Erschienen in: 2021 IEEE Symposium on Visualization for Cyber Security (VizSec). - Piscataway, NJ : IEEE, 2021. - S. 79-83. - ISBN 978-1-66542-085-3
https://dx.doi.org/10.1109/VizSec53666.2021.00014

VulnEx: Exploring Open-Source Software Vulnerabilities in Large Development Organizations to Understand Risk Exposure

Frederik L. Dennig[1], Eren Cakmak[1], Henrik Plate[2], and Daniel A. Keim[1]
[1]University of Konstanz, Germany* [2]SAP Security Research, France†

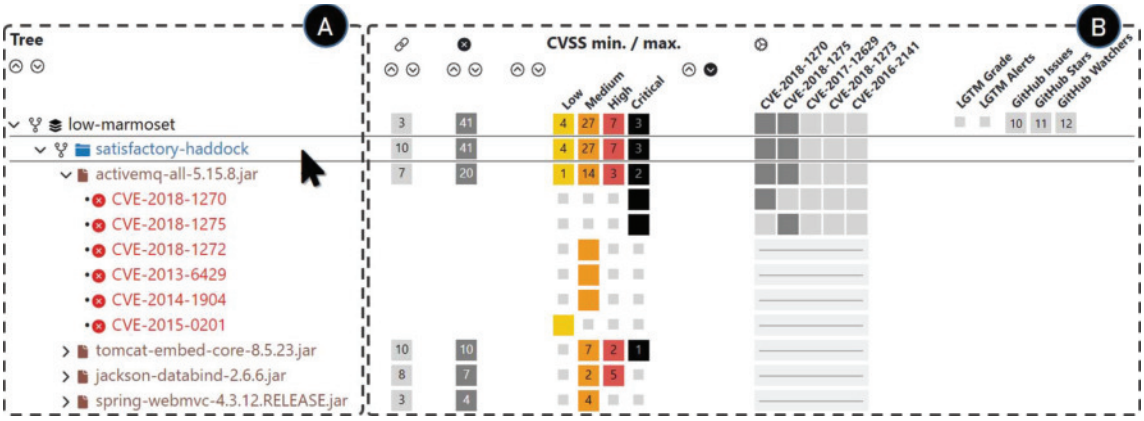


Figure 1: VULNEX is a tool for the investigation of exposure to open-source software vulnerabilities on an organization-wide level. The tool shows repositories, modules, libraries, vulnerabilities in a tree representation (A), and meta-information about each entry (B), such as the CVSS score. We can see that the "low-marmoset" repository is exposed to severe vulnerabilities, three critical and seven high. Two of the critical vulnerabilities are originating from the *activemq-all* indicating that the library should be updated swiftly.

ABSTRACT

The prevalent usage of open-source software (OSS) has led to an increased interest in resolving potential third-party security risks by fixing common vulnerabilities and exposures (CVEs). However, even with automated code analysis tools in place, security analysts often lack the means to obtain an overview of vulnerable OSS reuse in large software organizations. In this design study, we propose VULNEX (Vulnerability Explorer), a tool to audit entire software development organizations. We introduce three complementary table-based representations to identify and assess vulnerability exposures due to OSS, which we designed in collaboration with security analysts. The presented tool allows examining problematic projects and applications (repositories), third-party libraries, and vulnerabilities across a software organization. We show the applicability of our tool through a use case and preliminary expert feedback.

Keywords: Software security visualization, application security, open-source, vulnerability exposure analysis, software auditing.

1 INTRODUCTION

The extensive usage of open-source software (OSS) nowadays promotes a straightforward integration of common software features into existing applications [1,20]. However, software reuse also poses a significant risk as software with disclosed vulnerabilities is often extensively reused, affecting various applications across whole organizations [7]. For instance, the Equifax data breach in 2017 resulted from a missed OSS package update and led to the disclosure of the private data of over 145 million U.S. citizens [16]. Hence, an organization's governance or audit system must identify the organization's overall exposure to OSS vulnerabilities.

Developers and security analysts regularly utilize automated code analysis tools to identify vulnerabilities and investigate the mitigation of OSS security risks. For example, static [25] and dynamic code analysis [21,22] are applied to execute the developed code and detect inherent vulnerabilities. However, such code analysis tools heavily differ in their detection capabilities. They often only store the vulnerability metadata as text files that do not meet software developers' basic requirements, such as prioritizing the most severe vulnerabilities. Assessing the impact of software vulnerabilities is essential for organizations since the effects of exposures can vary significantly. Further, code analysis tools are usually used for single software applications and do not show the impact of OSS vulnerabilities across multiple applications in whole software organizations. Additionally, it is crucial to evaluate the quality of libraries and other dependencies if they originate from another source, such as that the source can be trusted [24], and that OSS developers are swift in addressing vulnerabilities [2]. The mentioned points are crucial for deciding whether a software development organization should use an OSS library.

We propose VULNEX (Vulnerability Explorer), a new tailored design to explore and assess the mitigation of OSS vulnerabilities for auditing and governance of whole software development organizations looking beyond individual applications and teams. In our user-centered design study, we designed three complementary table-based representations to identify and assess vulnerabilities across various applications. We demonstrate the applicability of our approach through use cases and initial expert feedback. VULNEX is open-source[1] and accessible online[2]. With this work, we present a first study to improve the analysis and mitigation of software vulnerabilities, especially from an organization-wide perspective. In

*e-mail: {frederik.dennig, eren.cakmak, keim}@uni-konstanz.de
†e-mail: henrik.plate@sap.com

[1]https://github.com/dbvis-ukon/vulnex
[2]https://dennig.dbvis.de/vulnex

summary, the primary contributions of this paper are: (1) A design study with problem characterization, findings, and lessons learned for the visual analysis of OSS vulnerabilities. (2) The interactive VULNEX analysis tool to interactively explore critical vulnerabilities. With this work, we hope to improve the analysis and mitigation of software vulnerabilities by addressing the need for an analysis tool for auditing entire software development organizations.

2 RELATED WORK

Software visualizations provide a comprehensive overview of complex systems, such as program structures, execution behavior, and the development process [10]. These visualizations are also useful to investigate security aspects, e.g., SecSTAR [11] automatically generates execution diagrams to examine, debug, and test software applications. For an overview of software visualization research, refer to the reviews of Wagner et al. [26], and Chotisarn et al. [9].

In software security visualization, some approaches for vulnerability exploration have been proposed. Harrison et al. [14] proposed the Nessus vulnerability visualization (NV) to discover and analyze network vulnerabilities of Nessus scans. The system simplifies and displays vulnerability assessment results to support security analysts, using zoomable treemap visualization with linked histograms. In a similar context, Angelini et al. [5] proposed Vulnus, which aims to increase situational awareness of security managers by visually analyzing vulnerability spreads in computer networks. Furthermore, CVExplorer [19] is a visual analytics system for analyzing vulnerability reports and enhancing network security using three linked views. These vulnerability systems differ from our approach since they primarily focus on exposing computer network vulnerabilities. Moreover, Goodall et al. [13] proposed a system to explore vulnerabilities and code weaknesses in software development. The goal is to help users understand their code's security status by displaying code vulnerabilities using an aggregated block metaphor for each file. Goodall et al. [13] approach focuses on identifying false positives, which we reduce in our application by checking whether third-party vulnerabilities are reachable. Assal et al. [6] presented Cesar, a collaborative code analysis system to reduce vulnerabilities and improve code security. The authors utilize a treemap visualization to help security experts and developers collaboratively explore static-code analysis methods' results. The treemap visualization displays a software package, and each leaf node shows a class file. Angelini et al. [4] presented a visual analytics approach to assist users in exploring program execution, describing in a use case of the detection of single vulnerabilities. However, the system is mainly targeted to investigate symbolic execution engine data. Recently, Alperin et al. [3] presented a study for the interpretable visual assessment of vulnerabilities. In their study, the authors focus on local explanations for predictive vulnerability analysis.

In summary, the listed approaches focus on exploring network vulnerabilities and improving the code security of individual software packages, such as investigating potential false positives. In contrast, we propose an initial approach that provides an overview of entire software development organizations. Our design study focuses mainly on the visual analysis of OSS vulnerabilities by supporting auditing teams in assessing OSS dependencies through table-based views to evaluate vulnerabilities in large software organizations.

3 PROBLEM STATEMENT

The main goal of this work is to design visualizations to explore security risks in large software organizations. We gathered knowledge about the domain and user requirements in three interviews with two security analysts and a software developer from SAP. The interviews provided valuable insight into the daily workflows and challenges faced by security analysts regarding vulnerability assessment.

Application Background: The essential user task is to understand the overall risk exposure of large development organizations, e.g., commercial software vendors or open-source foundations, due to the consumption of open-source components in a considerable number of development projects or applications. During software development, projects are regularly scanned with code analysis tools. At SAP, the developers regularly utilize *Eclipse Steady*[3] [22], which supports static and dynamic analysis to detect and assess vulnerabilities. *Eclipse Steady* scans projects for Common Vulnerabilities and Exposures (CVE), which have a unique identifier in the National Vulnerability Database. *Eclipse Steady* displays the Common Vulnerability Scoring System (CVSS) score to indicate the severity of identified security vulnerabilities. However, the CVSS score only captures the vulnerability severity. Organizations require complementary information from other sources to evaluate the general software quality of the most-used libraries and determine whether they have sufficient quality. The identification of low-quality libraries is a prerequisite for follow-up decisions. However, the visual exploration of applications, consumed libraries, and related vulnerabilities on an organizational level are not supported by any of the tools available to date.

Requirements: From the interviews and further discussions with domain experts, we derived the following requirements for our tool aimed at the organization-wide analysis of software vulnerabilities.
(R1) Repositories - The tool should provide views to detect vulnerable repositories and projects to apply countermeasures, such as training weaker teams and reallocating resources. For this, repositories need to be represented in a comparable way to estimate relevance and understand how they compare against each other.
(R2) Libraries - Software projects potentially depend on vulnerable libraries, which have to be updated. Thus, the tool needs to convey the overall exposure and allow for the inspection of specific bugs.
(R3) Vulnerabilities - Vulnerabilities need to be explored to address specific exploited known vulnerabilities, e.g., OSS vulnerabilities prominently discussed in mainstream media, where organizations may be required or expected to make a statement whether and which of their applications are affected. Thus, the tool needs to enable users to find specific bugs with a high-security risk.
(R4) Vulnerability Severity - Vulnerabilities can have different effects depending on the severity and how many projects the origin is, and thus need to be prioritized accordingly. Therefore, the tool needs to show the impact of specific bugs on the organization's codebase.

4 TABLE-BASED VULNERABILITY EXPLORATION

In a two-year process, we applied the guidelines by Chen et al. [8] to perform our design study. Our tool covers the pipeline from scanning to repairing or mitigating a vulnerability. The overall workflow of VULNEX follows the KDD pipeline [12]. The workflow of VULNEX: (1) The security analyst starts a scan of the source code of all software projects. (2) He then selects a type of analysis target, i.e., repositories, libraries, or vulnerabilities, choosing between overviews. (3) Then, the analyst defines criteria he is interested in, i.e., the number or severity of a bug allowing for filtering. (4) The analyst observes the findings and determines their relevance by drilling down to the specific issue. (5) In case of a relevant finding, the analyst can start a repair or mitigation process; this is supported by the detailed report of *Eclipse Steady*. Thus, we follow Shneidermans' mantra: *Overview first, zoom and filter, then details-on-demand* [23]. In Fig. 1 we show the dependency tree (A), allowing users to explore the hierarchy of the software project and the vulnerability information (B) to give insight into the exposure to vulnerabilities.

4.1 Dependency Tree

The dependency tree representation in Fig. 1 (A) shows the relationship of all repositories, modules, libraries, and bugs. VULNEX is inspired by the tree+table approach by Nobre et al. [17, 18]. We choose this because tree structures are common and

[3] https://github.com/eclipse/steady

known by domain experts and allow us to leverage the hierarchy inherent in software projects while supplying additional information about vulnerabilities, keeping a high level of detail. The tree representation allows for the analysis of vulnerabilities in three ways.

Repository-centered: → → → →
This order of levels allows for a repository-focused analysis. Starting with a repository, then showing information about modules and sub-modules, enabling analysts to locate severe vulnerabilities. If a module uses a vulnerable library, this can be quickly detected. Finally, we show the vulnerabilities caused by a library, allowing for detailed analysis and estimation of the impact.

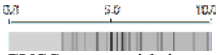
Library-centered: → → → →
Beginning with a library, displaying its vulnerabilities allows analysts to estimate the risk associated with a library. If a repository uses a vulnerable library, this repository is shown on the next level. Finally, we present the associated module or sub-module exposed to the CVE of that library, allowing for inspecting it in detail.

Bug-centered: → → → →
Starting with a CVE, then showing the affected library allows analysts to find specific bugs quickly. If a repository uses a vulnerable library, this repository is shown on the next level. Finally, we display the associated module or sub-module impacted by the CVE of that library, allowing for detailed analysis.

4.2 Vulnerability Information

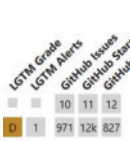
We provide additional information about the vulnerabilities of a repository, module, and libraries, shown in Fig. 1 (B), through which we support the detection and analysis of critical vulnerabilities, as well as the assessment of the quality of OSS dependencies, e.g., LGTM grade and score. The column shows the number of entities on the next level of the tree, indicating the number of related entities on the following hierarchy level. The column shows the number of vulnerabilities a repository, module, or library is exposed to. The absence of an element indicates that the information is not available or applicable to the entity of the row.

CVSS Score: The CVSS score column shows the number of CVEs with a given score. We use the common classification: Low (0.1 - 3.9), Medium (4.0 - 6.9), High (7.0 - 8.9), Critical (9.0 - 10.0), which is also mirrored in the coloring from the *National Security Database*[4]. The number in each square encodes the number of occurrences in the given range.
To inspect the distribution of CVSS scores in a more finely-grained way, we offer a representing of each CVE and its CVSS score with its precise value. It also indicates the range of CVSS scores.



CVE Matrix: The CVE matrix indicates the presence of a specific vulnerability. Each column shows the presence of a CVE with dark gray squares, while a light gray square indicates the absence of the CVE. We adopted this encoding from Nobre et al. [18]. Columns can be added and removed to highlight specific CVEs dependent on the user. The CVE matrix allows users to get an overview of the presence of specific vulnerabilities in repositories, modules, and libraries. It also enables the analysis of the co-occurrence of CVEs.

Meta-Information: We provide additional information from LGTM[5], a code analysis platform, and GitHub[6]. The columns describe the LGTM grade, LGTM score, GitHub issues, GitHub stars, GitHub watchers. The LGTM grade and score provide an additional measure for the quality of software artifacts. The number of GitHub issues provides an indicator for active development, while the GitHub stars and watchers provide an indicator for the popularity of a repository.



Dependency Graph: The user can view the structure of a software project by clicking . The repository is shown at the top, its modules and libraries in the middle, and the bugs at the bottom.

4.3 Filter and View Options

Based on expert feedback, we offer filter options to reduce the number of entries in the table and allow for a focused analysis. The user can search for a name of a given repository, library, or bug. We enable users to filter by the minimum and the maximum number of dependencies, vulnerabilities, and the CVSS score. Users can hide all repositories and modules that do not contain any vulnerability, as well as CVEs without a CVSS score.

5 Evaluation

5.1 Use case: Eclipse GitHub Repositories

We analyze all public GitHub repositories of the *Eclipse Foundation*[7] that are *Apache Maven*[8] projects in the *Java*[9] programming language. All projects are scanned using the *Eclipse Steady* tool. We scanned these repositories from January 21 to February 4, 2020. This yields a total of 295 projects that we analyze for common libraries and vulnerabilities. We replace the original repository and module names with pseudonyms not to blame the individual projects. At SAP, the analysis of an individual application follows a defined process, starting from the automated scanning with tools like *Eclipse Steady* to discover vulnerable open-source dependencies, the assessment of findings by a security expert, and finally, depending on the assessment result, the remediation of the vulnerability or the dismissal of the finding. However, open-source software analysis across multiple applications for an entire organization does not follow a defined process. To show the usefulness of VULNEX we analyze the gathered data to answer the following four questions. Questions (Q1-Q3) are examples for exploratory analysis, while (Q4) addresses a need when vulnerabilities in open-source components get a lot of public attention, even in mainstream media. In such cases, commercial vendors like SAP are expected to state to what extent and which of their applications are affected by a given vulnerability. Thus, we include (Q4) as a search task.

(Q1) Which repositories contain most severe vulnerabilities?
Security analysts utilize the *Repository Table* to analyze all repositories, depicted in Fig. 1. They find the "low-marmoset" repository, which has three critical bugs. We can see that all critical vulnerabilities are in the "satisfactory-haddock" module by expanding the entry. They inspect the module and see that the *tomcat-embed-core* library contains *CVE-2018-8014* and *activemq-all* contains *CVE-2018-1270* and *CVE-2018-1270*. They find that all three CVEs are critical, which should be addressed promptly.

(Q2) Which dependencies contain severe vulnerabilities and are often used across different applications?
The security analysts use the *Library Table* (see Fig. 2). They sort the table by the most severe vulnerability. The libraries *activemq-all*, *org.apache.lucene.queryparser*, *spring-data-commons*, *jgroups*, *groovy-all*, and *tomcat-embed-core* all contain critical bugs.

(Q3) Which severe vulnerabilities are present?
Using the *Bug Table*, the analysts find that eight critical bugs (see Fig. 3) are present, one in *activemq-all* affecting 20 repositories, one in *org.apache.lucene.queryparser* affecting 14 repositories, one

[4] https://nvd.nist.gov/
[5] https://lgtm.com/
[6] https://github.com/
[7] https://github.com/eclipse
[8] https://maven.apache.org/
[9] https://docs.oracle.com/en/java/index.html

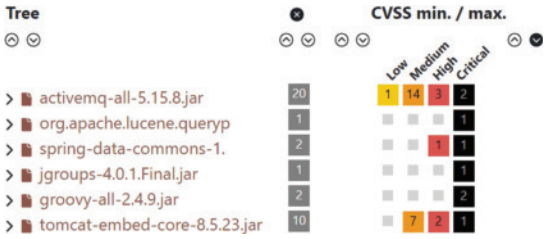


Figure 2: The analyst detects the most vulnerable libraries. *activemq-all* contains one low, 14 medium, three high, and two critical severity vulnerabilities, affecting 20 repositories.

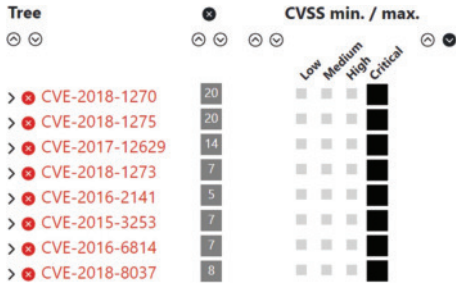


Figure 3: The analyst found eight critical vulnerabilities. *CVE-2018-1270* affects 20 repositories and has a critical severity.

in *spring-data-commons* affecting seven repositories, one in *jgroups* affecting five repositories, two in *groovy-all* affecting seven repositories, and one in *tomcat-embed-core* affecting eight repositories. They remark that these six libraries should be updated and fixed or replaced swiftly since they contain critical vulnerabilities.

(Q4) Are specific vulnerabilities present in any of the projects? Analysts searched for the oldest bug for the severities medium, high, and critical. For this task, they use the *Bug Table*. CVEs encode the years that they were detected. To find the oldest unfixed bug, they searched for the different years before 2019. They found *CVE-2009-2625*, a medium severity bug, present in *org.apache.xerces*, which affects 27 repositories. The oldest high severity bug they found was *CVE-2013-1768* in *openjpa-asm-shaded*, affecting three repositories and *CVE-2015-3253*, a critical bug, affecting seven repositories.

5.2 Preliminary User Feedback

We conducted an initial preliminary user feedback session with three software security analysts from SAP. All three participants (P1–P3) have five to ten years of experience in software security and work in dedicated security teams. Two participants support developers of mature applications regarding software security, including assessing the relevance and severity of vulnerabilities in open-source components. One participant acted as program manager for open-source security and DevSecOps, determining requirements, developing tools, and standardizing the secure consumption and publication of open-source components at SAP. We adopted the pair analytics guidelines of Kaastra and Fisher [15] to structure our interviews conceptually. During the one-hour interviews, we gathered regular user tasks, related employed visual interfaces, familiarity with information visualization for cyber-security, and afterward reviewed and compared in a live session their initial expectations to the proposed VULNEX tool.

All three participants approved the usefulness of VULNEX to visually explore the use of open-source libraries in large software organizations. P1 and P2 appreciated that the tool displays how often libraries and their potential vulnerabilities are used in the whole organization. P3 liked that the CVE matrix displays the top five bugs in the organization as it highlights the affected packages, including other prevalent vulnerabilities with their CVSS scores. Overall, all participants believed that VULNEX tool helps explore software organizations' vulnerabilities from different perspectives, such as in repository, library, and bug table views.

The participants expressed some concerns and outlined some shortcomings of our tool. P1 suggested adding additional information about the open-source libraries to the tool, such as short descriptions of the main functionality and purpose of the library. The participant argued that keeping track of each library's functionality without such additional information remains challenging due to the sheer number of 3rd party libraries. P2 emphasized that the current visual representations might not scale to large-scale organizations, e.g., organizations with more than 1000 repositories. P2 also proposed to enable the annotation of individual repositories, libraries, and bugs. Such annotations let analysts search for particular vulnerabilities and guide the auditing team to potential known solutions. P3 emphasized that his focus is heavily on vulnerabilities with critical CVSS scores above 9.0 that need to be resolved within several hours. Therefore, P3 suggested focusing on such vulnerabilities and recommending appropriate counter-measures. All participants suggested including potential solutions to resolve the vulnerabilities.

6 Discussion and Conclusion

We found that three security experts approved the usefulness of VULNEX. The experts found the different task-focused views useful. We learned that more detailed representations were less preferred. The domain experts had an easier time working with the categories low, medium, high and critical, rather than the precise values of the heatmap visualization. The CVE matrix gives a helpful overview of specific vulnerabilities. All vulnerability analysis tools at SAP focus on individual applications. Thus, we present VULNEX supporting organization- and enterprise-wide decision making. In terms of scalability, we performed our analyses on all public GitHub repositories of *Eclipse Foundation*. Therefore, we argue that VULNEX can be used for large software organizations since few organizations have more projects than the *Eclipse Foundation*.

We plan to address the feedback from the security experts by including a method to annotate repositories, modules, libraries, vulnerabilities and provide additional information for each item which could be taken from *libraries.io* or comparable online services. We also plan to include the temporal component, analyzing multiple "snapshots" to compare projects and understand how the organization's risk exposure develops over time. Another goal is to extend VULNEX for the assessment of libraries before choosing a specific one and provide a feedback loop to inform the open-source community and add the vulnerability information to the original repository. We also plan to evaluate VULNEX with experts external to the design process. Our approach is transferable to other organizations and open-source vulnerability analysis tools, but VULNEX is currently limited to the import and processing of scan results from *Eclipse Steady* allowing for the analysis of *Java* and *Python* code.

Determining the impact of vulnerabilities on software organizations is challenging due to the missing aggregation of software analysis results. As a solution, we propose the VULNEX (Vulnerability Explorer) tool, which we designed in a user-focused design process, which allows analysts to detect severe and relevant vulnerabilities and determine impacted libraries, modules, and repositories.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892.

REFERENCES

[1] State of the software supply chain (2020). Sonatype Inc., Technical Report, 2020. https://de.sonatype.com/resources/white-paper-state-of-the-software-supply-chain-2020, last accessed 2021-06-24.

[2] S. Akatsu, A. Masuda, T. Shida, and K. Tsuda. A study of quality prediction for large-scale open source software projects. *Journal of Artificial Intelligence Research*, 10(1):34–42, 2021. doi: 10.5430/air.v10n1p34

[3] K. B. Alperin, A. B. Wollaber, and S. R. Gomez. Improving interpretability for cyber vulnerability assessment using focus and context visualizations. In J. Kohlhammer, M. Angelini, C. Bryan, R. R. Gómez, and N. Prigent, eds., *17th IEEE Symposium on Visualization for Cyber Security*, pp. 30–39. IEEE, 2020. doi: 10.1109/VizSec51108.2020.00011

[4] M. Angelini, G. Blasilli, L. Borzacchiello, E. Coppa, D. C. D'Elia, C. Demetrescu, S. Lenti, S. Nicchi, and G. Santucci. SymNav: Visually assisting symbolic execution. In *16th IEEE Symposium on Visualization for Cyber Security*, pp. 1–11. IEEE, 2019. doi: 10.1109/VizSec48167.2019.9161524

[5] M. Angelini, G. Blasilli, T. Catarci, S. Lenti, and G. Santucci. Vulnus: Visual vulnerability analysis for network security. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):183–192, 2019. doi: 10.1109/TVCG.2018.2865028

[6] H. Assal, S. Chiasson, and R. Biddle. Cesar: Visual representation of source code vulnerabilities. In D. M. Best, D. Staheli, N. Prigent, S. Engle, and L. Harrison, eds., *13th IEEE Symposium on Visualization for Cyber Security*, pp. 1–8. IEEE Computer Society, 2016. doi: 10.1109/VIZSEC.2016.7739576

[7] B. L. Bullough, A. K. Yanchenko, C. L. Smith, and J. R. Zipkin. Predicting exploitation of disclosed software vulnerabilities using open-source data. In R. M. Verma and B. M. Thuraisingham, eds., *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, pp. 45–53. ACM, 2017. doi: 10.1145/3041008.3041009

[8] M. Chen and D. S. Ebert. An ontological framework for supporting the design and evaluation of visual analytics systems. *Computer Graphics Forum*, 38(3):131–144, 2019. doi: 10.1111/cgf.13677

[9] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, and W. Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, 2020. doi: 10.1007/s12650-020-00647-w

[10] S. Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007. doi: 10.1007/978-3-540-46505-8

[11] W. Fang, B. P. Miller, and J. A. Kupsch. Automated tracing and visualization of software security structure and properties. In D. Schweitzer and D. Quist, eds., *9th International Symposium on Visualization for Cyber Security*, pp. 9–16. ACM, 2012. doi: 10.1145/2379690.2379692

[12] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, 1996. doi: 10.1145/240455.240464

[13] J. R. Goodall, H. Radwan, and L. Halseth. Visual analysis of code security. In J. Gerth, ed., *7th International Symposium on Visualization for Cyber Security*, pp. 46–51. ACM, 2010. doi: 10.1145/1850795.1850800

[14] L. Harrison, R. Spahn, M. D. Iannacone, E. Downing, and J. R. Goodall. NV: Nessus vulnerability visualization for the web. In D. Schweitzer and D. Quist, eds., *9th International Symposium on Visualization for Cyber Security*, pp. 25–32. ACM, 2012. doi: 10.1145/2379690.2379694

[15] L. T. Kaastra and B. D. Fisher. Field experiment methodology for pair analytics. In *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization*, pp. 152–159. ACM, 2014. doi: 10.1145/2669557.2669572

[16] T. Moore. On the harms arising from the Equifax data breach of 2017. *International Journal of Critical Infrastructure Protection*, 19:47–48, 2017. doi: 10.1016/j.ijcip.2017.10.004

[17] C. Nobre, N. Gehlenborg, H. Coon, and A. Lex. Lineage: Visualizing multivariate clinical data in genealogy graphs. *IEEE Transactions on Visualization and Computer Graphics*, 25(3):1543–1558, 2019. doi: 10.1109/TVCG.2018.2811488

[18] C. Nobre, M. Streit, and A. Lex. Juniper: A tree+table approach to multivariate graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):544–554, 2019. doi: 10.1109/TVCG.2018.2865149

[19] V. Pham and T. Dang. CVExplorer: Multidimensional visualization for common vulnerabilities and exposures. In N. Abe, H. Liu, C. Pu, X. Hu, N. K. Ahmed, M. Qiao, Y. Song, D. Kossmann, B. Liu, K. Lee, J. Tang, J. He, and J. S. Saltz, eds., *IEEE International Conference on Big Data*, pp. 1296–1301. IEEE, 2018. doi: 10.1109/BigData.2018.8622092

[20] M. Pittenger. Open source security analysis - the state of open source security in commercial applications. *Black Duck Software, Technical Report*, 2016. https://www.vojtechruzicka.com/bf4dd32d5823c258c319cced38727dce/OSSAReport.pdf, last accessed 2021-06-21.

[21] H. Plate, S. E. Ponta, and A. Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In R. Koschke, J. Krinke, and M. P. Robillard, eds., *2015 IEEE International Conference on Software Maintenance and Evolution*, pp. 411–420. IEEE Computer Society, 2015. doi: 10.1109/ICSM.2015.7332492

[22] S. E. Ponta, H. Plate, and A. Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020. doi: 10.1007/s10664-020-09830-x

[23] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pp. 336–343. IEEE Computer Society, 1996. doi: 10.1109/VL.1996.545307

[24] A. S. Sohal, S. K. Gupta, and H. Singh. Trust in open source software development communities: A comprehensive analysis. *International Journal of Open Source Software and Processes*, 9(4):1–19, 2018. doi: 10.4018/IJOSSP.2018100101

[25] I. Sommerville. *Software Engineering*. it : Informatik. Addison-Wesley, Harlow, England, 9 ed., 2010.

[26] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, D. A. Keim, and W. Aigner. A survey of visualization systems for malware analysis. In R. Borgo, F. Ganovelli, and I. Viola, eds., *17th Eurographics Conference on Visualization*, pp. 105–125. Eurographics Association, 2015. doi: 10.2312/eurovisstar.20151114