Poster: Using CodeQL to Detect Malware in npm

Matías Gobbi[*]
Bundeswehr University Munich
Research Institute CODE
Munich, Germany

Johannes Kinder
Ludwig-Maximilians-Universität
München (LMU Munich)
Munich, Germany

ABSTRACT

Malicious packages are a problem on npm, but like other malware, they are rarely completely novel and share large semantic similarities. We propose to leverage the existing static analysis framework CodeQL to find malware on npm; but instead of detecting variants of vulnerabilities, we use it to detect variants of malware. We present a methodology for writing queries from recently reported packages, as a way of defining semantic signature for specific malicious behavior, where a single one can then be used to match entire families of malware. An iteration of our approach resulted in the discovery of 125 malicious packages from the registry, without producing a single false alarm.

CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**; • **Software and its engineering** → *Automated static analysis.*

KEYWORDS

malware, npm, static analysis

ACM Reference Format:
Matías Gobbi and Johannes Kinder. 2023. Poster: Using CodeQL to Detect Malware in npm. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3576915.3624401

1 INTRODUCTION

Malicious actors abuse package repositories to host and distribute malware, with all parties using these ecosystems becoming possible targets. Due to the wide adoption of package repositories in software development, we need practical solutions to meet this threat and detect and eliminate malware.

Given its size and popularity, npm tends to be a primary target for these attacks [12, 22]. In the last few years, there have been exceptional cases of highly targeted attacks which employed novel techniques to achieve their objective [8, 10, 17]. But more often, large numbers of malware are published in waves to registries, where all samples of such a campaign share similar malicious behavior. For instance, the `crossenv` case refers to a collection of almost 40 information-stealing typosquatting packages [16]. Some campaigns do not necessarily have malicious intent, but still break the terms of service: bug-bounty packages are attempts by independent security researchers to exploit *dependency confusion* vulnerabilities [2].

The security of the registry relies on its community, where npm triages the malware reported by users before deleting it. Given that this mechanism is limited, several solutions have been proposed in the literature, e.g., to reduce the potential attack surface of dependencies [9, 19, 21] or to protect from malicious updates [6, 7, 13]; other approaches function as overall malware detectors [5, 11, 15] or capture common patterns employed by attackers [14, 18, 20].

In this work, we propose a static analysis approach for the detection of samples similar to recently found malicious packages in registries. With the use of CodeQL, a security engine designed for bug finding, we are able to define semantic signatures for current malware campaigns running in npm. Leveraging removed packages as templates for writing queries, we focus on having an accurate method to match specific behavior seen in malware, keeping to a minimum the amount of potential false alarms. We discovered and reported 125 malicious packages with the use of our technique, without wrongly matching a single package along the way.

2 STATIC ANALYSIS WITH CODEQL

CodeQL is a static code analysis engine designed to automatically check for vulnerabilities in a project by executing queries against a database generated from the code. By leveraging CodeQL's taint tracking capabilities, we are able to capture malicious data flows between sources and sinks in npm packages, where the analysis returns the list of nodes involved in the detected flows of information. Furthermore, its integration with IDEs allows a human analyst to manually review the generated reports with ease.

A CodeQL database contains multiple intermediate representations of the code, including the abstract syntax tree, the control flow graph, and the data flow graph [1, 3, 4]. The libraries define classes to provide a layer of abstraction over the generated relational tables, resulting in an object-oriented view of the information. CodeQL queries are written in the logic programming language QL, which derives its semantics from Datalog [1, 3, 4]. After running a query, the produced results are styled as alerts in an interpreted format that points directly to elements from the source code.

In Listing 1 we show a simplified query which uses static taint analysis to detect specific behavior seen in malware. The package performed data exfiltration by sending operating-system information to a controlled address via a request. We defined a custom taint tracking configuration (lines 1 to 20) for this attack pattern. We declare the source node (lines 3 to 12) of the configuration, which retrieves specific data via the calls to `os.hostname()`, `os.homedir()`, and `os.userInfo()`. We declare the sink node (14 to 18) of the configuration, that uploads this information to a web server via passing it as an argument of a POST request, `req.write(...)`. Finally, with the `from.where.select` clause (22 to 24), we select all source-sink

[*] Also with Ludwig-Maximilians-Universität München (LMU Munich).

CCS '23, November 26–30, 2023, Copenhagen, Denmark

```
class TTConfiguration extends TaintTracking::Configuration {

  override predicate isSource(Node source) {
    exists( SourceNode os
          | os = moduleMember("os", [ "hostname"
                                    , "homedir"
                                    , "userInfo"
                                    ]
                             )
          | os = source.(InvokeNode).getCalleeNode()
          )
  }

  override predicate isSink(Node sink) {
    exists( ClientRequest client
          | sink = client.getAMemberCall("write").getAnArgument()
          )
  }

}

from TTConfiguration cfg, PathNode source, PathNode sink
where cfg.hasFlowPath(source, sink)
select source, sink
```

Listing 1: Abbreviated CodeQL query that matches source code from information stealing malicious package

pairings where there is a flow satisfying the conditions from the defined configuration.

3 METHODOLOGY

We need to use malware as a template for designing an appropriate query able to capture its semantics, and for this, we take the removal of a sample from the registry as ground truth. When a malicious package is detected, npm withdraws it and sets a security placeholder in its place. Taking advantage of this insight, and knowing that the same malware tends to be uploaded multiple times with slight syntactic changes, we propose an involved methodology to utilize recently reported samples to find further instances of malicious packages sharing similar behavior. The proposed technique consists of the following steps:

Step 1. Detecting Removed Packages: The publishing of a security placeholder in the registry should be a sign of the discovery of a potentially malicious package. Such events could be recognized by an automatic monitoring process. This step requires mirroring the registry to secure a copy of the malware's source code after its withdrawal. Note that not every removed package is malicious, and not every malicious package belongs to a malware campaign.

Step 2. Manual Inspection: An in-depth examination of the source code helps to understand the malware's intentions. Which consists of looking for common patterns seen in malicious packages [5, 12], such as the spawning of processes, the requests made, the interactions with the file system, or the runtime generation and loading of code, while studying the related information flows. Identifying multiples samples with similar semantics is a clear sign that a malware campaign might be active in the registry. Lastly, this step distinguishes attacks patterns that are out of scope for the analysis, such as those containing executable binaries, which should be covered with orthogonal methods for malware detection.

Step 3. Query Development: We design a CodeQL query to specifically match the identified behavior seen in the collected malware sample. This step relies on the domain knowledge of the analyst. A query has to be general enough to be able to find malicious packages in the wild and, at the same time, needs to be accurate enough to avoid producing too many false alarms. The analyst has to capture the semantics of all the relevant steps of the data flow in the malware, with the written query. In the case of a package that encodes sensitive information before stealing it, the developer has to consider the encoding process as an additional taint step in the tracking configuration.

Step 4. Query Refinement: The designed CodeQL query has to match intrinsically malicious behavior. One has to be aware of coding practices that might be wrongly flagged as malware due to sharing similar semantics, and purposely avoid capturing them. During this step, the expertise of the analyst plays an essential role given that it is necessary to strike a balance between generality and accuracy without knowing the true distribution in the registry. Continuing the case of information stealing, and considering there is no policy regulating user tracking, it is critical for the performance of the query to recognize the type of data being exfiltrated, since not every piece of information is sensible.

Step 5. Application to Registry: Once a query capturing the behavior of a malicious package was developed, it has to be applied to the entire registry. For every flagged sample, CodeQL provides the contextual information detailed in the query together with the exact position in the code responsible of the match. Leveraging both, one could review a package to assess its intentions. In case of being malicious, we report it through the system provided by the registry maintainers.

4 CASE STUDY

We followed our proposed approach to detect malware in npm. At the end of May 2022, we downloaded a snapshot of the registry containing the latest version of more than 1.8 million packages. One month later, we analyzed and classified all the packages that were removed for security reasons from npm to construct a dataset of 75 malware samples.

On Table 1 we show characteristics of each documented malicious cluster. Given the numbers from the dataset, we gathered samples from at least three different malware campaigns. Almost all packages trigger their malicious code during installation. We can summarize the behavior of the samples as follows. **Backdoor** creates a code execution backdoor on victim's machines for further attacks. **Sabotage** tampers, abuses, or destroys systems and computational resources. **Virus** tries to spread and infect other systems with malware. **Stealing** gathers sensitive information from the victim and sends it back to the attacker. This last one being the most common attack pattern.

We defined CodeQL queries to specifically match the behavior seen in each identified cluster, and applied them to our snapshot of the registry to find malware with similar semantics. On Table 1 we show the results obtained. In total, we found 125 malicious packages that were available in the registry. When reporting them, npm determined that almost all violated the open-source terms declared in the ecosystem, and placed a security placeholder in their place. The only exception being `researcher1772`, still available, which is a deprecated package currently supervised by npm.

Table 1: Taxonomy for the dataset of reported malware (# D), and results of applying our queries to the registry (# R).

Query	Trigger	Behavior	# D	# R
dependency-install	Install	Virus	1	0
dependency-save	Install	Virus	48	0
discord-injection	Runtime	Backdoor	1	0
discord-steal	Runtime	Stealing	1	0
discord-ware	Install	Sabotage	1	0
other-request	Install	Unknown	1	0
other-shell	Runtime	Unknown	1	0
theft-dns	Install	Stealing	15	91
theft-encoded	Install	Stealing	1	0
theft-environment	Install	Stealing	3	11
theft-os	Install	Stealing	1	17
theft-ping	Install	Stealing	1	6
Total			**75**	**125**

Analyzing the results, one could see that we found remnants from four different malware campaigns, all of them having information stealing as objective. This might indicate that this attack pattern tends to be preferred by malicious actors, since the only configuration needed is a place to store all the harvested information, and could lead to future, more elaborate, and highly targeted attacks. Note that the rest of the queries did not find any other sample outside of the collected dataset. Meaning that the removed packages were probably not part of an attack campaign, but one-off malware packages. Besides this, there was an entire campaign caught even before collecting the samples for the dataset. Considering that npm scans the registry in search of malware, it is possible that the analysis set up by the registry maintainers was able to detect these packages.

Lastly, a relevant aspect of our results worth mentioning is the absence of false alarms. This is a consequence of the highly targeted technique that we propose. Instead of designing a robust tool able to detect any potentially malicious package, we only develop queries to specifically match the behavior from recently reported malware. We expect to deal with wrongly matched packages in future iterations of our approach, yet we are focusing on having high precision (reducing the time spent manually reviewing results), possibly at the expense of high recall (potentially missing malicious packages in the registry).

5 CONCLUSION

We have presented a technique, based on static analysis, to leverage recently reported malware for detecting further instances with similar semantics on npm. By monitoring npm for discovered malicious packages, we were able to design specific CodeQL queries to match packages exhibiting the same behavior in the ecosystem. For our study, we developed and applied 12 queries while scanning the registry, reporting a total of 125 available malicious packages from four different malware campaigns, producing no false alarms in the process. We plan to investigate how the generality of the queries affects the performance of the proposed approach, and how to deal with the issue of obfuscation techniques used in npm.

REFERENCES

[1] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conf. Object-Oriented Programming ECOOP.* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25.

[2] Alex Birsan. 2021. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610 Accessed: 2023-03-15.

[3] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2007. .QL: Object-Oriented Queries Made Easy. In *Generative and Transformational Techniques in Software Engineering II, Int. Summer School, GTTSE (Lecture Notes in Computer Science, Vol. 5235).* Springer, 78–133.

[4] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. 2007. Keynote Address: .QL for Source Code Analysis. In *Seventh IEEE Int. Workshop on Source Code Analysis and Manipulation SCAM.* IEEE Computer Society, 3–16.

[5] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annu. Network and Distributed System Security Symposium (NDSS).* The Internet Society.

[6] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *43rd Int. Conf. Software Engineering (ICSE).* IEEE, 1334–1346.

[7] Kalil Anderson Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *Proc. 41st Int. Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER).* IEEE / ACM, 13–16.

[8] Harry Garrood. 2019. Malicious code in the PureScript npm installer. https://harry.garrood.me/blog/malicious-code-in-purescript-npm-installer/ Accessed: 2023-03-15.

[9] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd Int. Symp. Research in Attacks, Intrusions, and Defenses (RAID).* USENIX Association, 121–134.

[10] Andrej Mihajlov. 2018. GitHub Issue 39: Virus in eslint-scope? https://github.com/eslint/eslint-scope/issues/39 Accessed: 2023-03-15.

[11] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2020. If You've Seen One, You've Seen Them All: Leveraging AST Clustering Using MCL to Mimic Expertise to Detect Software Supply Chain Attacks. *CoRR* abs/2011.02235 (2020).

[12] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *17th Int. Conf. Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA).* Springer, 23–43.

[13] Marc Ohm, Timo Pohl, and Felix Boes. 2023. You Can Run But You Can't Hide: Runtime Protection Against Malicious Package Updates For Node.js. *CoRR* abs/2305.19760 (2023).

[14] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. 2022. On the Feasibility of Detecting Injections in Malicious Npm Packages. In *Proc. 17th Int. Conf. Availability, Reliability and Security.* ACM, 1–8.

[15] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious Npm Packages. In *44th Int. Conf. Software Engineering (ICSE).* ACM, 1681–1692.

[16] Ceej Silverio. 2017. 'crossenv' malware on the npm registry. https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry Accessed: 2023-03-15.

[17] Ayrton Sparling. 2018. GitHub Issue 116: I don't know what to say. https://github.com/dominictarr/event-stream/issues/116 Accessed: 2023-03-15.

[18] Matthew Taylor, Ruturaj K. Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. SpellBound: Defending Against Package Typosquatting. *CoRR* abs/2003.03471 (2020).

[19] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS).* ACM, 1755–1770.

[20] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and Combosquatting Attacks on the Python Ecosystem. In *IEEE European Symp. Security and Privacy Workshops (EuroS&P).* IEEE, 509–514.

[21] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in npm with Latch. In *ACM Asia Conf. Computer and Communications Security (ASIA-CCS).* ACM, 1139–1153.

[22] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proc. 28th USENIX Security Symposium.* USENIX Association, 995–1010.