Does This Dependency Matter

A Vulnerability Reachability Detection Prototype for Python Projects

Eldon Lake
Undergraduate Student
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
elake@ualberta.ca

Ridwan Andalib
Undergraduate Student
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
andalib@ualberta.ca

Abstract

The use of open source software (OSS) libraries is a near ubiquitous practice in software development. In order to address the continual release of publicly disclosed vulnerabilities in OSS libraries, developers rely on tools to detect vulnerable dependencies within their projects. While industry standard tools rely on metadata to detect the presence of affected dependencies, this may overestimate the incidence of actual security risks introduced by these dependencies. Recent research has produced code-centric tools that use static and/or dynamic analysis to assess the reachability of these vulnerabilities, and has been shown to greatly reduce the quantity of false positives reported by using metadata alone. This has resulted in open source and commercial solutions for vulnerability reachability that support a growing number of publicly disclosed vulnerabilities and programming languages. To date, none of these open source solutions have extended their static analysis support to Python programs. This paper discusses our approach in building a research prototype using static analysis to assess the reachability of a known vulnerability within a Python library. We detail the implementation of several design choices within our approach that differ in their classification of vulnerable behaviour, and provide an assessment of the tradeoffs between each method. Potential steps for extending this work into a viable industrial scale tool, similar to those existing for other languages, is also discussed.

Keywords: Static Analysis, Python, Vulnerability Disclosure,

1 Introduction

The use of open source components is a near universal practice when it comes to application development. The popularity and availability of open source libraries expands every year. This holds true across all sectors of industry, from enterprise, to government, to non-profit. Open source Java components alone grew to over 50 billion annual downloads in 2016, with no signs of slowing down anytime soon [1]. Despite common conceptions about the inherent security advantages of having millions of potential code reviewers with open source software, up to 80% of presently deployed in-house developed enterprise applications contain versions of open source software with known vulnerabilities[1]. These vulnerabilities can take months, even years to receive fixes, and many are never addressed at all. Even in the best case scenario, where a vulnerability is disclosed and a patch is quickly made available, there can be significant costs to upgrading dependencies in products that have already been deployed. Upgrading dependencies can mean deprecated features, new unintentional bugs, costly downtime, and having to commit additional development resources in order to accommodate changes in software. All of these demonstrate the need to quickly and accurately determine the impact of the vulnerabilities present in a project's dependencies.

To help keep up on these vulnerabilities, many tools and services exist to perform automated dependency checking. These products typically involve using the metadata of a project's source code to identify all of its direct and indirect dependencies, and then checking those dependencies against databases of known vulnerabilities. Examples include Dependency Check by OWASP, RetireJS, the Node Security Project, and many more. An issue present in these types of tools, however, is that a vast majority of them are not code-specific, and may flag dependencies in a project with vulnerabilities that are not actually reachable given the particular use case of the project. Without an easy way to identify these false positives, application maintenance teams may find themselves wasting valuable time and resources accommodating software updates that end up being completely unnecessary.

As early as 2015, research was published demonstrating the feasibility of using both static and dynamic analysis in order to provide dependency vulnerability detection that accounts for the reachability of a published vulnerability based on the specific usage of any given Java project[2]. This included a case study showcasing a single CVE at the time. By 2016 this had been developed by SAP Security Research into an industrial scale tool capable of assessing projects against a large database of disclosed vulnerabilities[3]. In 2018, after having gained significant commercial adoption, an open source version of this tool was released to the public under the name Eclipse Steady. Research published in the summer of 2020 provided a thorough evaluation of Eclipse Steady,

PL'18, January 01–03, 2018, New York, NY, USA
2018.

demonstrating that their code-centric approach could eliminate as many as 88.8% of the vulnerability alerts produced by the industry standard OWASP Dependency Check by demonstrating them to be false positives created by unreachable vulnerabilities[4].

In the past five years, a number of commercial solutions for detecting the reachability of components affected by disclosures in OSS libraries made it to market. Despite this spurred interest in the technique, to date there exist no open source projects that allow for reachability assessment using static analysis on Python programs. Eclipse Steady has introduced some features for Python programs, but as of the time of this writing, this has yet to include their static or dynamic analysis components. Snyk has added Python to their dependency tree checker, but have yet to extend their static reachability analysis features to the language either. In late September 2020, Github made their code checking service available to the public. This has some Python support, but their database of Python queries is currently limited to variant analysis, enabling users to detect when they have inadvertently created a new vulnerability that matches the form of a known vulnerability, but does not actually implement reachability analysis for depending on existing vulnerable libraries.

Our contributions. This gap in the open source space has presented us with a unique opportunity to present a novel case study of our own. Our prototype, Does This Dependency Matter (DTDM), offers a proof of concept artifact capable of assessing the impact of a select disclosure, CVE-2020-14422, on Python projects that depend on the ipaddress library. We use an open source static analysis framework in order to build queries that determine the reachability of the vulnerable portion of the affected library. We examine a variety of design decisions to assess how they impact the results and performance of our analysis, and contrast them with their equivalent design decisions in Eclipse Steady. Finally, we evaluate DTDM by analyzing a number of manually constructed source code snippers, as well as a selection of 11 open source repositories that have either flagged CVE-2020-14422 in their issues tracker as requiring remediation by automated dependency tracking systems, or have commits by developers referencing CVE-2020-14422 in the commit messages.

2 Background

2.1 CVE-2020-14422

CVE-2020-14422 was published on June 18, 2020 and last updated on November 16, 2020. It addresses a bug in Python's ipaddress module that produces constant hash values for IPv4Interface and IPv6Interface objects, which we will refer to collectively as interface objects. This means that every interface object's hash function will produce one of two possible numbers. Ordinarily, the hash function of an object is supposed to produce a unique identifier that is unlikely to match the hash value of another object. This creates the potential to allow an attacker to perform a denial of service (DOS) attack that exploits aspects of python where the runtime complexity is dependent on avoiding repeated hash collisions. The example given in the CVE is a dictionary. Dictionaries are implemented with hash tables. If interface objects are used as the keys for a dictionary object, the repeated collisions will bring operations with an $o(1)$ complexity up to $o(n)$, and operations with an $o(n)$ complexity up to $o(n^2)$. If an attacker was able to create a large number of these entries, it could cause the application to time out, thus achieving the DOS attack.

3 Approach

3.1 Overview

Our approach uses a manual inspection of the CVE in order to identify which aspects of the library are vulnerable, as well as to define what would be considered exploitable behaviour when using the affected library. We explore three different modes of classifying exploitable behaviour, and produce individual analyses tailored to each mode. Our analyses are built using CodeQL, an open source semantic code analysis framework developed at Semmle and subsequently released by GitHub. CodeQL allows us to generate an abstract syntax tree (AST) for any Python project that we wish to evaluate against the chosen CVE. We then design queries to analyze the AST for incidences of vulnerable behaviour as defined by our three modes of classification. Any alert generated by the

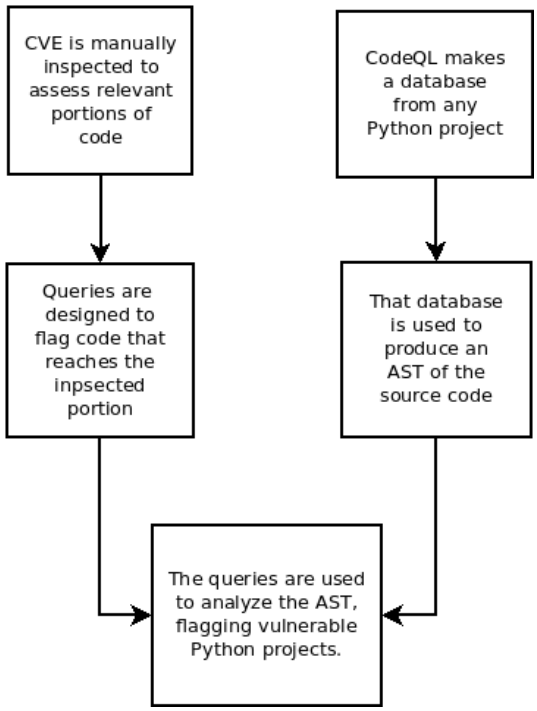

Figure 1. Basic workflow for DTDM

queries indicates a potential vulnerability risk. When this is encountered, a user would then be able to run the relevant query using an existing CodeQL graphical interface in order to identify the location and exact nature of the responsible line(s) of code. A simple workflow diagram can be seen in Fig. 1.

3.2 Classification of Exploitable Behaviour

The authors in Ponta et al.[3] developed an implementation within Eclipse Steady that automatically processed the fix commits related to a vulnerability disclosure in order to automate the identification of vulnerable code within libraries[3]. While their work is open source, this feature does not yet have Python support, and reimplementing the same strategy was beyond the scope of our project. One disadvantage of the Eclipse Steady approach is that simply reaching the affected portion of a vulnerable library may not necessarily be an indication of behaviour that is actually exploitable in practice. In contrast, we decided to manually inspect the CVE in order to discern the relevant details. This allowed us to explore three different design choices with regards to what should be considered exploitable behaviour, providing additional insight into the extent to which the results of an analysis may be impacted by this initial classification. Our three modes of classification are as follows:

safe-mode. This mode will consider any reachable path to the vulnerable code as potentially exploitable, regardless of how that code is used. Of our three modes, this one is the most analogous to the classification provided by Eclipse Steady. Because of the broad inclusion criteria, it was also predicted to produce the lowest rate of false negatives. Finally, this may help to identify potential sources of aberrant behaviour caused by the affected library, even in cases where it may not necessarily indicate a genuine vulnerability. For our chosen CVE, any code that results in a call to the __hash()__method of the IPv4Interface or IPv6Interface classes within the ipaddress module is flagged by this mode.

informed-mode. This mode will only consider use of the vulnerable code that matches the exploit strategy that is specified in the CVE. While this may not always be a feature of every CVE description, it is quite common for a disclosure to include a high level description of how a vulnerability would be exploited. In our example this would be any use of interface objects as keys in the key-value pairs of a dictionary object, as that is how this particular entry suggests the DOS attack would be triggered.

heuristic-mode. This mode will only consider incidents where there is data flow from an external source to the vulnerable library code. This includes HTTP requests and file reads, as they are common candidates for attackers to inject data into the environment, but can trivially be extended to any methods that a developer knows to be potential sources of user data in their project. The rationale behind this mode is that the type of data required to trigger an exploit will often not be encountered in realistic use cases. For instance, an attacker would want to provide a large number of illegitimate entries for interface objects. If the entries are only coming from legitimate sources, most projects would never use enough unique interface objects to encounter the performance issues required for this exploit.

Python Statement	Operation Variant
aDict[anObject] = 1	Dictionary assignment
aDict = {anObject: 1}	Dictionary literal
aSet = {anObject}	Set literal
aDict = {x: 1 for x in [anObject] }	Dictionary comprehension
aSet = {x for x in [aObject] }	Set comprehension
aDict.update({anObject: 1})	Dictionary update
aSet.add(1)	Set add
aDict = dict(anIterator)	Dictionary constructor
aSet = set(anIterator)	Set constructor
hash(anObject)	Explicit call

Table 1. Operation Variants for Predicates

3.3 Building CodeQL Queries For Analysis

Once the modes of classification had been established, we proceeded to build queries that would allow CodeQL to identify any AST nodes that met the description of each mode. Each query includes a single QL statement that must identify all of the operation variants that match the classification. In order to achieve this, queries are built from a number of predicates, each of which encapsulate the logic for a single operation. While every query is necessarily unique, there is considerable reuse among the internal predicates of the three queries. The implementation details for each mode are as follows:

safe-mode: As mentioned, a query for safe-mode must identify every call to the hash function of the interface objects. This is a non-trivial task, as the AST provided by CodeQL does not extend into the implementations of member operations, where a majority of these calls are found. ie. for the assignment statement in the first row of Table 1, the AST would not extend into aDict.setitem(self, anObject, 1) or the subsequent call to hash(anObject), where the private hash method of anObject would be accessed. Accordingly, this required us to identify all of the calls to hash() present in the Python standard libraries that were not in declarations of private hash functions. We used redundant sources for this, to ensure we had as full coverage as possible. First we

consulted the python data model reference. This states that unless a developer has explicitly invoked the built-in hash(), it is only called by operations on hashed collections, which includes dictionaries, sets, ordered dictionaries, and frozen sets. For brevity, we will refer to them simply as dictionaries and sets. To confirm these findings, our redundant source involved using CodeQL to analyze the standard libraries themselves. This produced only one exception to the data model reference, a call to hash in the LRU cache decorator in functools.py. As this was a niche case, it was omitted from our analysis. Any situation where an interface object's private hash() function is called would therefore require encountering one of three cases: (1) hash() is explicitly called on that object (2) that object is added to a set or (3) that object is added to a dictionary. Operation variants for these cases include calls to hash() and adding to a set or dictionary via a constructor, literal, comprehension, assignment statement, or add / update method. A summary of these variants can be seen in Table 1. Safe-mode required a predicate for each operation variant. The resulting query will flag any node that matches any of the predicates. As of the time of this writing we have yet to produce a fully functional predicate for detecting all the instances of interface objects added through dictionary comprehension, but have had success with all of the other variants, and hope to remedy that in the future.

informed-mode: No additional predicates are required for informed-mode, it uses only the dictionary predicates already constructed for safe-mode. Given that sets and dictionaries share similar implementations in Python, we can see right away that the DOS risk posed by dictionaries is equally relevant to sets, which will be reflected in the results of the evaluation.

heuristic-mode: CodeQL includes a library for easily implementing taint flow in Python. In order to implement heuristic-mode, we used qualified method names to identify potential sources where attackers could provide data. Taint sinks were identified using the same predicates produced for safe-mode. File reads and HTTP requests were our default taint sources, but as previously mentioned, users would be wise to provide their own sources of taint based on their knowledge of their projects. For this reason, heuristic-mode is the only analysis that could not be run optimally in a completely automated fashion for users that wish to analyze their own projects with our artifact. Finally, we have encountered implementation details that have prevented our heuristic query from running properly. It is built out, and we believe that it is based on sound theory, so we have included it in our artifact, but have excluded it from our evaluation. It is still our intention to debug the implementation so that we can see for ourselves the differences in performance, but it will not be complete by the end of the submission window for this paper.

	Vulnerable Dictionaries Detected	Vulnerable Sets Detected	Total Runtime (s)
Safe	6/10	20/20	150
Informed	6/10	0/20	142
		Difference:	**5.33%**

Table 2. Artificial Test Results

4 Evaluation

4.1 Experimental Setup

All evaluations were timed on an HP OMEN 880 desktop computer running Windows 10 64-bit with an Intel® Core™ i7-8700K processor and 16GB of memory. Each analysis mode was run with a separate instance of CodeQL so that it would not reuse any cached data from previous analyses on the same repo.

4.2 Artificial Tests

In order to ensure that our queries would accurately reflect the intended behaviour of our analysis modes, we constructed 16 small python snippets that covered basic permutations of the operation variants we discussed: set literals, set constructors, dictionary literals, dictionary assignments, etc. For each variant, one test was created with a tainted data source, and one test was created with a safe data source. Two tests that did not encounter any vulnerable code were also included. Each mode of analysis was performed in batch mode against the grouping of 16 tests, and the total runtime for each mode was recorded. In total the tests had 10 vulnerable lines involving dictionaries, and 20 vulnerable lines involving sets.

As shown in Table 2, safe-mode took a total of 150 seconds to complete, and informed-mode took a total of 142 seconds to complete, a difference of 5.33%. This was a smaller difference than initially expected, given that safe-mode processes more than twice as many predicates. We attributed this to the fact that our artificial tests were very small files, and so a majority of the runtime came from the CodeQL overhead. Our expectation was that when we collected some larger project repos to analyze that these differences would become more pronounced.

Because the high level description of the exploit provided by the CVE only mentioned dictionaries and not sets, the informed-mode version ended up missing a majority of the potentially exploitable snippets that were included in our artificial tests. Additionally, as previously mentioned we have yet to achieve a fully functional predicate for dictionary comprehension, meaning that safe-mode was only able to detect 26 out of 30 cases, a recall rate of 86.6%. By comparison, however, informed-mode caught only 6 out of 30 cases, a recall rate of only 20%, which provides evidence for the potential downsides of depending on natural-language

Application	Repo Size	CVE Reporter
thin-egress-app	790.5kB	bot
first-django-app	203.9kB	bot
AID	3.4MB	bot
eventpoints-backend	588.4kB	bot
scrapy-cookies	155.5kB	bot
another-ldap-auth	30.1kB	dev
SpeedTestLogger	12.6kB	bot
py_telegram_popot_bot	8.5MB	bot
YouTubeReviewBot	106kB	bot
serverless-transformers	10.4kB	bot
scrapy-pipelines	196.4kB	bot

Table 3. Real world test repo data. CVE Reporter 'bot' indicates an automated flag, 'dev' indicates a commit message by a developer.

Application	Safe (s)	Informed (s)
thin-egress-app	13.95	12.87
first-django-app	13.95	12.52
AID	18.29	14.74
eventpoints-backend	14.14	13.30
scrapy-cookies	16.55	15.38
another-ldap-auth	13.18	12.91
SpeedTestLogger	14.72	13.06
py_telegram_popot_bot	20.37	17.15
YouTubeReviewBot	14.27	13.17
serverless-transformers	14.17	13.16
scrapy-pipelines	16.05	14.10
Average:	**15.42**	**13.85**
Difference:	**10.19%**	

Table 4. Runtime comparisons for safe and informed mode on real world test repos, seconds.

vulnerability descriptions, something that Plate et al. had already suggested may be error-prone [2].

4.3 Real World Tests

In order to see if our prototype could improve on the precision of metadata based vulnerability trackers, as well as developer flagged concerns, we searched for public GitHub repos that had added posts to their issue tracker which flagged the need to update explicitly due to CVE-2020-14422. All of these issue tracker posts were created automatically by tracker bots. We also found one repo where a developer included a reference to CVE-2020-14422 in a commit message, indicating it as a reason for updating. Every repo was downloaded and analyzed by each of the analysis modes, providing a better indication of the performance issues between each mode, with an average 10.19% reduction in runtime going from safe-mode to informed-mode, and a maximum reduction of 19.4% for the second largest repo. We also manually inspected each repo to see if there was any potential impact from the CVE, and determined that all 11 repos were false positives. Our prototype successfully marked every repo as a false positive. This is a small sample size, but it is technically an increase in precision from 0% to 100%. Unfortunately we were unable to find any public repos which were both flagged as vulnerable and actually had potentially exploitable instances of the interface objects. Tables 3 & 4 are included to show the repo names, sizes, runtimes, and how the repo was marked as vulnerable. Given more time, we would have liked to have found some examples that encountered the vulnerability in an exploitable manner, and manually tested them with an existing python vulnerability checker.

5 Discussion

Our evaluation has shown the potential for static reachability analysis to provide increased precision for vulnerability tracking in Python libraries, similar to results found previously by researchers using tools for other languages. Our approach allowed for the comparison of multiple classification methods. This contrasted a method similar to existing strategies that consider all of the code addressed by a fix commit, to one that made use of the high level descriptions often available in disclosures to help further narrow the scope of what to consider during an analysis. While the latter method did produce some moderate performance gains, especially on larger repositories, it ultimately resulted in a loss of recall that would make it hard to justify based on the performance differences alone.

As mentioned previously, there are a number of features that we were not able to complete in the window of time we had to work on this project. This includes providing a more comprehensive predicate for detecting type sensitive dictionary entries added through comprehension, as well as debugging the taint flow issue in our heuristic-mode. Additionally, we had been investigating two additional CVE choices that we had hoped to include today, but other time commitments in our undergraduate careers prevented us from getting them built out on time.

While we were not able to achieve everything we had set out to do at the inception of this project, we still believe that we have demonstrated the viability of our original goal: extending recent advancements in vulnerability assessment to the Python language. If we were to attempt to extend this project to the point of building a commercially viable industrial scale tool, similar to the fate of other research we have made reference to, there are a number of lessons learned here that would inform our decision of how to proceed. Most notably, we would attempt to improve our method of classification to not depend on any expert knowledge of the disclosures in question. This design choice not only reduced the effectiveness of our analysis, but doing so makes

it a far less scalable approach, which would require either a dedicated team of engineers to add support for new disclosures as they are published, or a community driven effort to collectively grow and maintain a database of supported vulnerabilities.

6 Related Work

Our paper makes repeated references to Eclipse Steady, and three published works[2–4] that follow its progress from being a research prototype in 2015 with a case study featuring a single CVE, to a fully fledged open source suite in 2018 with widespread adoption. While this is arguably the most closely related body of work, especially with regard to motivation, there are a number of dramatic differences in scope beyond the choice of supported languages that could be addressed further. One of the main examples of innovation in Eclipse Steady has to do with their novel approach to classifying the elements of a library considered to be relevant to vulnerabilities. This is done in a systematized and largely automated fashion, requiring only a small number of metadata details to be maintained for each vulnerability, and using changes in the fix commits listed by the metadata to generate the necessary constructs to perform this classification. This is in addition to the reachability analysis performed on the actual projects whose dependencies are being considered. While we are still performing static analysis for reachability, our classification process is entirely manual and requires a high level understanding of any CVE we wish to support. As mentioned previously, this reduces scalability and introduces potentially critical errors. Lastly, our reachability analysis is a strictly static analysis, whereas their approach mitigates the downsides of static and dynamic analysis by using a combination of both.

While we have touted scalability as a potential barrier to expanding this technique into a practical tool, it is worth noting that community driven initiatives to build databases of manually written queries for vulnerability detection is not unheard of, and has been amply demonstrated by the CodeQL community since its open source debut. The evidence of this can be seen in the CodeQL query libraries, as well as in the work of the GitHub Security Lab. While the security-relevant work driven by the CodeQL community has tended to focus around the aforementioned application of variant analysis, it still provides a conceptual analog demonstrating that such a thing could be possible for dependency management as well. This could be considered related work insofar as it pertains to the kind of collaborative effort by security enthusiasts that would be required to scale our approach. If we too had the reach of the largest source code host in the world, perhaps it would be a more practical option.

7 Conclusion

We demonstrated a prototype for analyzing the reachability of the vulnerability present in a select CVE in Python. Our prototype was capable of accurately categorizing a number of public repos as false positives, that had been marked as vulnerable by both existing popular dependency tracking systems, as well as one instance of a manual flag by the repo's developer. This extended concepts first demonstrated by other researchers into a language for which there is currently no equivalent open source alternative. We explored the use of different design choices that differed from the approach by other leaders in the field, ultimately leading us to conclude that their design choice was quite sound. In the future we plan on improving the implementation of our analysis, with intentions of then expanding it to support additional disclosures in the near future.

Acknowledgments

This material was produced by pupils under the guidance of Karim Ali and Ifaz Kabir, over the course of an extremely satisfying semester's worth of CMPUT 416 / CMPUT 500 at the University of Alberta. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their instructors or other faculty members. Thanks for a great year, this was an awesome learning experience.

References

[1] Sonatype Inc. 2020. Sonatype 2017 State of the Software Supply Chain Report. https://www.sonatype.com/sonatype-2017-state-of-the-software-supply-chain-report-reveals

[2] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 411–420. https://doi.org/10.1109/ICSM.2015.7332492

[3] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 449–460. https://doi.org/10.1109/ICSME.2018.00054

[4] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (June 2020), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x

A Appendix

Artifact release: https://www.github.com/elake/dtdm