTIME RISES*

Since we removed all weeks with server downtime to avoid detecting false falls, there is a chance that there can be a false negative for a rise in those weeks.

X. FUTURE WORK AND CONCLUSIONS

It is our hope that our application can be used either as a basis or a guide for future developers. We have laid a solid framework that is purposed to be scaled up and built upon. This was designed intentionally so that future developers could modify our algorithms to reach deeper levels without much trouble. Specifically, embedding machine learning algorithms to find anomalies would be ideal using our work. Machine learning could be used to simplify our algorithm and make it more efficiently and accurately recognize a suspicious package or pattern. In other words, it would greatly enhance the ability to find the cause of an event. Additionally, a recursive search could be implemented and built upon to look deeper into possible causes and dependent lists. This search would traverse a packages dependents like a search tree: First, the packages dependents would be traversed to find an anomaly. If found, then the dependents packages would be traversed as well, and this would continue until the root cause for the suspicious package is found and the tree cannot extend further down. Embedding algorithms such as a machine learning one or a deep recursive search would greatly enhance our application, and we hope that it can be used as a model by future developers in their work.

XI. RELATED WORKS AND REFERENCES

- Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 385395.
- Tegawend F Bissyand, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Rveillere. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual.* IEEE, 303312.
- Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. *In Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on.* IEEE, 334344.
- Aparna A Chhajed and Susan H Xu. 2005. Software focused supply chains:Challenges and issues. In *Industrial Informatics, 2005. INDIN05. 2005 3rd IEEE International Conference on.* IEEE, 172175.
- Diya Datta and Sangaralingam Kajanan. 2013. Do app launch times impact their subsequent commercial success? an analytical approach. In *2013 International Conference on Cloud Computing and Big Data (Cloud Com-Asia).* IEEE, 205210.
- Barbara Farbey and Anthony Finkelstein. 1999. Exploiting software supply chain business architecture: a research agenda. (1999).

- Jack Greenfield and Keith Short. 2003. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages,and applications.* ACM, 1627.
- Jacqueline Holdsworth. 1995. *Software Process Design.* McGraw-Hill, Inc.
- Elias Levy. 2003. Poisoning the software supply chain. *Security Privacy, IEEE*1,3 (2003), 7073.
- Galit Shmueli. 2010. To explain or to predict? *Statistical science*(2010), 289310.
- Laurie Voss. 2014. numeric precision matters: how npm down-load counts work.https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts
- Simon Weber and Jiebo Luo. 2014. What makes an open source code popular on git hub?. In *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*. IEEE, 851855.
- Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Mining Software Repositories(MSR), 2016 IEEE/ACM 13th Working Conference on.* IEEE, 351361.
- Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and JessGonzlez-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse.* Springer, 95110.
- Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2014. Patterns of folder use and project popularity: A case study of GitHub repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* ACM, 30.
- Tapajit Dey and Audris Mockus. 2018. Are Software Dependency Supply Chain Metrics Useful in Predicting Change of Popularity of NPM Packages? .In *The 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE18), October 10, 2018, Oulu, Finland.* ACM,New York, NY, USA, 4 pages. https://doi.org/10.1145/3273934.3273942