Advisory: Vulnerability analysis in software development project dependencies

Germán Márquez
Dept. of Computer Languages and Systems
University of Seville
amtrujillo@us.es

José A. Galindo
Dept. of Computer Languages and Systems
University of Seville
jgalindo@us.es

Ángel Jesús Varela-Vaca
Dept. of Computer Languages and Systems
University of Seville
ajvarela@us.es

María Teresa Gómez López
Dept. of Computer Languages and Systems
University of Seville
maytegomez@us.es

David Benavides
Dept. of Computer Languages and Systems
University of Seville
benavides@us.es

ABSTRACT

Security has become a crucial factor in the development of software systems. The number of dependencies in software systems is becoming a source of countless bugs and vulnerabilities. In the past, the product line community has proposed several techniques and mechanisms to cope with the problems that arise when dealing with variability and dependency management in such systems. In this paper, we present *Advisory*, a solution that allows automated dependency analysis for vulnerabilities within software projects based on techniques from the product line community. *Advisory* first inspects software dependencies, then generates a dependency graph, to which security information about vulnerabilities is attributed and translated into a formal model, in this case, based on SMT. Finally, *Advisory* provides a set of analysis and reasoning operations on these models that allow extracting helpful information about the location of vulnerabilities of the project configuration space, as well as details for advising on the security risk of these projects and their possible configurations.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Requirements analysis*; *Software design engineering*; *Software implementation planning*.

KEYWORDS

Software Project,

Library,

Dependency,

Vulnerability,

CVE,

Security,

Verification,

Risk,

Impact

1 INTRODUCTION

Software projects usually delegate a large part of the functionality to external libraries, which means that vulnerabilities in these libraries can affect the project under development. Nowadays, multiple vulnerabilities are identified every day [8] that must be known and managed quickly by developers. We are aware that the cyber-attack chains used by attackers to penetrate systems are becoming increasingly sophisticated [12]. Thus, attackers can make versions of dependencies with known vulnerabilities, such as the recent vulnerability CVE-2021-44228[1] detected in Log4j[2], which has affected at least 186,352 projects in the Java ecosystem [7] due to the complexity of the analysis of its dependencies. Therefore, a misconfiguration (according to OWASP Top-10 vulnerabilities) in a software component (dependency) can be used as an entry point (attack vector) for an attacker. Due to the wide variety of dependency configuration options, it is a challenge to analyse the possible vulnerabilities of a software project [3][9][10].

Variability-intensive systems (VIS) are those software systems that, to function correctly, must manage and deal with a large number of dependencies [4][5]. In the literature, we find projects with hundreds of dependencies and configuration options, such as the Linux Kernel with more than 10^{60} different configurations. A configuration is defined in this context as a valid combination of versions of the libraries and software artefacts on which a project is dependent. This number of dependencies and libraries makes it difficult for developers to be aware of which vulnerabilities affect the software they are developing and how to take measures to mitigate security risks.

For example, the Requests[3] HTTP call package is used in approximately 1,390,000 repositories and 66,000 packages[4], which gives an idea of its impact on other projects. We can see that Requests in turn depend on urllib3 and flask. From now on, we will use Request, urllib3 and flask, and assume a dependency file that defines the following dependencies: $urllib3q1.21.1$ and $urllib3 < 1.27$, and $flask > 1.0$ and $flask < 2.0$. If we analyse the configuration space, we can see that there are a total of 29 possible versions for urllib3 and 9 for flask, which means a total of 261 possible version configurations. Furthermore, among the 261 configurations, we detect that at least one vulnerability is associated with one of the dependencies.

[1] https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[2] https://logging.apache.org/log4j/2.x/
[3] https://docs.python-requests.org/
[4] https://github.com/psf/requests/network/dependents

In other words, if the developer were to randomly choose a configuration from among the 261, approximately 83% of the time he would be assuming a security impact in his configuration. In other words, the configuration he chooses could have a vulnerability, and therefore an impact on his security. We take this as the motivating example for the rest of the paper.

Given the difficulty of analysing dependencies manually, the product line community proposed the automatic analysis of variability in VISs, for example with techniques such as AAFM (Automated Analysis of Feature Models) [6]. AAFMs enable reasoning about VISs by using artificial intelligence systems or ad-hoc algorithms to extract relevant information from the set of dependencies described in a VIS. Coupled with variability analysis, approaches have emerged that attempt to analyse the vulnerabilities of a software product line to optimise the test suite to be performed [11].

In this article, we present Advisory, a solution that allows to analyse and reason about the complete configuration space of the dependencies of a software project taking into account its vulnerabilities. To this end, Advisory is presented as a solution for, modelling the dependencies, i.e. the configuration space of a software project, and attributing that configuration space with security information related to its vulnerabilities. Enable techniques and operations that allow reasoning about the dependency space of a software project taking into account security information related to vulnerabilities (e.g. not using Log4j). And all this throughout the development and evolution of a software project.

The rest of the article is organised as follows: Section 2 presents Advisory, a solution for software project analysis that takes into account dependencies and vulnerabilities that may affect the project. Finally, the 3 section presents our conclusions, lessons learned and future works.

2 ADVISORY: AN AUTOMATED SCANNING-BASED TOOL FOR VULNERABILITY DETECTION

Fig. 1 shows an overview of the process supported by Advisory. The process is divided into four main components: a) Extracting the dependency graph, e.g., from requirements.txt file of a Python project, we extract its dependency graph using information from GitHub[5]; b) Attribute the graph with vulnerability-related information from the NIST NVD vulnerability database; c) Encode the information into a formal model based on an SMT solver [1], which allows us to reason about the dependencies and their vulnerabilities, and finally; d) Apply a set of operations to facilitate the analysis of the dependency information and its vulnerabilities.

2.1 Extract the dependency graph

In the first component (Extract) we build the dependency graph of a software project hosted in a repository. This process consists of the following steps: 1) we get the information from the repository; 2) we filter the valid versions for those dependencies; 3) we build node by node the dependencies of the graph. As described in more detail below:

- First, we get the information about the dependencies from the code repository of the software project, extracting it automatically. In case the dependencies have other (indirect) sub-dependencies, a recursive process will be performed, to a given graph depth. Currently, Advisory can obtain this information from repositories hosted on GitHub and of a Python nature. To do so, it relies on GitHub's GraphQL API. In the case of the motivating example in the previous section, a single call has been made to the repository containing the files with the dependencies, in this case, urllib3 and flask.
- We will then use the restrictions to filter the valid versions. We filter them one by one and choose the ones that satisfy the constraints in the dependency file. For example, we are working with projects of a Python nature, so we use the Python Package Index (PyPI)[6] to extract all versions of these dependencies, and then filter them according to the constraints in the dependency file. For the ones in the motivating example, we get the following versions: flask={1.0.1, 1.0.2, ..., 1.1.3, 1.1.4} and urllib3={1.21.1, 1.22, ..., 1.26.8, 1.26.9}.
- Finally, with this information and the filtered versions, we build a new node of the graph for each dependency and each version, and we add the directed arcs to relate each dependency node with those of its version, as well as include other relationships with parent and child nodes. We will do this for all extracted dependencies. Note that given the more than likely combinatorial explosion of dependencies, it will be necessary to specify a depth level of the graph. Starting with the root which will be our project (Request for example) which will have depth 0, we make a call that will construct both the root node and all nodes of depth 1, which will be our project's dependencies (or direct dependencies). The graph at level 1 of the motivating example would look like in Fig. 2. Note that in the figure we have added information on the vulnerabilities that will be detailed below.

2.2 Attribute the dependency graph

We will now describe how the nodes of the graph, i.e. the dependencies, are attributed with information related to the vulnerabilities (CVE). For this process, we currently use the NIST NVD database API. First of all, we will use the name of the dependency as a key to finding those vulnerabilities when performing the searches. From these searches, we extract all the CVEs that include the name of the dependency in one of their CPEs. In the case of flask and urllib3, NVD returns 2 and 8 CVEs respectively associated with those dependencies[7].

These CVEs may be associated with a version or not included in our graph, so we will have to analyse if any of the versions of the dependencies are included in any CPE of these CVEs, otherwise, that CVE should not be assigned to any version of our graph. For example, for flask, none of the 2 detected CVEs will be associated because it does not match any version included in its CPEs. Also, for urllib3, 7 out of 8 CVEs are associated with valid versions of the motivating example. We do this because making individual requests for each version would slow down the attribution process because each query is a request to the NVD API and would be too time-consuming.

[5] https://docs.github.com/es/graphql

[6] https://pypi.org/

[7] These data are those obtained at the time of writing this article.

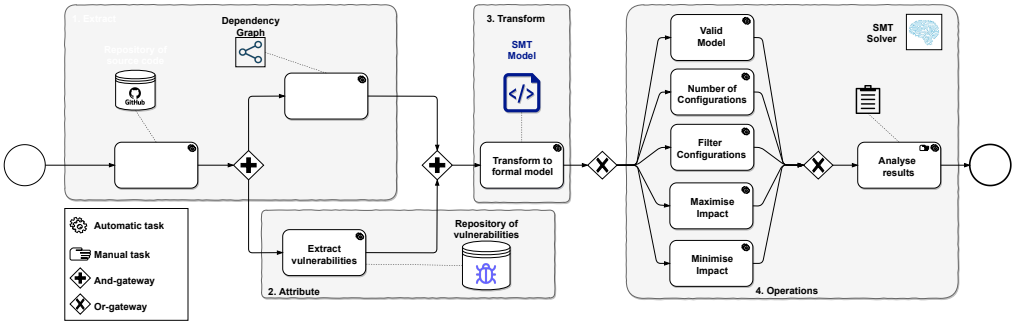

Figure 1: Process supported by Advisory.

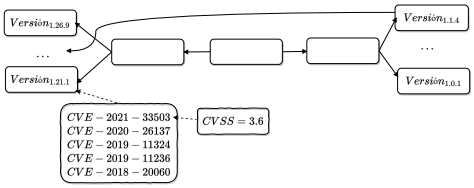

Figure 2: Attributed dependency graph

In this way, we relate the versions of the dependencies with the CVEs that we have extracted and that directly affect them. In addition, we store for each CVE the CVSS (impact) value, as well as the set of metrics of its attack vector. For example, we can see the associated CVEs for urllib3 dependency 1.21.1 and the CVSS impact of one of them in Fig. 2. This impact also comes with the CVE and will be fundamental to determining the impact of the configuration space.

2.3 Transform into SMT model

Finally, the attributed graph will be transformed into a Satisfiability Modulo Theories (SMT) model. An SMT model is a formal generalised constraint satisfaction (SAT) model that allows the use of more complex formulas involving real numbers, integers and/or various data structures such as lists, arrays, bit vectors and strings. For our approach, an SMT will be defined by the tuple: $\langle D,\ V,\ DC,\ Vul,\ f_d(D,\ V),\ f_t(I)\rangle$. Where:

- D is the set of nodes in the graph and will represent the set of variables in the model. For a dependency $d_i \in D$ if $d_i = 0$ it means that it is not selected for analysis, and if $d_i = 1$, it is selected. If we choose flask and urllib3, both dependencies must appear in the SMT model as variables.
- V is the set of version values of each dependency chosen from the graph. The dependency can only take the values of the versions it has available in the graph. As mentioned, if flask and urllib3 are chosen, they must take as value one of the valid dependencies we saw in the previous section.
- DC represents the restrictions that we apply to the versions of each dependency of the graph. In this case, since these are numeric variables, we can make use of logical range operations. For example, for the case of urllib3 in the motivating example, $urllib3 \geq 1.21.1$ and $urllib3 < 1.27$.
- Vul is the set of impact values, i.e. the impact defined in each CVSS associated with a CVE.
- $f_d(D, V)$ is a function that allows us to calculate the impact for each dependency chosen from the graph depending on the version configuration taken by the solver. In our case, we are going to consider all the CVEs with their impacts (CVSS), for which we must add this information for each node. To do this, we can use different functions that calculate the impact of the dependency by aggregating the hits, for example, we can use the mean, the median or the mode. If we were to use the mean to determine the impact of the dependency d_x on the version v_s that has n vulnerabilities (CVE), it implies that if the dependency takes that version, it would be such that for each dependency the calculation of the impact would be: $Impact_{(d_x,v_s)} = \frac{\sum_{i=1}^{n} CVSS_{CVE_i}}{n}$
For example, if the urrlib3 dependency were to take version 1.21.1, the impact would be calculated as follows:
$$\frac{CVSS_{CVE-2021-33503}+...+CVSS_{CVE-2018-20060}}{5}$$
- $f_t(I)$ is a function that calculates the total impact of a project by adding the impact of all dependencies, i.e. if our project had for m dependencies each with its versions it would look like this: $Impacto_{total} = \frac{\sum_{i=1,j=1}^{m,s} Impacto_{(d_i,v_j)}}{m}$
Like the previous function, we can use different functions to calculate the impact of the dependency such as mean, median or mode. For the motivating example, using a mean, the total impact would look as follows: $\frac{Impacto_{urllib3}+Impacto_{flask}}{2}$

To apply this transformation in the current version of Advisory we have used models for the Z3 solver[8].

2.4 Operations

Once we have built the SMT model from the graph, we can apply reasoning operations. Among them, the ones currently implemented by Advisory are:

- **Valid model:** to know if the model can find a solution that satisfies all the constraints and operations created. This operation would return a boolean $True$ or $False$ indicating that the project is valid or not. The motivating example is valid since there is at least one configuration, for example, the one composed by {urllib3 = 1.21.1, flask = 1.0.1}.

[8]https://github.com/Z3Prover/z3

- **Number of configurations:** if the model is valid, extract the number of possible successful configurations. Note that this operation is not feasible to execute in projects with many dependencies as long as we use SMT solvers. The maximum number of configurations that Advisory can return is the number of configurations representable in an integer of Z3 in Python. Defined by the function *sys.maxsize()*, a total of 922.333.372.036.854.775.807 configurations.
- **Analysis of configurations:** if the previous operations do not provide us with sufficient information on the security impact of our dependencies. We have developed the following operations that allow us to refine the analysis of a configuration. These operations are the following: a) **Filter configurations by a minimum and a maximum threshold**, which allows us to create a range of total impact on the configurations. For example, get all configurations whose impact is between 0 and 1.5. If not specified, by default the minimum is set to 0 and the maximum is set to 10, which are the maximum and minimum impact values for a vulnerability according to CVSS. For example, in the case of the motivating example, setting the maximum threshold to 0 would give a total of 45 configurations. This operation returns a set of configurations; b) **Minimise or Maximise** if we want to get the configurations with the lowest impact or the configurations with the highest impact, we can apply one of the two optimisation operations for the impact. For example, in the case of the motivating example, we could maximise the impact to get the configuration $\{urllib3 = 1.22, flask = 1.1.3\}$ with a maximum impact of 1.83. These operations return a set of configurations; c) **Limit scan.** These three operations have a parameter to limit the number of configurations that can be returned by Advisory. Currently, analysing the number of valid configurations for satisfiability problems becomes, in cases where there is a lot of combinatorics, a non-deterministic problem in time (NP-complete). So we need to limit the number of configurations we want to receive. For example, we could get 3 configurations that meet the maximum threshold of 2.5 or minimise the impact and return the 3 configurations with the lowest impact.

Once the solver solves the model together with the required operation, the solver will return the results in terms of propositional logic, that is, at the level of assigning values to each of the variables, and Advisory then interprets the results and presents them as dependencies and versions to be used to meet the objectives set by the user. Note that the different optimisation functions are implemented in propositional logic within the solver itself (Z3).

3 CONCLUSIONS AND FUTURE WORK

In summary, we have learned the following important lessons: 1) **Bridging variability and security**. We can analyse the variability found in the dependencies of software projects and extract information regarding their security; 2) **Evolution of dependency management**. The need to manage our software's dependencies is becoming more critical due to the increasing use of software dependencies and the possibility that their versions may not be secure. and 3) **Inability to analyse the entire configuration space**. The analysis of a very large configuration space makes it impossible for a solver to analyse all possible configurations. This is an open problem for the complete security analysis of a tool with a large number of dependencies and versions.

The development of Advisory leads us to the solution described in section 2. Still, this solution has a series of future works that proposes to extend the tool to give more support to the general purpose of vulnerability analysis in the evolution of software development projects. These works are:

- **Develop new operations**: We will implement new operations that give the person using the tool information relevant to the security of their project. One of these future operations is to analyze the technical debt of our project's dependency configurations. This includes the use of new AI systems to extract relevant information.
- **Variability reduction**: In the product line community, it is well known that increased variability in a domain is problematic [2]. Because it makes it impossible to analyze the entire configuration space, we design a configuration return limit for existing operations. This method might not be the most suitable. We will implement other strategies that prune the configuration space with the restrictions the user wants, e.g., the three most recent versions per dependency.
- **Incompatibility between files**: Each dependency file may represent a different environment (production, development, test, etc.). We will implement the ability to analyze whether these files are compatible with each other to determine if the different environments in which developers work have incompatibilities or problems due to restriccions for example.

ACKNOWLEDGMENTS

This work has been funded by the projects AETHER-US (PID2020-112540RB-C44/AEI/10.13039/501100011033), COPERNICA (P20_01224) and METAMORFOSIS (US-1381375).

REFERENCES

[1] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2015. Generating tests for detecting faults in feature models. In *2015 IEEE 8th ICST*. IEEE, 1–10.
[2] Jan Bosch. 2018. Towards a new digital business operating system: Speed, data, ecosystems, and empowerment (keynote). In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2–2. https://doi.org/10.1109/SANER.2018.8330190
[3] Shuvalaxmi Dass and Akbar Siami Namin. 2020. Vulnerability Coverage for Adequacy Security Testing. In *35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. ACM, New York, NY, USA, 540–543. https://doi.org/10.1145/3341105.3374099
[4] José Angel Galindo Duarte. 2015. *Evolution, testing and configuration of variability systems intensive.* Ph. D. Dissertation. University of Rennes 1, France.
[5] José Angel Galindo, David Benavides, and Sergio Segura. 2010. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In *ACOTA, Belgium, September, 2010*, Vol. 688. CEUR-WS.org, 29–34.
[6] José A Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433.
[7] Wenhui Hu, Yu Wang, Xueyang Liu, Jinan Sun, Qing Gao, and Yu Huang. 2019. Open source software vulnerability propagation analysis algorithm based on knowledge graph. In *2019 IEEE SmartCloud*. 121–127.
[8] Jeffrey R. Jones. 2007. Estimating Software Vulnerabilities. *IEEE Security Privacy* 5, 4 (2007), 28–32. https://doi.org/10.1109/MSP.2007.81
[9] P.V.R. Murthy and R.G. Shilpa. 2018. Vulnerability Coverage Criteria for Security Testing of Web Applications. In *2018 ICACCI*. 489–494. https://doi.org/10.1109/ICACCI.2018.8554656
[10] Salvador Martínez Perez, Valerio Cosentino, and Jordi Cabot. 2017. Model-based analysis of Java EE web security misconfigurations. *Comput. Lang. Syst. Struct.* 49 (2017), 36–61. https://doi.org/10.1016/j.cl.2017.02.001
[11] Ángel Jesús Varela-Vaca, Rafael M. Gasca, Jose Antonio Carmona-Fombella, and María Teresa Gómez López. 2020. AMADEUS: towards the AutoMAteD secUrity teSting. In *24th ACM SPLC '20, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*. ACM, 11:1–11:12. https://doi.org/10.1145/3382025.3414952
[12] Tarun Yadav and Arvind Mallari Rao. 2015. Technical Aspects of Cyber Kill Chain. In *Security in Computing and Communications*. 438–452.