

Analysis of patterns in the source code of malicious NPM packages

MSc Research Project
MSc Cyber Security

Timofei Milov
Student ID: X22221964

School of Computing
National College of Ireland

Supervisor: Niall Heffernan

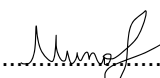
National College of Ireland
MSc Project Submission Sheet
School of Computing



Student Name: Timofei Milov
Student ID: X22221964
Programme: MSc Cyber Security **Year:** 2023/2024
Module: Research Practicum
Supervisor: Niall Heffernan
Submission Due Date: 16/09/2024
Project Title: Analysis of patterns in the source code of malicious npm packages
Word Count: 6175 **Page Count:** 17

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: 
Date: 13/09/2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Analysis of patterns in the source code of malicious NPM packages

Timofei Milov
X22221964

Abstract

Supply chain security is evolving nowadays, since it's hard to control all dependencies and packages that are used in the projects. Everybody can submit their own packages into public registries and there are a lot of malicious software there. Malicious packages can be very dangerous: it can steal sensitive information, affect reliability, infect private networks and so on. NPM is the biggest registry of JavaScript software, it's one of the most popular programming languages. To fight with those threats, we not only need to create identification tools and rules, but to analyze its effectiveness and reliability. The aim of this work is analysis of the source code patterns in malicious NPM packages. To do this we used GuardDog tool and its rules to analyze these patterns, create weights for them and optimize those weights using gradient descent algorithm script. After experiments we identified the most reliable patterns in the source code of malicious npm packages and proposed a system for the pattern analysis.

1 Introduction

Malicious code in public software packages is a rising trend, the amount of it increased by 11973% from 2021 to 2022-2023 years according to Snyk (Idan Digma, 2023). NPM ¹ is the biggest registry of software packages and JavaScript is one of the most popular programming languages. So, that's why it's important to care about security of NPM packages, it is a big scope of software that is used every day by a lot of developers. The specifics of public software packages registry are that anyone can upload any code to the registry. That is very bad from a security perspective. These are some of the possible attacks vectors (Dilki Rathnayake, 2023):

1. Typosquatting – attack actors upload malicious package with small typographical mistake, which look like popular package to the public registry. Users can accidentally install this package.
2. Masquarading – threat actors upload benign code that copy functionality of popular package but add malicious trojan horse code.
3. Dependency confusion – threat actors upload package to public registry with a name of legit internal package, but with higher version. When developer tries to install the package, it will be pulled from the external source.

¹ <https://www.npmjs.com>

4. Dependency hijacking – attack actor compromises benign package and inject malicious code there.

Why is it hard to detect malicious packages? Let's review some of the most common reasons (Dilki Rathnayake, 2023):

1. Public repositories don't require security checks.
2. Installation tools don't test for security.
3. Some packages lack of content visibility.
4. A lot of dependencies.
5. Malicious servers masquerading.

Security vendors try to solve this problem, but their products cost a lot of money and not available for small companies. There is not much open research in this field, security vendors keep their research closed. This work can help small companies to set up their own protection and increase awareness of this problem.

This research is about identification of reliable patterns in malicious NPM packages that can be found with automatic tools. In this research GuardDog tool is used as base for our research. It is written in Python and can analyse PyPi and NPM packages. It uses Semgrep for rules creation. Semgrep is a regular expression tool. In this work we focused on the GuardDog source code rules analysis with the help of gradient descent algorithm, that is used to calculate weights for the rules.

Research question: What are the most reliable patterns in the source code of malicious npm packages that can be detected through automated static analysis?

Next sections are including related work analysis: npm security, malicious npm detection methods, malicious npm detection heuristics; research methodology, what dataset we used, how results are verified, what scientific methods are used; design specification, design of our solution, weights creation and automatic calculation; implementation, how our solution is implemented, what technologies were used; evaluation, test results; conclusion and future work.

2 Related Work

Security code reviews can be an effective way to find security vulnerabilities in the code and keep the program secure. Alfadel *et al.* (2023) in their research analyze different kind of security issues in 10 popular npm open-source packages and how they are solved with the code review, what are the mitigations and so on. They found 171 security-related PRs in these projects, issues related to 14 types of vulnerabilities or security problems: race condition, access control, ReDos, XSS, SQL injection, documentation (poor documentation), improper authentication, sensitive data exposure, remote code execution, overflow, deadlock, improper input validation, vulnerable package, DOS. Code review is especially effective for mitigating: race condition, access control, deadlock and poor documentation. So, it is a good idea to use

code review for security testing. The problem is that it's very time consuming and requires attention of very skilled specialists. That's why it is very important to automate this process. There are different SAST (Static Application Security Testing) tools that can be used for the security review automation. This work is also about automation improvement in static security testing, since the aim of this work is creation of rules for automated detection of malicious npm packages. Ruohonen, Hjerpppe and Rindell (2021) in their work they use automation tool bandit to scan open source PyPI packages for vulnerabilities. They found 749864 security issues in about 197 thousand python packages. The most common security problems are poor exception handling and code injections. This shows how effective can be automation of security code reviews. Even though it produces many false positives, it saves a lot of time of the security engineers. It shows how important to develop automated security testing tools that analyze source code and metadata of the software.

2.1 NPM security

We understand that security code reviews, especially automatic are important and effective, but what can we do improve security of the npm packages? Sejfia and Schäfer (2022) in their work solve this problem by implementing classifier Amalfi, which predicts that package is malicious based on the trained data of malicious and benign packages. They applied machine learning algorithms to detect malicious npm packages. This tool consists of the feature extractor, classifiers (Decision tree, Naïve bayes, One-class SVM), reproducer (build code from source), clone detector (calculates hash and compare it to the known malware hash). Since this work is about analysis of the source code patterns in malicious npm, the most interesting part of this work is features that are used for classification. Sejfia and Schäfer (2022) provide us with these features: access to personally identifying information (credit cards, passwords and so on), access to specific system resources (file system, network), specific APIs (cryptographic, encoding, code generation), package installation scripts, minified code and binary files. The most popular features in the malware according to their research are package installation scripts (33/95), file system access (11/95), network access (10/95). But these are also features that could possibly lead to false positives, since this functionality is also common for benign packages. Sejfia and Schäfer (2022) confirm this in their work by saying that these features are not suspicious by them-selves, but rather their combination becomes suspicious. In their research they show a malware sample found in npm that use post install script to execute script that harvests hostname of a system and send it to attacker. Other example of distractive malicious npm exploiting postinstall scripts is eslint-scope package, that stole npm login tokens (NPM, 2018). Nevertheless, npm scripts is an ordinary functionality of npm package, it's called lifecycle scripts and they help to package software and manage installation (Rohan Mukherjee, 2023).

Vu, Newman and Meyers (2023) in their research paper analyze package security in the PyPi ecosystem. They analyze different tools for the malware detection like: PyPi malware checks, Bandit, OSS detect backdoor. Also, they interviewed some of security researchers and package registries admins. Analysis showed that current tools have a lot of false positives and that it

really reduces the usefulness of these tools. As was mentioned before in previous works, threat actors use the functionality that can be also used by ordinary developers. This shows that our work is really important because we analyze those patterns and range them to decrease false positives and improving confidence of detection of true malicious samples. Other conclusion of their work is that packages registries are overwhelmed with the reports about malicious packages, they can't triage them all and do analyze only a small amount of all suspicious packages. It raises again the importance of the automated solutions. And the last conclusion they came up is that not every malware sample in the packages is really dangerous. Cyber security engineers should always do prioritization of the threats, because it is always more threats than security engineers could mitigate and defuse. In our work we range patterns of the malicious packages not only in a sense of reliability, but also in a sense of severity. Vu, Newman and Meyers (2023) in their work analyze PyPi package security, but it is also true for the NPM ecosystem. Let's analyze features that are used for python packages analysis, Vu, Newman and Meyers (2023) concluded that metaprogramming rules have the highest false positives rate, networking rules are more reliable and process spawn and subprocess rules are not reliable like metaprogramming. Networking rules showed that they trigger mostly on true malicious samples and not trigger on benign, the difference is about 10 times. So, malicious packages have malicious network patterns 10 times more often, then benign packages.

Li *et al.* (2023) in their work propose a tool for malicious code detection in PyPi and NPM packages. They use such innovative approaches as source code slicing, inter-procedural analysis, cross-file inter-procedural analysis. In the background of the problem section, they provide classification of malicious code: hidden authentication, backdoor, cryptojacking (mining), embedded shell (suspicious scripts), suspicious obfuscation (code obfuscation), remote control, send sensitive information, suspicious commands execution. Nevertheless, Li *et al.* (2023) stated that labeling source code with obfuscation as malicious is not really reliable behavior since obfuscation could be used by benign programs. In their research paper they also provided Virus Total analysis of the different malware packages. The analysis shows that the most detection accuracy was in the suspicious commands, cryptomalware and suspicious embedded shell categories. The worst results were in the sensitive data leakage and obfuscation categories, they have many false positives and true negatives. That means that exact suspicious commands detected is a more reliable pattern than obfuscation and sensitive data leakage. These indicators also seem much more destructive. In the evaluation section they show statistics of malware identification with their tool. Most of the Java Script malware they detected were from send sensitive information category (132 out of 236), suspicious obfuscation (47 out of 236) and suspicious command execution (16 out of 236). Though more packages were detected in send sensitive information and obfuscation categories, we think that suspicious commands execution and embedded shell patterns are more reliable for malware detection. But these categories are also harder to detect, because it requires a good knowledge of the threat actors' techniques and threat intelligence.

Zahan *et al.* (2022) in their research work analyse malicious npm packages and weak links in the supply chain or in other words indicators of the malware in npm package. Most of the work is related to npm metadata analysis, but they have an interesting analysis of installation scripts

in npm. Zahan *et al.* (2022) in their work describe how installation scripts can be used for malicious purposes: to steal sensitive data, to start a child process for a backdoor or other code execution. Also threat actor can infiltrate into dependencies and attack using tampered dependencies. Though presence of installation scripts doesn't mean that package is automatically malicious as was mentioned before, it adds a lot of capabilities for the threat actors and can't be ignored. NPM recommends avoiding installation scripts for packages, since you usually don't really need it. Zahan *et al.* (2022) found that 33249 of about 1,63 million packages contain installation scripts, 3412 of 3635 (93.9 %) malicious scripts contain at least one installation script. Installation scripts from the analysis usually: transfer data to third party servers, download malicious tool, reverse shell, delete directories. Malicious installation scripts usually use keywords like curl, wget, /etc/shadow, /etc/passwd. Installation scripts are important indicators of malware to consider, but they should be combined with other indicators to avoid false positives.

Huang *et al.* (2024) proposed DONAPI tool for malicious npm packages detection based on machine learning, that uses static and dynamic analysis. They used these features in the project for malware identification: send sensitive information outside, query system env variables, download the content and execute it as string, write to file and execute, read the context of the file and execute, read and execute code dynamically, modify file permissions and process creation, operating system identification, modify the data flow of system command execution results, systems commands execution, performing sensitive file operations. They didn't range these features, but they stated that you can't trust separate rules and should create more complex analytical system. That's why we created analytical system with weights for every rule.

Next very interesting tool is the GuardDog tool that analyze PyPi and NPM packages for malware and supported by the Data Dog company. Ellen Wang, Christophe Tafani-Dereeper, (2022). Ellen Wang, Christophe Tafani-Dereeper (2022) developed it during summer internship in Data Dog security research department. We used this tool for our work because sometimes it is better not to reinvent the wheel and just use it. We use this tool and it's source code rules to analyze source code patterns of malicious npms and create analytical system that reduces false positives and helps triage findings. Let's analyze their research and tool. Ellen Wang, Christophe Tafani-Dereeper, (2022) reverse engineer malicious python packages and found that the most popular techniques are: typosquatting for initial access, execution of payloads with eval or exec, downloading of additional malware, exfiltrate system information, environment variables (typically through HTTP). These are really common techniques as we saw it already in previous works. But how they detect malware and what are the features of the GuardDog? Guarddog uses Semgrep rules, that are based on regular expressions. Semgrep rules are mostly used for the source code analysis and Python is used for the metadata analysis. The whole tool is written in Python, which is very convenient, since Python is easy to read and understand, it is rather easy to write, so it is a good choice. This work is focused on source code patterns in malicious npm, so let's analyze this part of the tool. There are 10 Semgrep rules for source code analysis of npm (Timofey, 2024):

- NPM serialize environment (if a package serializes process.env to exfiltrate env variables).

- NPM obfuscation (if a package uses common obfuscation method used by malware).
- NPM silent process execution (if a package silently executes an executable).
- Shady links (if a package has a URL with suspicious extension).
- NPM exec base64 (if a package executes code through eval).
- NPM install script (if a package has pre or post install script with automatically running commands).
- NPM steganography (if a package retrieves hidden data from image and executes it).
- Bidirectional characters (if a package has bidirectional characters, that display code differently than its actual execution).
- NPM DLL hijacking (if a package manipulates a trusted application into loading a malicious DLL).
- NPM exfiltrate sensitive data (if a package reads and exfiltrates sensitive data from the local system).

Most of these rules were analysed in previous works above, except NPM steganography, bidirectional characters, NPM DLL hijacking. NPM steganography and NPM DLL hijacking could be classified as malicious code execution and bidirectional characters as an obfuscation.

NPM security and supply chain security are very important. Security source code review is good, but we need more automation. There are many tools for an automated security source code review and malware detection, but the problem they are not very reliable and generates a lot of false positives. We can divide all rules detection malware in packages in 3 categories: suspicious code execution, obfuscation, sensitive data exfiltration. The most important and reliable is suspicious code execution, then goes sensitive data exfiltration and obfuscation is the last in this rating. This can be concluded from the literature review, more information is provided in next sections below.

2.1 Optimization algorithms

To create analytical system for the malware analysis rules in npm we need to add weights for them. To adjust these weights to be the most effective in malware detection we need an optimization algorithm that could do it automatically. In this section we analyze some popular optimization algorithms: gradient descent, natural gradient descent, Adam and genetic algorithm.

Tyurin (2023) in his work analyze and compare gradient descent and natural gradient descent algorithms. Gradient descent is based on parameters update using gradient of the function. For example, we have a function $L(\theta)$, where θ is parameters, $L(\theta)$ is a loss function. Parameters changes according to this formula:

$$\theta_{t+1} = \theta - a * \nabla L(\theta) \quad (1)$$

a here is the learning rate and ∇ is the gradient, t is the number of steps in the algorithm. a defines the size of the step and steps go in the opposite direction of the gradient. The goal of

the gradient descent is to find minimum of the function, most of the time it is a loss function. In the natural gradient descent Fisher matrix is used and helps to normalize the gradient. Natural gradient descent shows good results when parameters have big correlation, but it is not always very time effective. Classical gradient descent is a simple and straight forward algorithm.

Adam is another popular optimization algorithm mostly used in deep learning. Adam uses past gradient information to speed the convergence, it leverages exponentially decaying average of the past gradients and exponentially decaying average of the past squared gradients, which smooth parameters updates. Also, parameters have different learning rates, which is also can be very effective in certain situations. Adam is not very good for small datasets and where you need precise control in tuning (Wei, 2024). Because of that it is not very suitable algorithm for our work, since we have small dataset and need a good tuning control.

Yang (2023) in his work uses genetic algorithm to improve motion tracking technology. Let's see if we can also use genetic algorithm in our work. Genetic algorithm simulates evolution mechanisms to find the optimal solution of the problem. It uses 3 operations: heredity, mutation and selection. And every solution is like a chromosome containing multiple genes. The result of the algorithm is the optimal chromosome from the initial population. Algorithm starts with the population initialization, assigning fitness values to the solutions, then new generation is created with the operations: crossover and mutation. Then according to the fitness score natural selection is carried out. Then fitness score is calculated again for the new generation. And the process repeats again until special condition. The plus of the algorithm is that it doesn't require any knowledge about the object function. It has good abilities for parallel computing. Cons of the genetic algorithm are that sometimes it has problems with local optimal solution and needs to start over for several times and also genetic algorithm calculations may be long. So, again it is not very suitable for our work, since npm scanning is a long process and we don't want to spend additional time on long working algorithms.

Overall, we decided to use classical gradient descent algorithm, since it is very simple, straight forward, easy to explain calculations and results, we have a small dataset. We think that gradient descent is the most suitable optimization algorithm for this work.

3 Research Methodology

In this work we use the GuardDog tool to analyse malicious npm source code patterns and create analytical system that provide confidence and severity score for each tested package. To reach these goals we added confidence and severity weights to each source code rule in the GuardDog and developed system that provides scores. Then we tested our weights on the dataset of malicious and benign npm packages. After that we developed a program that optimize weights automatically with the gradient descent algorithm that was chosen in the literature review. We labelled test group of the malicious and benign packages with the confidence and severity scores according to the available information about these packages and malware. This program runs the GuardDog tool and compare labelled results with the actual tool output and updates rules weights with the help of gradient descent. Gradient descent works

on reducing a loss function, that computes deviation between labelled results and actual tool output. Then we compared optimized weights with the manually created ones. This showed how accurate were our analytical weights creation in comparison with the automatically calculated weights. Then we also compared results of the analytical system with the different weights. Weights optimization with gradient descent was implemented to assess our analytical approaches and research this method, that can be used for further research in the field of source code patterns analysis. To protect our computers from the potential threat of the malware we did all experiments in the isolated environment on the virtual machine. Rules of the GuardDog were described in the literature review, there are 10 rules for npm malware source code detection. For each rule we added 2 weights between 1 and 10, for confidence score and severity score. Confidence score is how confident we are that package is malicious and severity score is how critical the vulnerability is. Confidence and severity are the scores between 0 and 100 %.

3.1 Dataset

In this work we used a dataset with malicious npm packages vetted by security researchers. This dataset was uploaded on Github by the GuardDog security researchers and it is always updating ('DataDog/malicious-software-packages-dataset', 2024). It is issued under the Apache 2 licence, so we totally free to use it in this work. 26 malicious packages were used for this research from this dataset. Here is the listing of them:

2024-04-17-@mosfe_beam-git-util-v1.0.0.tar.gz, 2024-04-17-@mosfe_beam-lint-config-v1.0.5.tar.gz, 2024-04-17-@mosfe_beam-plugin-s3plus-v0.1.16.tar.gz, 2024-04-17-@mosfe_beam-v2.0.0.tar.gz, 2024-04-17-@mosfe_owl-config-v1.0.8.tar.gz, 2024-04-17-@mosfe_portal-proxy-v1.1.7.tar.gz, 2024-04-17-@mosfe_s3plus-v0.1.7.tar.gz, 2024-04-17-@mosfe_sso-sdk-v0.0.6.tar.gz, 2024-05-03-@sxmp_detector-v0.100.4.tar.gz, 2024-05-05-avx-web-build-v1000.0.1.tar.gz, 2024-05-29-hello-1st-anni-v4.4.25.tar.gz, 2024-05-30-flexfone-reseller-v100.0.3.tar.gz, 2024-05-30-flexfone-reseller-v100.0.4.tar.gz, 2024-05-30-flexfone-reseller-v100.0.5.tar.gz, 2024-05-30-hello-1st-anni-v4.4.30.tar.gz, 2024-05-30-hello-1st-anni-v4.4.31.tar.gz, 2024-05-30-react-zutils-v1.0.1.tar.gz, 2024-06-03-hello-1st-anni-v0.0.1-security.tar.gz, 2024-06-07-@atticuss-sra_test-pkg-x4-v3.2.0.tar.gz, 2024-06-07-@atticuss-sra_test-pkg-x4-v3.2.1.tar.gz, 2024-06-07-@atticuss-sra_test-pkg-x4-v3.2.2.tar.gz, 2024-06-07-@atticuss-sra_test-pkg-x4-v3.2.3.tar.gz, 2024-06-07-test-pkg-x5-v3.2.0.tar.gz, 2024-06-07-test-pkg-x5-v3.2.1.tar.gz, 2024-06-07-test-pkg-x5-v3.2.2.tar.gz, 2024-06-07-test-pkg-x5-v3.2.3.tar.gz.

Also, we used 26 benign popular packages from the public NPM registry:

async-3.2.5.tgz, axios-1.7.2.tgz, chalk-5.3.0.tgz, date-fns-3.6.0.tgz, day-0.0.2.tgz, express-4.19.2.tgz, grunt-1.6.1.tgz, html-entities-2.5.2.tgz, jest-29.7.0.tgz, js-yaml-4.1.0.tgz, karma-6.4.3.tgz, lodash-4.17.21.tgz, mocha-10.6.0.tgz, moment-2.30.1.tgz, mongoose-8.5.1.tgz, mysql-2.18.1.tgz, passport-0.7.0.tgz, path-0.12.7.tgz, pm2-5.4.2.tgz, sentry-0.1.2.tgz, socket-

0.0.1.tgz, ts-node-10.9.2.tgz, vue-3.4.32.tgz, vue-router-4.4.0.tgz, web3-4.11.0.tgz, webpack-5.93.0.tgz.

There are also 6 packages (3 malicious and 3 benign) for the experiments with the weight's optimization program, they are also labelled with the scores:

2024-04-17-@mosfe_beam-git-util-v1.0.0.tar.gz	Confidence: 80% Severity: 70%
2024-05-30-flexfone-reseller-v100.0.4.tar.gz	Confidence: 80% Severity: 60%
2024-05-30-hello-1st-anni-v4.4.31.tar.gz	Confidence: 80% Severity: 65%
day-0.0.2.tgz	Confidence: 10% Severity: 10%
passport-0.7.0.tgz	Confidence: 10% Severity: 10%
sentry-0.1.2.tgz	Confidence: 10% Severity: 10%

3.2 Experiments

This is a description of our experiments. First, we tested our analytical system with manually added weights on all 26 benign and 26 malicious packages. We analysed which rules were used, what are confidence and severity scores. Then we adjust weights with the gradient descent program on 6 packages. We picked only 6 packages because gradient descent runs GuardDog many times and scanning of the package is not very fast process, so experiments with more packages could be very long. After weights optimization we tested it on all 26 benign and 26 malicious packages. Then we compared results and analysed the difference in weights.

4 Design Specification

Let's see the weights and how confidence and severity scores calculated. These are the weights for the confidence score:

npm-serialize-environment: 5
npm-obfuscation: 4
npm-silent-process-execution: 8
shady-links: 6
npm-exec-base64: 9
npm-install-script: 5
npm-steganography: 6
bidirectional-characters: 5
npm-dll-hijacking: 8
npm-exfiltrate-sensitive-data: 6

These are the weights for the severity score:

npm-serialize-environment: 4
npm-obfuscation: 3
npm-silent-process-execution: 7

shady-links: 4
 npm-exec-base64: 9
 npm-install-script: 6
 npm-steganography: 7
 bidirectional-characters: 4
 npm-dll-hijacking: 8
 npm-exfiltrate-sensitive-data: 5

Formula for the confidence and severity scores is based on logarithmic scale, that is done to normalize scores (Formula 2):

$$scores = 100 * \frac{\log(1+tw)}{\log(1+mw)} \quad (2)$$

Where tw is total weight, sum of the weights and mw is max possible weight, so sum of all weights. So, when rule was used, and pattern detected its weight adds to the total weight and then all used rule's weights summed up into total weight.

This is the formula for gradient descent (Formula 1):

$$\theta_{t+1} = \theta - a * \nabla L(\theta) \quad (1)$$

where θ is weights, $L(\theta)$ is a loss function, ∇ is a gradient, a is a learning rate and t is an iterations number. In this project learning rate is 0.1 and iterations is 100. This is because npm package scan is long and we need to make program execution faster.

Gradient is calculated using finite difference method, because it is not possible to use analytical gradient calculation in this situation. The formula for the gradient calculation is (Formula 3):

$$gradient = \frac{lfp-lfm}{2*eps} \quad (3)$$

Where lfp is loss function plus and lfm is loss function minus, eps is epsilon const value that is 0.3 in this work. Loss function plus or loss function minus is calculated like this (Formula 4):

$$lfp(m) = mse(labelled, predicted) \quad (4)$$

where mse is mean squared error of predicted confidence and severity scores. These predicted scores are compared to the labelled values results. The difference between loss function plus and loss function minus is that weights are changed to $+eps$ and $-eps$ accordingly before predictions.

4.1 Program design

In the Fig. 1 you can see a program design for our project. Weights Optimization runs the GuardDog tool to execute gradient descent algorithm and change weights according to labelled packages. Weights are stored in the JSON files. Results are printed in the console.

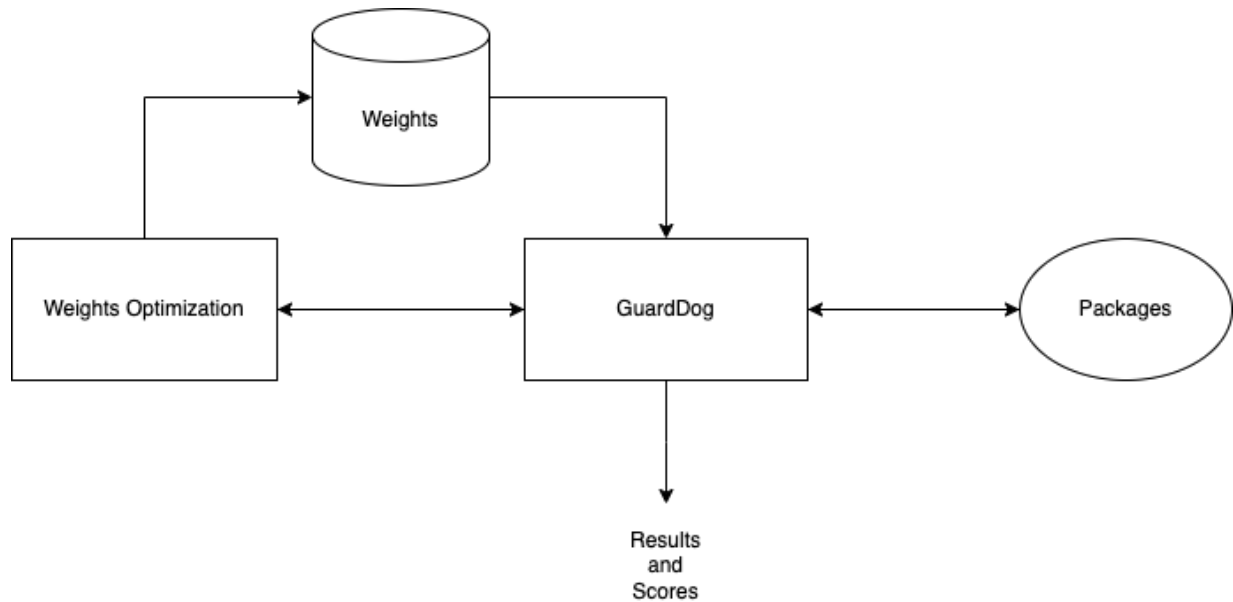


Fig. 1 – Program design.

5 Implementation

We developed analytical score system and weights optimization program in Python language, since the GuardDog tool itself is written in Python. Python is a good interpretation programming language with simple syntax, it is very popular in cyber security field. Weights are stored in different JSON files. JSON files are more transparent than a database, that's why we choose them. They are stored in `guarddog-research-testing/guarddog:confidence_weights.json`, `severity_weights.json`, `confidence_weights_backup.json`, `severity_weights_backup.json` (Timofey, 2024). Backup JSON files contain original weights to save them. `Confidence_weights.json` and `severity_weights.json` change during gradient descent algorithm. Analytical system code stored in `guarddog-research-testing/guarddog/analyzer/analyzer.py` in the `analyze_sourcecode` function. Weights optimization are stored in the different repository and this is a fully our self-written code.

To run the GuardDog tool from our weights optimization program we used Python module `subprocess` and `Popen` interface (Python, 2024). That helps us to work in the Python virtual environment of the GuardDog and also read results from the console/stdout of this process.

In the Fig. 2 you can see how results look like. You can see how confidence and severity scores displayed and also, which rules detect patterns in the package.

```
Confidence total weight: 15/62
Severity total weight: 13/57

Confidence: 66.92 %
Severity: 64.99 %

shady-links, npm-obfuscation, npm-install-script
66.92
64.99
```

Fig. 2 – Results displayed.

6 Evaluation

6.1 Experiment 1

We used weights optimization program based on genetic algorithm to calculate new weights. This is how new weights look like:

Confidence Score

```
npm-serialize-environment: 5
npm-obfuscation: 15.235367499999999
npm-silent-process-execution: 8
shady-links: 10.9757399999999984
npm-exec-base64: 9
npm-install-script: 4.2232774999999992
npm-steganography: 6
bidirectional-characters: 5
npm-dll-hijacking: 8
npm-exfiltrate-sensitive-data: 11.808979500000012
```

Severity Score

```
npm-serialize-environment: 4
npm-obfuscation: 5.4873344999999993
npm-silent-process-execution: 7
shady-links: 5.3226459999999998
npm-exec-base64: 9
```

npm-install-script: -0.7020749999999869
 npm-steganography: 7
 bidirectional-characters: 4
 npm-dll-hijacking: 8
 npm-exfiltrate-sensitive-data: 3.6885410000000007

These weights are different that we created manually. Weights were changed only if rules were used in the experiments dataset. Let's analyse confidence score weights. Npm-obfuscation weight has changed from 4 to 15.235367499999999, the difference is 11.2353675. This is maybe because npm obfuscation was used in 2 packages from 6, that were used for weights optimization. Shady links has changed from 6 to 10.975739999999984, the difference is 4.97574. Npm install weight has changed from 5 to 4.223277499999992, the difference is 0.7767225. Npm exfiltrate sensitive data weight has changed from 6 to 11.808979500000012, the difference is 5.8089795. Some of the weights changed dramatically, this can be explained by the nature of the dataset that is rather small and focused on certain malicious patterns like npm obfuscation, which is really popular element of the malware, shady links, sensitive data exfiltration.

Let's see what has changed in severity weights. Npm obfuscation weight has changed from 3 to 5.487334499999993, the difference is 2.4873345. Shady links weight has changed from 4 to 5.322645999999998, the difference is 1.322646. Npm install script weight has changed from 6 to -0.7020749999999869, in the program it is changed to 0, so the difference is 6. Npm exfiltrate sensitive data has changed from 5 to 3.6885410000000007, the difference is 1.311459. We can see that npm install script weight become 0 after optimization, that means that it become not dangerous, according to the score. This makes sense as npm install script shouldn't be an indicator by itself, this was also mentioned in related work section.

In this section we analyze result scores with optimized and manual crafted weights. We run the GuardDog tool with our analytical system on experiment dataset of packages with 3 benign and 3 malicious packages. In the Table 1 you can see results with the manual crafted weights, in the Table 2 you can see results with the optimized weights. Average confidence score difference is 5.545. Average severity score difference is 15.235.

Number	Package name	Confidence score	Severity score	Rules
1	2024-04-17-@mosfe_beam-git-util-v1.0.0.tar.gz	66.92	64.99	npm-install-script, npm-obfuscation, shady-links
2	2024-05-30-flexfone-reseller-v100.0.4.tar.gz	59.98	61.2	npm-install-script, npm-exfiltrate-sensitive-data
3	2024-05-30-hello-1st-anni-v4.4.31.tar.gz	55.58	56.71	npm-obfuscation, npm-install-script
4	day-0.0.2.tgz	0	0	

5	passport-0.7.0.tgz	0	0	
6	sentry-0.1.2.tgz	43.25	47.92	npm-install-script

Table 1 – Manual crafted weights results.

Number	Package name	Confidence score	Severity score	Rules
1	2024-04-17-@mosfe_beam-git-util-v1.0.0.tar.gz	77.77	60.41	npm-install-script, npm-obfuscation, shady-links
2	2024-05-30-flexfone-reseller-v100.0.4.tar.gz	63.94	34.7	npm-install-script, npm-exfiltrate-sensitive-data
3	2024-05-30-hello-1st-anni-v4.4.31.tar.gz	68.08	44.05	npm-obfuscation, npm-install-script
4	day-0.0.2.tgz	0	0	
5	passport-0.7.0.tgz	0	0	
6	sentry-0.1.2.tgz	37.29	0	npm-install-script

Table 2 – Optimized weights results.

According to manual crafted weights, this is the rating of the most reliable patterns:

1. npm-exec-base64 (code execution with eval) weight: 9.
2. npm-silent-process-execution / npm-dll-hijacking weight: 8.
3. shady-links / npm-steganography / npm-exfiltrate-sensitive-data weight: 6.

And the most dangerous patterns:

1. npm-exec-base64 (code execution with eval) weight: 9.
2. npm-dll-hijacking weight: 8.
3. npm-silent-process-execution / npm-steganography weight 7.

According to the new optimized weights, the rating of the most reliable patterns looks like this:

1. npm-obfuscation weight: 15.235367499999999.
2. npm-exfiltrate-sensitive-data weight: 11.808979500000012.
3. shady-links weight: 10.9757399999999984.

and the most dangerous patterns:

1. npm-exec-base64 (code execution with eval) weight: 9.
2. npm-dll-hijacking weight: 8.
3. npm-silent-process-execution / npm-steganography weight 7.

6.2 Experiment 2

In the second experiment we analyzed with the GuardDog tool all 26 malicious and 26 benign packages from the dataset. Let's see what the difference between optimized and manual crafted weights is. An average of the confidence score difference is 9.903 and an average of the severity score difference is 11.977. This is statistics of rules usage on our dataset:

npm-serialize-environment: 0
npm-obfuscation: 21
npm-silent-process-execution: 2
shady-links: 11
npm-exec-base64: 0
npm-install-script: 25
npm-steganography: 0
bidirectional-characters: 0
npm-dll-hijacking: 0
npm-exfiltrate-sensitive-data: 4

Npm install scripts are the most used rule (25), because it very useful for threat actors and npm obfuscation (21) is the second most used rule, since threat actors always want to hide their malicious code. Next popular rule is the shady links (11), threat actors can use it to send sensitive data or download additional malware.

6.3 Discussion

After these experiments we can make conclusions about what are the most reliable patterns of the malicious npm packages. Weights optimization with the gradient descent from the experiment 1, showed that the most reliable pattern is obfuscation. This is true, because mostly all malicious packages contain some form of obfuscation, but this pattern can also lead to a lot of false positives. Benign packages could also use obfuscation. Sensitive data exfiltration is another pattern with high weight, that is also understandable, since the aim of the many malware packages is to steal data. And shady links pattern is also one of the most reliable, according to gradient descent experiment. Our analytical weights creation was mostly focused on identifying patterns that can't be ignored like code execution with eval, silent process execution and others. We don't have such malware samples in our dataset, so the results are a bit focused on one type of malware. But we suggest that this type is also the most popular. So, the most reliable pattern of the npm malware is the combination of obfuscation, sensitive data exfiltration and shady links. If you found these 3 patterns in one package, that means that this package could be marked as malicious with the high level of confidence. Also 2 of these patterns can identify a malware like it was with 2024-04-17-@mosfe_beam-git-util-v1.0.0.tar.gz package, that has npm-install-script, npm-obfuscation, shady-links patterns and have 77.77 confidence score with optimized weights.

Beside identification of the most reliable patterns we created a system of pattern analysis based on weights and it's optimization with gradient descent. This work showed that the system can be used to identify reliable rules for malware detection. It has a lot of potential and can be used for further research in this field. It will be even more effective on big datasets

7 Conclusion and Future Work

In this work we identified the most reliable source code patterns for the npm malware detection, it is the combination of obfuscation, sensitive data exfiltration and shady links. This conclusion is based on experiments with gradient descent weights optimization and our analysis of rules

and related works. Also, we propose our analysis system that is based on rule weights and gradient descent optimization of these weights. This system has a good potential.

Future work can be focused on experiments with bigger datasets, but it is very time consuming, because each package scanning takes time. More tools and rules could be used for the future experiments. To research the field of malicious npm packages dynamic analysis tools could be also considered. These tools will analyse malicious packages behavior, and it can be combined with the static analysis. The field of supply chain security and malicious packages is very big and research in this field is in big demand, because these attacks become more popular every day.

References

- Alfadel, M. *et al.* (2023) ‘Empirical analysis of security-related code reviews in npm packages’, *Journal of Systems and Software*, 203, p. 111752. Available at: <https://doi.org/10.1016/j.jss.2023.111752>.
- ‘DataDog/malicious-software-packages-dataset’ (2024). Datadog, Inc. Available at: <https://github.com/DataDog/malicious-software-packages-dataset> (Accessed: 17 June 2024).
- Dilki Rathnayake (2023) *Understanding Malicious Package Attacks and Defense Strategies for Robust Cybersecurity | Tripwire*. Available at: <https://www.tripwire.com/state-of-security/understanding-malicious-package-attacks-and-defense-strategies-robust> (Accessed: 15 June 2024).
- Ellen Wang, Christophe Tafani-Dereeper (2022) *Finding malicious PyPI packages through static code analysis: Meet GuardDog | Datadog Security Labs*. Available at: <https://securitylabs.datadoghq.com/articles/guarddog-identify-malicious-pypi-packages/> (Accessed: 15 April 2024).
- Huang, C. *et al.* (2024) ‘DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping’. arXiv. Available at: <http://arxiv.org/abs/2403.08334> (Accessed: 15 April 2024).
- Idan Digmi (2023) *The rising trend of malicious packages in open source ecosystems, Snyk*. Available at: <https://snyk.io/blog/malicious-packages-open-source-ecosystems/> (Accessed: 14 June 2024).
- Li, N. *et al.* (2023) ‘MalWuKong: Towards Fast, Accurate, and Multilingual Detection of Malicious Code Poisoning in OSS Supply Chains’, in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg, Luxembourg: IEEE, pp. 1993–2005. Available at: <https://doi.org/10.1109/ASE56229.2023.00073>.
- NPM (2018) *Compromised version of eslint-scope published*. Available at: <https://status.npmjs.org/incidents/dn7c1fgr7ng> (Accessed: 25 July 2024).
- Python (2024) *subprocess — Subprocess management, Python documentation*. Available at: <https://docs.python.org/3/library/subprocess.html> (Accessed: 31 July 2024).

Rohan Mukherjee (2023) *Understanding package.json II: Scripts, Dyte*. Available at: <https://dyte.io/blog/package-json-scripts/> (Accessed: 25 July 2024).

Ruohonen, J., Hjerpe, K. and Rindell, K. (2021) ‘A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI’, in *2021 18th International Conference on Privacy, Security and Trust (PST)*. *2021 18th International Conference on Privacy, Security and Trust (PST)*, pp. 1–10. Available at: <https://doi.org/10.1109/PST52912.2021.9647791>.

Sejia, A. and Schäfer, M. (2022) ‘Practical Automated Detection of Malicious npm Packages’, in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1681–1692. Available at: <https://doi.org/10.1145/3510003.3510104>.

Timofey (2024) ‘Timofey21/guarddog-research-testing’. Available at: <https://github.com/Timofey21/guarddog-research-testing> (Accessed: 29 July 2024).

Tyurin, A. (2023) ‘Comparative Analysis of the Rate of Convergence of the Methods of Gradient Descent and Natural Gradient Descent in Regression Analysis Problems’, in *2023 5th International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency (SUMMA)*. *2023 5th International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency (SUMMA)*, pp. 252–254. Available at: <https://doi.org/10.1109/SUMMA60232.2023.10349623>.

Vu, D.-L., Newman, Z. and Meyers, J.S. (2023) ‘A Benchmark Comparison of Python Malware Detection Approaches’, in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 499–511. Available at: <https://doi.org/10.1109/ICSE48619.2023.00052>.

Wei, D. (2024) ‘Demystifying the Adam Optimizer in Machine Learning’, *Medium*, 30 January. Available at: <https://medium.com/@weidagang/demystifying-the-adam-optimizer-in-machine-learning-4401d162cb9e> (Accessed: 29 July 2024).

Yang, Y. (2023) ‘Upgrading and Optimization of Motion Tracking Technology Based on Genetic Algorithm’, in *2023 IEEE 15th International Conference on Computational Intelligence and Communication Networks (CICN)*. *2023 IEEE 15th International Conference on Computational Intelligence and Communication Networks (CICN)*, pp. 348–352. Available at: <https://doi.org/10.1109/CICN59264.2023.10402265>.

Zahan, N. *et al.* (2022) ‘What are Weak Links in the npm Supply Chain?’, in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 331–340. Available at: <https://doi.org/10.1145/3510457.3513044>.