Known Vulnerabilities of Open Source Projects: Where Are the Fixes?

Antonino Sabetta, Serena Elisa Ponta, Rocio Cabrera Lozoya, and Michele Bezzi | SAP Security Research
Tommaso Sacchetti | Eurecom Biot
Matteo Greco | Be-Innova Trento
Gergő Balogh, Péter Hegedűs, and Rudolf Ferenc | FrontEndART Ltd. and University of Szeged
Ranindya Paramitha | University of Trento
Ivan Pashchenko | TomTom
Aurora Papotti | Vrije Universiteit Amsterdam
Ákos Milánkovich | SEARCH-LAB
Fabio Massacci | University of Trento and Vrije Universiteit Amsterdam

Every day, developers have the daunting task of tracing vulnerabilities back in a morass of commits. In this article, we report the experience of the industrial open source tool, Prospector, to support developers in this task.

Detailed code-level vulnerability data are essential to fuel software composition analysis (SCA) tools that are used to detect known vulnerabilities in open source software (OSS) dependencies. However, such data are scarce; advisories rarely contain information about the code changes that fix the flaws they describe. Finding such code changes (for example, in source code repositories such as Git) manually is time consuming and error prone as it involves the analysis of multiple unstructured resources.

Introduction

To help security experts map vulnerability advisories onto the corresponding fix in the source code, we developed a tool called *Prospector* that employs a set of heuristics that mimics and automates the strategies inspired by those employed by human security experts. Given an advisory expressed in natural language, Prospector processes the commits found in the target source code repository, ranks them based on a set of predefined rules, and produces a report that the user can inspect to determine which commits constitute the fix. The tool, which is freely available under the Apache 2.0 license, has been tested on a dataset of 1,300+ vulnerabilities, and with minimal user input, it can successfully rank the candidate commits so that the actual fix is among the top 10 candidates in more than 90% of the cases.

OSS has undergone a remarkable evolution over the years, transforming from a spontaneous niche movement into a cornerstone of today's software industry and a critical component of modern technology stacks. The collaborative nature of open source development brings about undeniable benefits, such as considerable cost savings and faster innovation cycles achieved by integrating community-developed reusable building blocks.

In the last decade or so, however, the industry's initial enthusiasm for open source has given way to a sobering realization and a more realistic stance; incorporating open source components may speed up development

Digital Object Identifier 10.1109/MSEC.2023.3343836
Date of current version: 5 January 2024

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

(coding), but it comes with significant maintenance costs due to complex dependency structures that can be hard to reason about and to manage. A vulnerability in one component can have dramatic cascading effects. Developers and organizations must grapple with challenges such as version compatibility, conflicting dependencies, and the need to ensure that security updates are applied in a timely manner across all interconnected components.

A concrete effect of this realization is a growing awareness of the need for effective automated security analysis of open source components used within software projects. It is now imperative to understand and manage the complex web of dependencies that underlie modern software applications and that make up a large part of their attack surface. This realization led to the emergence of a thriving market of SCA tools. These tools, employing diverse techniques, become crucial as they offer a structured approach to deal with the complexity of open source dependencies.

The growing awareness of the security implications of open source has led to the development of a growing market for tools specifically designed to mitigate these challenges. One such category of tools is SCA tools. SCA tools serve as a crucial bridge between OSS and secure software development practices. They provide a systematic approach to identifying and managing OSS components within a software project, helping developers and organizations track their dependencies, vulnerabilities, and licensing requirements. Despite the technical differences in the methods they employ under the hood, their common foundation is constituted by some form of vulnerability databases. These databases act as vital repositories of knowledge, constantly updated with information on vulnerabilities discovered in open source components. Obviously, the analysis provided by SCA tools is only as robust as the quality and timeliness of the data they draw from these databases.

The National Vulnerability Database (NVD) stands as a de facto standard repository of security advisories, but it is not without its shortcomings; it is well documented that the NVD contains inconsistent or incomplete data related to vulnerability publication dates and to applications affected by the vulnerability, their severity scores, and their high-level type categorization.[1] Furthermore, NVD advisories that refer to OSS do not systematically link the code-level details about the vulnerability and its correction. This lack of code-level information has significant practical consequences. If the code changes that implement the fix to a given vulnerability are known, SCA can use this information to determine whether a particular artifact contains that code (and is therefore safe because it was fixed) or if it contains the code as it was *before* the fix (and hence, it is vulnerable). Without this information, the remaining alternative is to match artifact filenames (or maybe the contents of some metadata file included in the dependency package) against the contents of the advisory. For a discussion of the reasons why this method is less reliable and inadequate for industrial use, see the work of Ponta et al.[6,7]

The lack of comprehensive databases of code-level vulnerability fixes is also an obstacle to researchers who seek to study how vulnerabilities are introduced and corrected, and it hinders the development of more effective methods to detect, correct, and possibly prevent vulnerabilities. Finally, the availability of code patch data is essential for the development and training of machine learning (ML) applications that can improve the process of detecting and addressing vulnerabilities. It turns out that, in the process of building datasets that were instrumental to training ML models, researchers developed ad hoc tools that apply simple Common Vulnerabilities and Exposures (CVE)-to-commit mapping criteria to collect just enough fix commits to train an ML model.

Issues like inconsistencies and the absence of references to detailed code-level fixes in the NVD have spurred SCA vendors to build and maintain their own proprietary vulnerability databases. While these databases may enhance the accuracy of vulnerability information, they present a somewhat paradoxical situation; the data concerning vulnerabilities within OSS are not open themselves. By their proprietary nature, not only do these vulnerability databases hinder the further development of SCA tools (particularly of open source tools) that could push the state of the art in vulnerability detection and mitigation, but they also have the same scalability and coverage issues that we have experienced ourselves.

This paradox could be interpreted as a sign of a market that is still immature; therefore, vendors leverage their knowledge bases as differentiators rather than focusing on the unique analysis capabilities of their tools. Arguably, a community-driven approach would be more beneficial for the industry at large, including the vendors of SCA products themselves, who could concentrate their efforts on the differentiating capabilities of their tools rather than on the endless effort of building yet another vulnerability database (which is bound to be incomplete, no matter the amount of effort put into maintaining it).

How Do SCA Tools Work? The Case of Eclipse Steady and the Need for Accurate Code-Level Data About OSS Vulnerabilities

At an abstract level, the basic operation of an SCA tool consists of collecting the set of (third-party open source) dependencies that are imported by the application

under analysis and then querying a vulnerability database to obtain, for each such dependency, a list of vulnerabilities that affect it. The method to determine the set of dependencies is crucial. The open source tool OWASP Dependency Check relies on the *name and version* of the package as extracted from its metadata (that is, the name of the package archive, the contents of the metadata included in the archive, etc.), allowing it to readily query the NVD to obtain a list of relevant CVEs—an approach that has the advantage of being straightforward and lightweight but that comes with important shortcomings (see Ponta et al.[7] for more details). It bases its vulnerability-package mapping on metadata (package names and versions), but these metadata are known to be unreliable since they may be inconsistently reported in NVD records.[2] As a result, it can produce inaccurate results (both false positives and false negatives).

Furthermore, in several practical scenarios, developers may include "customized" versions of some dependencies (for example, through practices such as "repackaging" and "rebundling") that may alter filenames and metadata, making the task of correctly identifying the resulting packages nontrivial. To discern accurately whether such a dependency is affected by some vulnerability, one needs to consider the actual (byte-)code contained in the package.

For these reasons, accurate SCA relies on code-based analysis (as opposed to a purely metadata-based one), and this explains why code-level vulnerability data are so critical for SCA tools to work. But how are such code-level vulnerability data used in practice by SCA tools? Very little information is available on the internals of modern SCA tools, with a notable exception. Originally developed by SAP Security Research, Eclipse Steady (simply referred to as *Steady* in the following) was used at SAP between late 2016 and April 2021 to scan the open source dependencies of all SAP products developed in Java (later, some limited support for Python was added). At the peak of its utilization, the Steady instance deployed at SAP served thousands of distinct development teams, and by being integrated into the corporate continuous integration and continuous delivery pipelines, it performed more than 250,000 scans a month.

In a nutshell, Steady is used to 1) detect whether a given application depends on open source components that are affected by known vulnerabilities; 2) collect evidence regarding the execution of vulnerable code in the context of the application at hand, through a combination of static and dynamic analysis techniques; and 3) support developers in replacing the vulnerable dependency artifacts with alternative versions that are not vulnerable. For all these three functions, Steady relies on code-level knowledge of the vulnerability and of its correction. To detect if a dependency is affected by a particular vulnerability, the package for that vulnerability is searched to find the functions that are touched by the fix; if they are found, Steady checks whether the code of those functions in the dependency artifact at hand corresponds to the version before the fix (vulnerable) or after the fix (safe) found in the source code repository of that dependency. Analogously, knowing the functions that are related to a given vulnerability, Steady enables an impact assessment based on the reachability of those functions from the code of the application under analysis.

Vulnerability Datasets and Mining Tools

Given the code-based approach on which Steady relies, having a rich knowledge base of code-level vulnerability data to rely upon is a priority when operating the tool productively in an enterprise context. Collecting and curating vulnerability information, however, is difficult and time consuming—as the team that operated Steady at SAP learned the hard way by committing considerable effort to the mining of source code repositories and the curation of a knowledge base of fix commits. Given the increasing size of open source ecosystems and the pace at which new vulnerabilities affecting OSS are discovered at a continuous pace, a manual approach to creating and maintaining such a knowledge base cannot scale.

The problem of mining source code repositories to build datasets of vulnerability data received attention in the scientific literature, too, resulting in several useful datasets and mining techniques (see Table 1). Tool availability, however, is severely lacking. As shown in Table 1, scientific articles that present repository mining approaches to find vulnerability-fixing commits are rarely (virtually never) accompanied by a publicly available tool that implements the approach that the article describes. This makes it hard to reconstruct and extend those datasets.

Building a Vulnerability Database for Eclipse Steady: Project KB and Prospector

To fill this gap, SAP published Project KB, an initiative to support the creation and maintenance of a collaborative knowledge base of vulnerability data about open source projects. Project KB consists of a vulnerability dataset (a collection of plain-text files, each containing information about a vulnerability, including the references to the fixing commits) as well as a tool, named *Prospector*, whose goal is to assist security experts in searching open source code repositories to locate the commits that address a given known vulnerability. Prospector aims to significantly reduce the effort that is

necessary to search open source code repositories for commits that correct a given vulnerability and thus contribute to the construction and maintenance of comprehensive datasets of code-level vulnerability fixes.

To do so, the tool employs a set of heuristics to search for commits that match predefined criteria that indicate the presence of vulnerability fixes. By analyzing the advisory given as input and by scanning commit messages, code changes, and related metadata, the tool aims to rank and pinpoint the commits that address the vulnerability described in the advisory while providing a plain English explanation of the reasons why the selected commits are considered a vulnerability fixing commit.

Prospector is released under the Apache 2.0 license terms as part of SAP's Project KB (https://github.com/sap/project-kb). To the best of our knowledge, it is the only tool of its kind that can provide a clear explanation for its outputs, is open source, and has been validated on a large scale in an industrial context.

Figure 1 illustrates the workflow of Prospector. The tool takes as input a vulnerability identifier, a repository URL, and (optionally) a version interval and produces a report that lists commits found in the repository likely to fix the vulnerability. The report presents the commits ranked by relevance, which is computed by applying a set of rules to each candidate. The steps of the workflow in Figure 1 are as follows.

Table 1. Other works on mapping advisories to vulnerability fixing commits in source code repositories.

Authors	Method summary	Tool availability
Perl et al.[5]	Mapped CVE to Git commits using direct references in both directions	No publicly available implementation
Xu et al.[11]	*Tracer*: finds patches from multiple knowledge sources	Tool not available from the URL indicated in the article*
Hong et al.[4]	*xVDB*: finds patches by analyzing code repositories, bug trackers, and Q&A sites; by doing so, it discovers more patches than those	Not available
Tan et al.[10]	*PatchScout*: a ranking-based approach that computes a correlation score between the commits in a repository and a given vulnerability; the approach is demonstrated on a set of five popular open source projects	Not available
Dunlap et al.[3]	*Vfcfinder*: a recently published tool with limited documentation; uses CodeBERT and similarity measures to match commits and advisories	Open source tool available on GitHub
Wang et al.[12]	*PatchDB*: collects patches obtained from NVD references as well as other sources on the web; focuses on C/C++ only	Not available

**https://patch-tracer.github.io* (23 May 2023).

Advisory Retrieval and Processing

Given the vulnerability identifier in input, Prospector retrieves the advisory from the NVD and processes it to extract information for the later steps. (Note that sources other than the NVD could also be used.) In particular, it retrieves the advisory description, time stamps, and references. From the description, Prospector extracts keywords characterizing the vulnerability by using natural language processing techniques, the presence of security-relevant keywords by matching a static list, and mentioned file/classes/method names by using pattern matching. For relevant references, for example, GitHub pages and bug-tracking tickets, Prospector employs parsing techniques to extract hyperlinks therein present. The identified keywords, files, classes, methods, and hyperlinks are later used during the rule application step and are key for the ranking of the candidate commits.

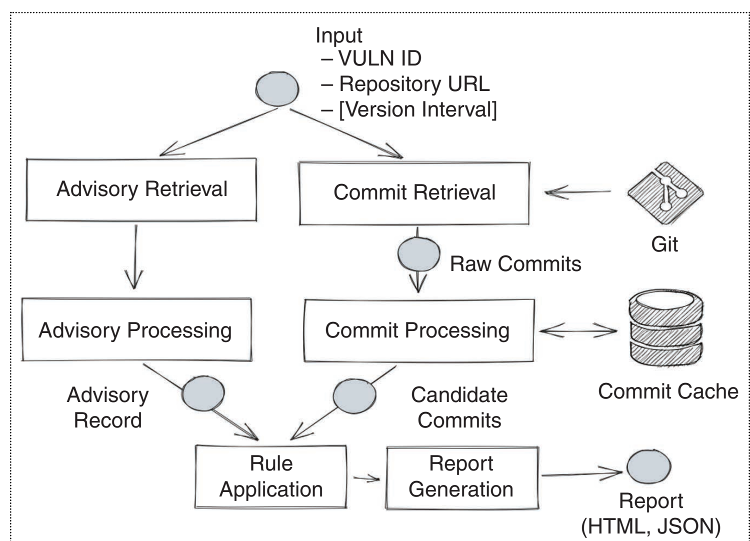

Figure 1. A high-level overview of Prospector's workflow. JSON: JavaScript Object Notation; VULN: vulnerability identifier.

Commit Retrieval and Processing

Besides building and analyzing the advisory record, Prospector retrieves a set of candidate commits from the source code repository. The commit retrieval step relies on Git to streamline the cloning of repositories and enable the efficient selection of commit subsets. If a

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

version interval is provided in input, Prospector uses it to retrieve commits whose time stamp falls in between the release dates; otherwise, commits in the time window of 60 days before and after the advisory reservation date are retrieved.

Once the candidate retrieval process is completed, each commit undergoes further processing to extract additional information to be used in the rule application step (unless already available in the commit cache). This includes metrics such as the number of hunks, changed files, code changes, and references to bug-tracking identifiers and GitHub issues in the commit message. Processed commits are stored in the cache.

Rule Application

Given a processed advisory and a candidate commit, Prospector applies a set of rules to establish whether there is a match, that is, whether the input satisfies the rule definition. The rules currently implemented in Prospector are summarized in Table 2. For example, rule `CVE_ID_IN_MESSAGE` reads the attribute of the advisory record that contains the *vulnerability identifier* and checks whether that identifier appears in the log message of the candidate commit at hand. As another example, the rule `XREF_BUG` is satisfied if the candidate commit in the input contains in its commit message a bug-tracking ticket identifier extracted during the processing of the advisory (that is, mentioned in the advisory itself or in any of the referenced pages).

When a match is determined, the candidate commit is annotated with the rule identifier and a plain English explanation of the match. This explanation and the annotations are used when producing the final report that the user will inspect.

Ranking and Report Generation

After the rule application step, the annotations associated with each candidate commit are leveraged to rank the commits. Rules are associated with a relevance weight (weight *W* in Table 2) that is assigned following

Table 2. A summary of the rules.

Rule identifier	Weight	Rule description
VULN_ID_IN_MESSAGE	64	The commit message mentions the vulnerability identifier.
COMMIT_IN_REFERENCE	64	The commit is mentioned directly by the advisory or in any of the pages referenced by the advisory.
XREF_BUG	32	The commit message contains the identifier of a bug-tracking ticket that is mentioned in the advisory or in any of the pages referenced by the advisory.
XREF_GH	32	The commit message contains the identifier of a GitHub issue that is mentioned in the advisory or in any of the pages referenced by the advisory.
VULN_ID_IN_LINKED_ISSUE	32	The commit message mentions a GitHub issue or bug-tracking ticket containing the vulnerability ID.
CHANGES_RELEVANT_FILES	8	The commit modifies files mentioned in the advisory description.
CHANGES_RELEVANT_CODE	8	The commit modifies code containing a filename or method mentioned in the advisory description.
RELEVANT_WORDS_IN_MESSAGE	8	The commit message contains a filename or method mentioned in the advisory description.
ADV_KEYWORDS_IN_FILES	4	The commit modifies files whose names contain keywords extracted from the advisory description.
ADV_KEYWORDS_IN_MSG	4	The commit message contains one or more keywords extracted from the advisory description.
SEC_KEYWORDS_IN_MESSAGE	4	The commit message contains one or more predefined security-related keywords.
SEC_KEYWORDS_IN_LINKED_GH	4	The commit points to a GitHub issue containing one or more predefined security-related keywords.
SEC_KEYWORDS_IN_LINKED_BUG	4	The commit points to a bug-tracking ticket containing one or more predefined security-related keywords.
GITHUB_ISSUE_IN_MESSAGE	2	The commit message mentions a GitHub issue.
BUG_IN_MESSAGE	2	The commit message mentions a bug-tracking ticket.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

a logarithmic scale, where a higher value implies a higher confidence in identifying the commit as the vulnerability patch. This logarithmic scale enables differentiating between strong (that is, $W \geq 32$) and weak rules, ensuring that the application of multiple weak rules cannot outweigh a single strong rule during the ranking process. The current values are based on expert knowledge and need to be validated empirically in future work. To rank each candidate, a relevance score is calculated by summing the weights of all the matched rules. The results are saved in a report that displays the list of candidate commits in descending order of relevance. By presenting the commits in this manner, the report effectively highlights at the top the commits that have the highest likelihood of fixing the given vulnerability.

An Illustrative Example

To illustrate how Prospector works, but also to explain the kinds of tasks that a human expert would have to perform to find the fix commits corresponding to a given advisory, we examine a sample vulnerability. `CVE-2020-1925` (https://nvd.nist.gov/vuln/detail/CVE-2020-1925) refers to a server-side request forgery vulnerability in Apache Olingo (https://olingo.apache.org). The vulnerability description (from the NVD) indicates that an attacker could exploit a bug in the implementation of class `AsyncRequestWrapperImpl` in versions 4.0.0 to 4.7.0.

Manual Search

To find the fix commit *manually*, one could rely on the list of references provided by the NVD advisory. For `CVE-2020-1925`, one of the references points to a message in the mailing list archive of the project (https://lists.apache.org/thread/6jfkh27wgdyq1w8ghq2qy3ggk27p74mx). The e-mail message does not point to the patch itself; however, it mentions a bug-tracking ticket where the issue is discussed (https://issues.apache.org/jira/browse/OLINGO-1416), and that, in turn, points to a GitHub pull request (PR) (https://github.com/apache/olingo-odata4/pull/63) used to merge the patch into *Apache Olingo*. Following this chain of links, it is possible to identify the commit containing the patch of the vulnerability as the commit merging PR #63 into the main branch. However, this approach is time consuming, tedious, and error prone.

Alternatively, one could use the first fixed release as the starting point for searching to determine which commits fall in the interval between the last vulnerable release (4.7.0) and the first fixed (4.7.1 in this case). Each of these commits should be examined to identify which ones best match the vulnerability description. For `CVE-2020-1925`, it is quite easy to identify that the merge commit of PR #63 is the (only) fix commit; all other commits either update project dependencies or address clearly unrelated issues (for example, OData-related improvement). `CVE-2020-1925` is a rather straightforward case; still, performing the manual tasks outlined previously requires browsing and analyzing four different webpages (from the NVD advisory to the fix commit) and five commits, which also involves inspecting the files and classes changed by each of them. Note that for `CVE-2020-1925`, the NVD and OSV do not provide any fix commit, and Snyk [https://security.snyk.io/vuln/SNYK-JAVA-ORGAPACHEOLINGO-541532 (23 May 2023)] points only to the GitHub PR page but not to individual commits.

Tool-Assisted Search

To find the fixing commit, Prospector must be given at least two input parameters: the vulnerability identifier (`CVE-2020-1925`) and the URL of the corresponding source repository (https://github.com/apache/olingo-odata4). During the *advisory retrieval and processing* step, the NVD application programming interface is queried to retrieve the advisory. The analysis of the vulnerability description results in the identification of keywords, for example, `header`, `client`, and the relevant file, that is, `AsyncRequestWrapperImpl`. Then, from the NVD advisory page, Prospector analyzes the content of any relevant reference, that is, the Apache mailing-list message (https://lists.apache.org/thread/6jfkh27wgdyq1w8ghq2qy3ggk27p74mx) in this case. Next, it parses the content of this message and extracts any valid URLs. Notably, during this process, the tool successfully identifies a hyperlink leading to the bug-tracking ticket labeled `OLINGO-1416`.

In the *commit retrieval and processing* step, Prospector retrieves 35 candidate commits, that is, those committed fewer than 60 days before or after the time stamp available in the advisory. Each one of these commits undergoes further processing to extract additional information, such as the list of changed files and bug-tracking identifiers mentioned in the commit messages.

In the *rule application* step, Prospector applies the rules of Table 2 to each of the 35 candidate commits. Commits that do not modify any source code file are filtered out. Then, Prospector generates a report showing, for each commit, the commit message, the relevance score, the commit ID, and the rules that matched (including an explanation, in natural language, of the match).

Figure 2 shows the highest-ranked commit (`9f9aebd`) from the report for `CVE-2020-1925`. This commit has a relevance of 58 resulting from the sum of the weights of the rules that matched. The identifiers of these rules are in blue in Figure 2. The most important are as follows:

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

- *XREF_BUG:* The commit message and an advisory reference both mention the bug-tracking ticket with the identifier "OLINGO-1416."
- *CHANGES_RELEVANT_CODE:* The commit modifies code containing the class mentioned in the advisory description (*AsyncRequestWrapperImpl*).
- *ADV_KEYWORDS_IN_MSG:* The commit message contains the keyword header, present in the advisory description.

Rule XREF_BUG mimics a human expert manually navigating and scrolling through four different webpages. Moreover, rules CHANGES_RELEVANT_CODE, CHANGES_RELEVANT_FILES, and ADV_KEYWORDS_IN_FILES highlight the commits that touch files/classes/methods whose names are likely to be related to the advisory. A manual search by a human would require browsing through and analyzing one-by-one the commits preceding the release of the fixed library; even using some ad hoc scripts for this task is cumbersome, time consuming, and error prone.

The report offers a consolidated view of the information that users would need to look up manually across several resources. The straightforward ranking system of Prospector gives higher visibility to commits that match rules that, based on our experience, provide a stronger signal (for example, XREF_BUG).

Evaluation

To evaluate if Prospector could provide useful support to users and reduce the manual effort required to find fix commits, we tested it on the Tracer "depth dataset" by Xu et al.[11] This dataset includes 1,319 vulnerabilities affecting 723 open source projects developed in seven programming languages.

For the sake of conciseness, we provide only a brief overview of the evaluation here; a more comprehensive description will be the topic of a future article. After a preliminary manual review of the Tracer dataset (which resulted in several corrections and additions of missing fixes), we executed Prospector in three distinct configurations: 1) by supplying no version interval (designated by *NV*); 2) by supplying an automatically extracted version interval obtained by applying basic pattern matching to the text of the advisory (designated by *AEV*); and 3) by supplying manually specified version intervals (designated by *MEV*).

When no version interval is provided or when the tool fails to match the versions provided as input to tags found in the source code repository, Prospector is instructed to fall back to a fixed time interval of 120 days centered around the advisory reservation date (that is, the candidates are selected starting from 60 days before the advisory reservation date, until 60 days after the same date). We set a 30-min execution time limit for each vulnerability, and we aborted executions that would retrieve more than 2,000 candidate commits.

Fixing Commit Ranked Among the Top 10 in the Report

When a known fixing commit is found among the top-10 candidates, we distinguish the following cases.

- *High confidence:* The commit is ranked first with at least a strong rule matching.
- *Medium confidence:* The commit is ranked first with only weak rules matching.
- *Low confidence:* The commit is ranked among the first 10 (but not in the first position).

Prospector Report

Copy to clipboard | Download as YAML statement

[OLINGO-1416] Better header processing

Relevance: 58 | Tag: 4.7.1-RC01

XREF_BUG | CHANGES_RELEVANT_CODE | CHANGES_RELEVANT_FILES | ADV_KEYWORDS_IN_MSG | ADV_KEYWORDS_IN_FILES | BUG_IN_MESSAGE

9f9aebde557b791f275d6156d8bec12ac334425d

Open commit

Matched rules

- The commit and the advisory (including referenced pages) mention the same bug tracking ticket: OLINGO-1416
- The commit modifies code containing relevant filename or methods: AsyncRequestWrapperImpl
- The commit changes some relevant files: lib/client-core/src/main/java/org/apache/olingo/client/core/communication/request/AsyncRequestWrapperImpl.java
- The commit message and the advisory description contain the following keywords: header
- An advisory keyword is contained in the changed files: client, asyncrequestwrapperimpl, request
- The commit message references some bug tracking ticket: OLINGO-1416

Figure 2. A Prospector report.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 3. A summary of the experiment results.

Result	Manually supplied versions (MSVs)	%	Automatically extracted versions (AEVs)	%	No versions provided (NV)	%
High confidence	984	74.83%	976	74.56%	928	71.61%
Medium confidence	109	8.29%	97	7.41%	50	3.86%
Low confidence	98	7.45%	90	6.88%	45	3.47%
Low relevance (rank > 10)	39	2.97%	35	2.67%	27	2.08%
Not found	73	5.55%	97	7.41%	232	17.9%
False positives	12	0.91%	14	1.07%	14	1.08%
Total	**1,315**		**1,309**		**1,296**	

Fixing Commit Does Not Appear Among the Top 10 in the Report

When a known fixing commit is not found among the top-10 candidates, we distinguish the following cases.

- *Found with low relevance (rank > 10):* The commit was found by Prospector, but its position in the ranking is not among the first 10
- *Not found:* The commit does not appear in the report at all
- *False positive:* A commit matches a strong rule but is not an actual fixing commit.

In the Tracer dataset, some vulnerabilities had more than one fixing commit (which could be completely different commits or replicas of the same commit in multiple parallel branches). We considered a fixing commit for a vulnerability as *found* if Prospector could identify at least one of the commits listed in the Tracer dataset.

The results of the executions in the three conditions are reported in Table 3. Some executions were aborted when no versions were supplied (since, in this case, Prospector is more likely to consider an excessive number of candidates, therefore hitting the 2,000 commit threshold). Also, executions were aborted when the automatically extracted versions were incorrect and could not be matched with tags in the Git repository. These aborted executions are the reason why the number of vulnerabilities successfully analyzed in each condition differs.

Table 4 shows that when the tool is provided with accurate version intervals (that is, when such intervals are provided manually), the actual fix commits are ranked among the top 10 candidates in 90.57% of the cases. (The percentage is 74.83% if one considers only high-confidence findings, for which one or more "strong" rules match.) When extracting version intervals automatically from the advisory text, we observe a success ratio of 88.85% (74.56% when considering only the high-confidence matches). Finally, when version intervals are not specified at all, the success rate drops to 78.94% (71.61% when considering only the high-confidence matches).

The execution mode we are more interested in is AEV because it constitutes the base accuracy for fully automated runs of Prospector. The results we observed are close to the manually supplied versions (MSVs) case, which is quite encouraging. In the cases where the references do not contain a link to a commit, the success ratio drops to 72.44%.

The false positives are the cases in which a strong rule matches incorrectly and the report does not contain any of the fixing commits. These cases represent only 0.91% of the total and were mostly due to wrong references to fix commits being present in the advisory itself or to corner cases where the advisory contained a link to bug-tracking tickets that were related to the actual ticket with which the vulnerability was discussed and corrected. (This kind of indirection could be handled in future versions of the tool but is not correctly dealt with at present.)

The evaluation we conducted is encouraging as it shows that with relatively simple human-crafted rules, it is possible to considerably simplify the task

Table 4. The summary of results.

Result	MSV	AEV	NV
Fix commit **found** in the top 10	1,191 (90.57%)	1,163 (88.85%)	1,023 (78.94%)
Fix commit **not found** in the top 10	124 (9.43%)	146 (11.15%)	273 (21.06%)

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

of security experts. With minimal human intervention, the tool can be made to produce reports that a human expert can examine and understand, thanks to the plain English explanations that are generated for each rule match.

After our experience with mining source code repositories to find vulnerability-fixing commits (with and without Prospector), we are left with a fundamental question: "Why does it have to be so difficult?" To answer this question, one needs to consider several factors combined. The practices adopted across the open source community vary considerably. Even in the scope of a single project, the vulnerability management processes are often executed in an inconsistent manner. More broadly, the community still hasn't reached a broadly shared consensus as to how to identify, maintain, and share the link between a vulnerability description, the content of bug-tracking systems, and ultimately, the correction in the source code. Some project maintainers seem to be worried about the risk of giving attackers an advantage by explicitly annotating security fixes in their source code repositories.

In many projects, however, vulnerability fixing commits are occasionally annotated, but not always, and not consistently, giving the impression that the link between a security issue (for example, as described in a bug-tracking ticket) and its fix in the source code is just not deemed worth preserving by many project maintainers and contributors. More generally, it seems that the importance of commit quality is often disregarded, which, among other undesirable effects, results in security fixes being mixed with other kinds of changes.

What to expect from the future? Vulnerability management processes seem to be improving, and we hope that with them, developer awareness will increase as well, leading maintainers to pay more attention to the importance of sharing detailed fix information with their security advisories. This would make the task of Prospector much easier; eventually, it could even make a tool like Prospector completely unnecessary, which is precisely our wish. In the meantime, the problem of efficiently and effectively mining repositories for accurate code-level vulnerability data will remain open, and tool support will remain crucial. The introduction of rules that use large language models is the obvious next step in the evolution of the tool, and it is part of our plans. Also, further evaluation of the effectiveness of the tool when used by real users is ongoing and will be the subject of another article.

Prospector is released under the Apache 2.0 license terms as part of SAP's project KB, and it can be downloaded from the Project KB repository at https://github.com/sap/project-kb. Two datasets that were obtained using Prospector are publicly available as open data.[8,9] ■

Acknowledgment

This work was partially supported by EU-funded projects Sec4AI4Sec (Grant 101120393) and AssureMoss (Grant 952647) and NWO-funded project Theseus (Grant NWA.121518006). Antonino Sabetta would like to thank Henrik Plate, Bonaventura Coppola, Daan Hommersom, Damian A. Tamburri, and Dario Di Nucci for insightful discussions.

References

1. A. Anwar, A. Abusnaina, S.Chen, F. Li, and D. Mohaisen, "Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses," in *Proc. 51st Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw., Supplemental Volume (DSN-S)*, 2021, pp. 1–2, doi: 10.1109/DSN-S52858.2021.00011.
2. Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *Proc. 28th USENIX Conf. Secur. Symp. (SEC)*, Berkeley, CA, USA: USENIX Association, 2019, pp. 869–885, doi: 10.5555/3361338.3361399.
3. T. Dunlap et al., "VFCFinder: Seamlessly pairing security advisories and patches," 2023, *arXiv:2311.01532*.
4. H. Hong, S. Woo, E. Choi, J. Choi and H. Lee, "xVDB: A high-coverage approach for constructing a vulnerability database," *IEEE Access*, vol. 10, pp. 85,050–85,063, Aug. 2022, doi: 10.1109/ACCESS.2022.3197786.
5. H. Perl et al., "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: Association for Computing Machinery, 2015, pp. 426–437, doi: 10.1145/2810103.2813604.
6. S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution (ICSME)*, 2018, pp. 449–460, doi: 10.1109/ICSME.2018.00054.
7. S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Softw. Eng.*, vol. 25, pp. 3175–3215 Jun. 2020, doi: 10.1007/s10664-020-09830-x.
8. A. Sabetta, S. E. Ponta, M. Greco, T. Sacchetti, and M. Bezzi, 2023, "AssureMOSS Vulnerability Statements Dataset (Tracer) (1.0) [Data set]," Zenodo, doi: 10.5281/zenodo.8173990.
9. A. Sabetta, S. E. Ponta, M. Greco, and T. Sacchetti, 2023, "AssureMOSS Vulnerability Statements Dataset (Steady) (1.0) [Data set]," Zenodo, doi: 10.5281/zenodo.8163119.
10. X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the security patches for disclosed OSS vulnerabilities with vulnerability-commit correlation ranking," In *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 3282–3299, doi: 10.1145/3460120.3484593.

11. C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, "Tracking patches for open source software vulnerabilities," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 860–871, doi: 10.1145/3540250.3549125.
12. X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, "PatchDB: A large-scale security patch dataset," in *Proc. 51st Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Taipei, Taiwan, 2021, pp. 149–160, doi: 10.1109/DSN48987.2021.00030.

Antonino Sabetta is a principal research scientist at SAP Security Research, 06254 Mougins, France. His research interests include software security, open source software, and applications of machine learning to software development and analysis. Sabetta received a Ph.D. from Università degli Studi di Roma "Tor Vergata." Contact him at antonino.sabetta@sap.com.

Serena Elisa Ponta is a principal research scientist at SAP Security Research, 06254 Mougins, France. Her research interests include open source software (OSS) security, the security of OSS supply chains, the analysis and management of known vulnerabilities, and the detection of malicious code in OSS libraries. Ponta received a Ph.D. in mathematical engineering and simulation from the University of Genova. Contact her at serena.ponta@sap.com.

Rocio Cabrera Lozoya is a senior data scientist with SAP Security Research, 06254 Mougins, France. Her research interests include the use of machine learning algorithms for security applications. Lozoya received a Ph.D. in 2015 on modeling and classification of cardiac electrophysiology signals from her work done at the Asclepios Research team in INRIA Sophia Antipolis. Contact her at rocio.cabrera.lozoya@sap.com.

Michele Bezzi is the research manager at SAP Security Research, 06254, Mougins, France. His research interests include software security and artificial intelligence. Bezzi received a Ph.D. in physics from the University of Bologna. He coordinated the EU project ASSERT4SOA on security certification. Contact him at michele.bezzi@sap.com.

Tommaso Sacchetti is currently a Ph.D. candidate in the S3 research group at Eurecom, 06410 Biot, France, working on Bluetooth and Internet of Things security. Sacchetti received a master's degree at the University of Trento. Contact him at tommaso.sacchetti@eurecom.fr.

Matteo Greco is a network and security system engineer at Be-Innova, 38123 Trento, Italy. Within the Security Operation Center, he is responsible for analyzing and managing security incidents. Additionally, he actively conducts vulnerability assessments and penetration testing. Greco received a master's degree in cybersecurity at the University of Trento. Contact him at matteo.greco@be-innova.eu.

Gergő Balogh is a researcher and software developer at the Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary. His research interests include human-centric aspects of software engineering, focusing on methodologies from psychology and social sciences to enhance empirical studies in development environments, algorithm usability, and developer productivity. Contact him at geryxyz@inf.u-szeged.hu.

Péter Hegedűs currently works both as an assistant professor at the Department of Software Engineering, University of Szeged and as a researcher at FrontEndART Ltd., 6720 Szeged, Hungary. His research interests include software maintainability models, deep learning applications, source code analysis, and vulnerability detection and prediction. Hegedűs received a Ph.D. in computer science from the University of Szeged in 2015. He was a program committee member in the CSMR, MSR, QUATIC, ESEM, and SCAM conferences and received various awards and scholarships during his career, like the prestigious Bolyai János research scholarship. Contact him at peter.hegedus@frontendart.com.

Rudolf Ferenc is currently an associate professor, acting as the head of the Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary. He leads the Static Code Analysis Group, which develops tools for analyzing the source code of various languages. His research interests include static code analysis, metrics, quality assurance, design pattern and antipattern mining, and bug detection. Ferenc received a Ph.D. in computer science from the University of Szeged in 2005 and received a Habilitation degree in 2015. Contact him at rudolf.ferenc@frontendart.com.

Ranindya (Nanin) Paramitha is a Ph.D. student at the University of Trento, 38123 Trento, Italy. Her research interests include software security, focusing on empirical analysis of secure software ecosystems, mining software repositories, and how developers can apply security. Paramitha received a master's degree in informatics with distinction from Institut Teknologi

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Bandung, Bandung, Indonesia. She participated in the H2020 Project AssureMOSS and is involved in the HE Project Sec4AI4Sec. She is on the program committee of ICSE SVM'23. She is a Graduate Student Member of IEEE. Contact her at ranindya.paramitha@unitn.it.

Ivan Pashchenko is an expert security engineer at TomTom, 1011 AC Amsterdam, The Netherlands. His research interests include software security, open source software security, and machine learning for security. Pashchenko received a Ph.D. from the University of Trento. In 2017, he was awarded a Second Place Silver Medal at the Association for Computing Machinery/Microsoft Student Research competition in the graduate category. He was the UniTrento main contact in the Continuous Analysis and Correction of Secure Code work package for the Horizon 2020 Assurance and Certification in Secure Multi-Party Open Software and Services project. Contact him at ivan.pashchenko@tomtom.com.

Aurora Papotti is a Ph.D. student at Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands. Her research interests include the human usability of advanced software security technology from artificial intelligence to static analysis. Papotti received a double master's degree in computer science from the University of Trento, Italy, and the University of Turku. She participated in the NWO project Theseus and HEWSTI and is involved in the HE Project Sec4AI-4Sec. Contact her at a.papotti@vu.nl.

Ákos Milánkovich is currently a security researcher at SEARCH-LAB, Budapest, H-2370 Dabas, Hungary, leading EU-funded research and penetration testing activities. His research interests include wireless sensor networks, agricultural monitoring, ultrawideband localization, and security. Contact him at akos.milankovich@search-lab.hu.

Fabio Massacci is a professor at the University of Trento, 38123 Trento, Italy, and Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands. His research interests include empirical methods for the cybersecurity of sociotechnical systems. Massacci received a Ph.D. in computing from the Sapienza University of Rome. He participates in the Cyber Security for Europe pilot and the NWO Theseus project and leads the Horizon 2020 AssureMOSS project and the Horizon Europe Sec4AI4Sec. For his work on security and trust in sociotechnical systems, he received the Ten Year Most Influential Paper Award at the 2015 IEEE International Requirements Engineering Conference. He is a Member of IEEE. Contact him at fabio.massacci@ieee.org.