Common Practices in Fixing Open Source Related Vulnerabilities

Lero Technical Report No. 2023_TR_01_Patching_OSS_Vulnerabilities

Gul Aftab Ahmed[*], Muslim Chochlov[†], David Gregg[‡],
Jim Buckley[§] and James Vincent Patten[¶]
Email: [*]ahmedga@tcd.ie, [†]muslim.chochlov@ul.ie, [‡]david.gregg@tcd.ie,
[§]jim.buckley@ul.ie, [¶]james.patten@lero.ie

14 January 2023

Table of Contents

I. Introduction

A Software vulnerability is a weakness in a software system that can be exploited by an attacker to gain unauthorized access, execute arbitrary code, or cause a denial of service. Software vulnerabilities can be found in all types of software, including operating systems, applications, and firmware. Information about these vulnerabilities is collected and disseminated via various vendor neutral or vendor specific databases. These databases are useful as they provide a central location where information about vulnerabilities can be collected, organized, and disseminated. They also provide a way for software vendors to track and respond to vulnerabilities in their products, and for users to be notified of vulnerabilities that may affect them.

There are two popular databases which are de facto-authoritative resources to discover or report vulnerabilities; The first is the CVE List [1] which was launched by MITRE as a community effort in 1999. The second is the U.S. National Vulnerability Database (NVD) [2] which was launched by the National Institute of Standards and Technology (NIST) in 2005. Both databases compliment each other. The Common Vulnerabilities and Exposures (CVE) list provides a comprehensive dictionary of vulnerabilities that contains a standard identifier number, a brief description, and references to related vulnerability reports and advisories. All reported vulnerabilities are inspected by security professionals before they get assigned CVE identifiers. The NVD retrieves all vulnerability information from the CVE list and is updated immediately when a new vulnerability is added. The NVD team analyzes the vulnerabilities and augments them with attributes such as affected software and versions, a base score, and an original release date, usually using the Common Vulnerability Scoring System (CVSS) 2.0.

The adoption of open-source software (OSS) has grown manyfold over the years. However, the open nature of these projects and public visibility of their development also makes them vulnerable to security risks, as malicious actors can target and exploit vulnerabilities in the code. The use of OSS has become increasingly prevalent in recent years and application developers frequently rely on third-party OSS components for common features, while focusing their efforts on the unique aspects of their applications. As a result, OSS is frequently integrated into modern software systems. The Open Source Security and Risk Analysis (OSSRA) report [3] by Synopsys revealed that out of 2,409 commercial codebases they analyzed, 97% contained open source components and 81% had at least one known open source vulnerability. Among those, 49% had at least one high-risk vulnerability. The Sonatype's 8th Annual State of the Software Supply Chain report [4] estimates that the four major ecosystems - Java, JavaScript, Python, and .NET - related to open source development are expected to surpass three trillion downloads soon. While the popularity and growth of OSS continues to climb, so are the malicious attacks against open source repositories which, according to the report, have risen by 633% year-over-year and 742% since 2019.

An interesting finding in the same report is that 1.2 billion vulnerable open source dependencies are reported to be downloaded each month but 96% of these vulnerable downloads actually had a fix/patch versions available at the time of downloads. There are possible reasons for delays on the part of users and software developers, and we will explore some of the reasons in human aspects of patch management in the subsection III-D

In this report we aim to provide an overview of the latest research efforts and highlight the standard practices used to identify and patch vulnerabilities in OSS. We also discuss the challenges of human aspects of securing software and provide recommendations for vendors and users using OSS libraries on how to mitigate these risks.

Section II gives software vulnerability related definitions and background literature search and the review protocol is discussed. In Section III, important papers in the field are grouped into aggregated themes and reviewed in detail. Section IV concludes this report and highlights key insights.

II. Background and Scope of Study

Vulnerabilities can be classified in various ways, but from a process-oriented perspective, a vulnerability life cycle begins when it is introduced (erroneously or maliciously) and this first stage is referred to as the injection stage. The discovery stage is when the vulnerability is discovered and reported, and the final stage is when a fix for the vulnerability is provided [5], although knock-on/ripple effects may mean that addressing the patch does not finish there.

When the decision to develop and deploy patches is made, it is often the result of a process that starts with the discovery of the vulnerability. When a vulnerability is first discovered, it is assigned a severity score, typically by a public organization such as NVD, which has a team of experts who assess the impact of each new vulnerability reported. The NVD receives its vulnerability listings directly from the CVE.

Different organizations then assess whether a new reported vulnerability affects any of their systems, attempting to characterize the nature of the vulnerability and whether an immediate effort is needed to address the issue or if a regular cycle of updates can be used to patch the vulnerabilities. Depending on the available human and technological resources, a remediation plan is decided, and a patch development and deployment plan is made. This report aims to review research papers that address the assessment phase of vulnerability patches in OSS, with a focus on papers that study the different challenges of developing and deploying patches. Since this is not an exhaustive literature review, we focus on papers providing empirical evidence to ascertain the practices adopted by practitioners.

A. Methodology

The research for this report involved an examination of relevant research papers and articles on OSS security, with the focus on patching behavior and factors affecting particular patching behaviors.

While the review was not a Systematic Literature Review the protocol was formalized to a certain degree into a three-stage process where 1) several search strings were generated, to be employed by individual authors in the group, 2) databases were then searched for relevant articles using these search strings and, when insufficient targeted papers were identified, 3) the references and forward-cited articles from the relevant papers identified were inspected for other relevant papers. Additionally contextual references were added as required. The three steps are now discussed in more detail.

Step 1: Design of the Search Strings: Several search strings were generated by the group in a day-long workshop held in Dublin. The basis for the search-strings was the 'Clarification for Literature Review (Milestone 13)' email sent by Lu to the group on the 27/12/2022. This email was first inspected for keywords and semantics, allowing us to derive a set of keywords for our search. Synonyms for the keywords were then identified and discussed, resulting in the following search strings:

1) "open source" AND "vulnerability" AND ("fix" OR "patch" or "repair") AND ("behavior" OR "behaviour" OR "practice")
2) "open source" AND "vulnerability" AND ("fix" OR "patch" or "repair") AND ("behavior" OR "behaviour" OR "practice") AND ("empirical" OR "case study")
3) "open source" AND "vulnerability" AND ("fix" OR "patch" or "repair") AND ("behavior" OR "behaviour" OR "practice") AND "embed"

Step 2: Searching the Database(s): We used Google Scholar and SCOPUS databases to search for papers using the search strings. We only considered those papers published after 2008 as candidates for further review. We then looked at the title and abstracts of the returned results and stopped looking at the search results when 20 consecutive papers (or 2 pages, in the case of Google Scholar) of search results were deemed irrelevant. The encompassing relevance criteria was defined as "patching practice in OS". From the candidate list of papers derived from this search, each author selected 8 papers for a more thorough reading and, out of those 8 papers each selected up to three top papers to be included in the report.

Step 3: Following References and Forward Citations: In cases where the reviewers were not able to identify three relevant articles using this protocol they identified relevant references or forward citations from the (relevant) articles they had identified, based on their full reading of those papers. From those relevant references they individually chose the paper(s) they thought most relevant to include in the review, to bring their papers-to-be-reviewed up to three per individual.

These three steps cumulatively, resulted in each member of the review team having three papers to review and summarize individually. Subsequent group discussions worked to abstract the papers into more encompassing themes across the literature concerning vulnerability patching (based on the individually-produced summaries).

III. Vulnerability Patching Themes Arising from the Literature

Based on the methodology presented in Section 2.1, the group identified a number of provisional themes across the selected literature. Over a series of meetings these themes were refined into four core themes that reflect the contents of the articles at a more abstract level:

- Characteristics of vulnerability patching (empirical perspectives);
- OSS community responses to vulnerability (as opposed to closed source projects);
- Use of automated tools for assessment and patching;
- Human aspect of vulnerability patching: Collaborative and individual approaches.

These four themes are now discussed in more detail.

A. *Characteristics of Vulnerability Patching (Empirical view)*

The first paper we selected provides some key characteristics of how vulnerability patching is conducted in open source software. In a large-scale empirical study of security patches [6], Li and Paxson had investigated the patch development life cycle. They analyzed 3,000 vulnerabilities from 682 different open-source software projects. Some relevant key findings of their reports are:

1) That vulnerabilities in code bases exist for prolonged periods. The median life span of vulnerabilities is over 438 days, with a quarter of them lasting over three years;
2) That security patches applied to a vulnerable code base are mostly localized to a single file, small in size (seven lines of code at the median), and are limited in their side-effects (59% of security changes reside in a single function). The implications of this insight is that security related patches are more amenable to automatic program repair methods;
3) That there is no clear correlation between a vulnerability's severity and its life span. Even though developers may be more motivated to fix severe vulnerabilities, the time it takes to discover and patch them is not affected by their severity. However, the type of vulnerability has an effect on its lifespan e.g. Web-oriented vulnerabilities such as SQL injection and cross-site scripting have shorter life spans compared to errors in software logic and memory management. Vulnerabilities like race conditions, numeric errors, and buffer overflows tend to remain undiscovered for longer;
4) The study found that 21.2% of vulnerabilities were not fixed when they were publicly disclosed. 78.8% of vulnerabilities were fixed by the time of public disclosure, and nearly 70% of patches were committed before disclosure. In the context of open source projects, the presence of patches before disclosure allows attackers who monitor these source repositories to take advantage of the fact that a vulnerability fixed in the patch is unlikely to be fixed by the end users before public disclosure, thus giving them ample time to develop and deploy exploits;
5) That developers do prioritize higher impact vulnerabilities in terms of addressing them before disclosure. For example, 88.1% of high-severity vulnerabilities were patched before public announcements.

Another empirical study that looks at OSS projects, but with a view to determine differences in patching behavior in different branches of OSS projects, is by Tan et al. [7]. They outline details of a large-scale empirical study they conducted on the stable branches in OSS projects. Their stated goal was to understand security patch management practice across stable branches. They investigated 608 stable branches belonging to 26 popular OSS projects, looking at more than 2,000 security fixes for 806 CVEs deployed on stable branches. They outline details of how security issues arise because of developers having multiple stable branches within a project and that when a patch is deployed on the mainline it may need to be ported, as opposed to deployed, on other stable branches. As the porting process is resource intensive and time consuming it results in delays to patches being issued.

The OSS projects reviewed in their study are written in C, C++, Java, PHP and Python and cover a range of different application types. They used the U.S. National Institute of Standards and Technology. 2021. National Vulnerability Database (NVD), https://nvd.nist.gov/home.cfm, to collect details of the vulnerabilities of the chosen OSS projects.

Some of the key findings of this study were that:

- "there are significant differences in the management of stable branches across the projects. This includes security patch management";
- "more that 80% of CVE-Branch pairs are unpatched".

As noted in the introduction, software developers and security practitioners rely heavily on public and private-operated databases to discover the new vulnerabilities. These databases are accumulating very large collections of vulnerabilities, along with information about how to identify the code vulnerability and patch the problem. These databases are very useful but are they always consistent and do they always contain correct information?

Existing research suggests that the information that can be found in these databases is not always accurate, up-to-date, and consistent between different databases. Xu et al. [8] conducted an empirical study of vulnerabilities found in the Veracode and Snyk industrial databases, which provide a "patch" field containing information on a software patch to fix the vulnerability. Each vulnerability has a CVE identifier from the Common Vulnerability and Exploit (CVE) database. They find that 45.7% of vulnerabilities have a patch listed in at least one of the two databases, indicating that a majority of vulnerabilities have no patch in either database. Further there are significant differences in the patches between the two databases. Only 19.7 of vulnerabilities have a consistent patch in both databases, and 69.2% of vulnerabilities are missing a patch in one or both databases. The remaining 11.1% of CVE IDs in the dataset have patches in both databases, but the patches are inconsistent. These inconsistent patches may arise because the vulnerability may have been fixed multiple times in the same software but in different repositories, or the same code might be fixed in multiple pieces of software that uses that code, or perhaps because the fix might be contained within multiple commits to the same repository. The authors manually inspected many cases of inconsistent fixes to the same CVE ID and found that in the great majority of cases the fixes were compatible and correct. Given the inconsistencies between two commercial databases that provide a publicly-available dataset, it seems likely that there is a problem with missing patches and somewhat inconsistent information across the many other databases that were not studied.

B. OSS community response to vulnerability (any difference to closed source projects)?

Schryen [9] examines whether open source communities and closed source software vendors differ in their approaches to addressing vulnerabilities in software. The author selected 17 diverse and widely used software projects in various categories, including operating systems, web browsers, email clients, office software, web servers, and database management systems. The vulnerability data for these projects was collected from the National Vulnerability Database (NVD). To determine whether the vulnerabilities had been patched, the study used vendor sites and vendor-neutral platforms like NVD, the MITRE site, the US-CERT Vulnerability Notes Database, and SecurityFocus. After analyzing the data, the author found that there is no significant difference in patching behavior between open source and closed source software. Instead, the patching behavior is determined by the policy of the development community or vendor, not the type of software development. A follow up paper [10] extends this study in two ways:

- It adds new data to the pool of patching data collected for their previous study, including data from sources that is not publicly accessible;

- This new data was used to examine the behaviour of vendors in terms of which vulnerabilities get patched.

They outline a number of clear hypotheses for the study. One is focused on what impact different development styles may, or may not, have on the patching behaviour. As part of this investigation they refer to two particular development styles, "Bazaar" and "Cathedral", which the define respectively as a development style in which "any volunteer can provide a source code submission" and "software is created by individual wizards and the development process is characterized by strong control of design and implementation".

The empirical results provided an outline of the severity of vulnerabilities along with details of patching behaviour. In the Results section they suggest that "…open source and closed source software do not significantly differ in terms of the severity of vulnerabilities and vendors' patching behavior." Regarding open source software developed in bazaar or in cathedral style, again, no statistically significant difference was found in the proportions of unpatched vulnerabilities. They additionally suggest that there is a need to provide "strong economic incentives for software producers to provide patches (at least for disclosed vulnerabilities)".

C. Use of automated tools for assessment and Patching

An important consideration for a developer or system administrator is to keep the down time of a running system to a minimum. The availability of tools that can help streamline the process of identifying, testing, and deploying patches is of great practical value. These tools can help ensure that patches are applied consistently across all systems, and can also help reduce the risk of errors or missed patches, but need to be comprehensive. A good, illustrating example is the scenario where an application may rely on an open-source library, which in turn may depend on other libraries or OSS components. These recursive dependencies, known as transitive dependencies, can often be the source of avoidable vulnerabilities as they are less visible to non-comprehensive security tools and audits. To mitigate this risk, it is essential to use tools or processes that can identify and audit all the dependencies in an application.

Due to the large number of open-source components being used across the industry, assessing the impact of vulnerabilities manually is a time-consuming and error-prone process. To address this, there are several tools available that help in detecting the use of vulnerable libraries, such as OWASP DependencyCheck [11] or the Victims Project [12]. These tools support checking whether a project depends on libraries for which there are any known, publicly disclosed, vulnerabilities by considering both direct and transitive dependencies.

One proposed approach for refining this analysis [13] is a code-centric approach which uses code analysis to determine whether the application actually makes use of the fragment(s) of the library where the vulnerable code is located. By having this information, if the automated assessment comes back with an answer that the vulnerable parts (methods, constructors, functions) of the OSS library are not actually called by the application, it will allow the vendor to schedule the update to the application with regular release cycle, thus saving the vendor the additional burden of developing and deploying patches on an urgent basis.

Shen and Chen [14] reviewed automated approaches to vulnerability detection, program repair, and fault prediction, focusing on deep learning approaches and their applicability to open-source software. For software vulnerability detection, the authors distinguish between code similarity and code pattern-based approaches. The former utilize clone detection techniques (the authors looked at 4 such approaches) and the latter use both static and dynamic analysis (the authors looked at 5 approaches here). Abstract modelling of programming languages, transfer learning, and improvements to deep learning models were highlighted by the authors as possible future improvements in this area. For automated program repair, grammar-based patching techniques (learning to adhere to language specifications) and semantic based patching techniques (conforming to the expected behaviour) are used. The authors claim that both types could benefit from the inclusion of structural code information into deep learning models. For automated defect prediction, the authors looked at within-project, cross-project, and just-in-time prediction techniques. The first two types could benefit from hyperparameter tuning, and for just-in-time prediction, CNN models can be used effectively. The authors summarize the following future directions: better feature generation, selecting appropriate neural network models for the task, the need for datasets and performance evaluation, and selection of feature parameters.

Islam et al. [15] conducted a survey of existing literature/approaches and proposed a taxonomy focusing on the runtime patching of mission-critical software systems. The proposed taxonomy has four major categories: patch granularity, patch strategy, patch implementation capability, and patch responsibility. Patch granularity characterizes the scale of patch and can range from fine-grained instruction-level to larger container-level, for example. Patch strategy is how a patch is applied to a running system. Patching systems may have one or several patching capabilities and these are reflected in the taxonomy: for example, memory management and resource transformation. Patch responsibility refers to entities that are responsible for patch application. These can be original software vendors, end-users, or third-party software patch developers. The authors highlight existing runtime patching challenges such as change compatibility, deployment overhead, and socio-technical issues (for example, empirical risk assessment).

Gao et al. [16] introduce Fix2Fit, a program repair system that seems to improve automated patch application. In test-driven automated program repair, candidate patches and test suites are generated automatically to select a suitable patch and to test that patch. However, such patch candidates can overfit a test suite and can still cause a crash (in the code not covered by the test suite). Fix2Fit prioritizes tests that can discard crash-inducing patches, as opposed to simply generating candidate patches and test suites. The effectiveness of Fix2Fit was evaluated through comparison with two other baselines (AFL/AFLGO) on 81 known vulnerabilities from 6 open-source systems. The results of the study suggest that Fix2Fit is able to discard a higher number of irrelevant patches in the same amount of time and generate more crash-free patches than the baseline approaches.

Xu et al. [8] propose a tool, Tracer (https://patch-tracer.github.io/), to automate the process of finding patches. It creates a graph that starts at the NVD, Debian, and Red Hat vulnerability advisory databases. Tracer extracts URL links from the advisory databases, such as links to blog entries, links to commits of fixes, existing patches, and discussions of the vulnerability. Where Tracer finds multiple patches to fix the problem, it ranks the patches based on the level of authority of the databases that link to the patch, and also the number of links to the patch from databases and other sources. Xu et al. evaluate their system and find that it provides high precision and recall for finding high-quality patches. The authors argue that Tracer is also a good tool to find weaknesses in vulnerability databases, and to automate the process of finding good patches.

Backporting is the process of applying patches and updates from new versions of software to older versions of the same software. In many cases older versions of the software are still being used widely within other software packages. In these cases it may not be practical to update to the newest version of the used package, but it is nonetheless important to fix security vulnerabilities and other problems in the older version of the used software. Decan et al. [17] conduct an empirical study into backporting practices on four widely-used package distributions (or sets of software packages): Cargo (Rust), npm (Javascript), Packagist (PHP) and RubyGems (Ruby). These distributions contain many individual software packages, and each package contains dependencies on libraries and other packages. When package A uses package B, then we can call package A the “dependent package” and package B the “required” package. They find a significant number of dependent packages in the distributions, that continue to use old versions of required packages, sometimes more than one major version earlier. This highlights the need to backport fixes to older versions. Unfortunately, a majority of the packages in their study are not maintained in this way, and backporting of fixes to earlier versions is not routine for most projects. They find that the packages that are most-likely to have fixes backported are those that are longer-lived and more widely used by other packages. Decan et al. found thousands of dependent packages that rely on old versions of software that contain vulnerabilities, where the vulnerability has already been fixed in a newer version.

There is a great deal of software that is used as a component within other software or with wider systems that cannot always be updated to the latest version for a variety of reasons. Thus, it is important that critical fixes are applied to older versions of packages that are still in widespread use. Shariffdeen et al. [18] study the case of the Linux kernel where most systems run an older stable version of the kernel rather than the very latest mainline version. They find that about 8% of patches to the mainline version of the kernel are backported to earlier versions. It is important to note that Linux kernel development rules allow a patch to be applied to an earlier stable version only if it fixes a major vulnerability, so there may be many patches to recent versions that are not eligible to be backported to earlier versions. They find that in 23% of cases the patch was applied to the old version without change. In 66% of cases the patch could be applied directly, but in a changed location, whereas 3% of cases require a change to the namespace and almost 8% require logical changes to the patch to make it applicable to the earlier version of the kernel. Where patches were backported to earlier versions, 80% took more than 20 days to be backported and 50% took more than 46 days. Shariffdeen at al. propose FixMorph [18], a tool and domain-specific language for backporting patches to earlier versions of software. Given a patch for a piece of code, FixMorph generates a transformation rule to transform the original to the new code. FixMorph then attempts to relax the transformation rule so that it becomes more generally applicable, but retains its ability to perform the transformation contained in the patch. FixMorph can then apply this new transformation rule to apply the original patch to earlier versions of the software. It uses compilation and existing test suites to validate the correctness of the patch to the earlier version. In a test of backporting 30 security patches to earlier versions of the Linux kernel, FixMorph was able to correctly patch 21 of the cases.

D. *Human aspect of vulnerability patching: Collaborative and individual approaches.*

As noted earlier, developers and system administrators may be reluctant to patch their systems based on the risks involved. Indeed, given the risks for developers/system administrators, they often seek the help of others before/when addressing vulnerabilities. Jenkins et al. [19] looked specifically at explicit support from the community during different phases of patch-related processes via the sys-admin mailing list PatchManagement.org (mostly targeted at Windows OS). Specifically they assessed how users engage with it and their type of information needs. The site generally communicates work-arounds (before patches exist), guidance regarding prioritizing of patches, discussion of post-patch ripple effects, tool selection and facilities discussion. But PatchManagement.org does have a restriction that vulnerabilities can only be posted when they are accompanied by a mitigation. This study focused on first messages only in threads, meaning that it is probably less focused on the fix over time.

The core context here is for system administrators to fix (patch) as fast as possible, and to do so with correctness. But their fear ripple-effects-after patching and that means that many systems (55%) are out of date. They argue that, to effectively patch, system administrators need high situational awareness and high collaborative skills to encompass the whole system and gather all the information/access-rights that they need. They also make the VERY valid point that sys-admins are typically stand-alone in their organizations, do very complex risk-assessment work that has large implications for their company, and that communities like PatchManagement.org are vital in that context.

Some of the ways a system administrator uses resources like PatchmManagement.org for the life cycle of a patch are:

- Vulnerability identification: typically alerting people to new threats or asking for help on new vulnerabilities identified;
- Error detailing/refinement: identifying ripple effects or seeking information when bugs are encountered;
- Updating error information: patching problems alluded to by others or asking for work-arounds;
- Updating error information: circulating potential errors as reported on other blogs;
- For patch-prioritization information like patch release-descriptions, platforms or severity;
- Updating query information: information seeking on a patch (for example is it still available?)

- Seeking how-to information and tools knowledge: seeking help to inform their current process;
- Scenario-based requests to leverage the knowledge of others;
- Sharing tool information: information sharing on tools or settings/tricks the user had found useful;
- Tool queries: information requests on tools for specific task (identification and usage);
- Mechanics and documentation information: seeking Windows Update Service adaptations and corrections of patch mechanic misinterpretations;
- Seeking information on update mechanisms;
- Seeking or providing insight on vendor behavior: gaining awareness of MS direction and their patching policy;
- Seeking or providing information on facilities provided by the PatchManagement community (mailing list rules/running information);

The authors also mention the importance of participating in a community like PatchManager.org to create a collective evidence for the source of a vulnerability, which can then drive a vendor like Microsoft to fix the issue.

In a similar collaborative vein, Wang and Nagappan [20] outline details of a large-scale empirical study they conducted to characterize and understand developers' interactions during their security activities regarding security introducing and fixing. As part of this they looked at more than 16K security-fixing commits and over 28K security-introducing commits from nine large-scale open-source software projects.

They claim this is the first study to "…characterize and understand developers' interactions in introducing and fixing security vulnerabilities by analyzing the developer networks built on their security activities." The 9 OSS projects chosen were from several existing published studies (details of which they outline in the paper). The projects were implemented in a range of programming languages, C/C++, Java, PHP, and JavaScript, and varied in terms of size, age and application domain. They identified 45K distinct developers from the 9 projects and, among these, those involved in security activities ranged from 3.98% to 50.39%, while those involved in non-security activities ranged from 49.32% to 83.77%.

In reviewing the projects included in their study, they look at whether each has the characteristics of a "hero-centric" project, which they defined as being projects where "80% or more of the contributions (i.e., number of commits) are made by around 20% of the developers)." These percentages were calculated by looking at developers' contributions in security. Interestingly, the authors state that while all of the projects they looked at were hero-centric, "when looking at developers contributions to non-security activities, most (8 out of 9) of them are non-hero-centric".

They summarized the finding of their study as follows:

1) "For non-security activities, developers that introduced and fixed bugs have low overlap rates with each other. However, for security activities, developers that introduced and fixed security vulnerabilities have higher overlap rates."
2) "Most of the experimental projects are examined as non-hero-centric projects in non-security activities, while all of them are hero-centric projects in security activities."
3) "The percentages of the developer interaction patterns vary dramatically in different projects. However, two patterns of interaction dominate across all projects: where two developers introduce the same security vulnerability and where a security vulnerability is introduced by one developer and fixed by another developer."
4) "The percentages of developer interaction patterns vary over time but projects do not witness a change in terms of the dominating patterns."
5) "Developers' interaction in security activities is correlated with the density of security vulnerabilities."

Another study [21] employed a data-driven approach to look at coordination problems in teams, across teams and across organizations. The authors report on 51 patch meetings, with 21 industry practitioners from 2 organizations/8 teams, in the health domain. These meetings focused on reporting security patch management status, tracking vulnerability remediation progress, discussing issues, planning patch cycles and decision making. Within their analysis, the authors focused on the role of 'coordination in security patch management', highlighting four dimensions:

1) **Causes (social and technical dependencies that define the need for communication)**;
2) **Constraints (factors that hinder the coordination):** These include legacy software-related dependencies, lack of automated tool support, and increased patch work-load;
3) **Breakdowns (patching failures due to ineffective coordination):** Example here include sudden escalation of patch schedules due to not tying down the dependencies/pre-requisites correctly in advance, delays in organizational approval and lack of dependency awareness across delocalized teams;
4) **Mechanisms (means of supporting coordination):** Examples here include early across-organization investigation of dependencies, collaborative decision making, continuous measures of progress, and globally-frequent meetings.

Li et al. [22] adopted a different perspective, looking at the individual behavior of system administrators. They conducted a survey and interview-based study to investigate the processes and actions of system administrators in managing software updates in sizable organizations. The research questions for the study were: 1) what processes do system administrators follow for managing updates, and 2) how do administrator actions impact the effectiveness of their system updates. The study aimed to provide suggestions for managing-update systems, as well as better-designed updates and policies.

The study identified five main stages of the update process:

- Learning about updates from highly dispersed (often non-canonical) information sources;
- Deciding to update (or not), based on update characteristics like whether it was a security/non-security fix and its severity;
- Preparing for update/installation: This involves making backups, preparing machines when necessary (configurations) and preparing for testing. Testing, in particular, is a pain-point for system administrators and some do no testing at all!
- Deploying the update: This can be manual, based on scripts (with manual support) or vendor-provided. (Unsurprisingly, most used a mix);
- Handling post-deployment issues. Frequently the system administrators back-out, or revert to snapshots/backups (leaving insecurities in place). Some persevere and look for workarounds instead.

The study also identified several challenges that system administrators face, such as difficulty acquiring comprehensive and meaningful information about available updates, difficulty effectively testing and deploying updates in a timely manner, and difficulty recovering from patch ripple effects. Additionally, the study found that organizational and management influence through policies had a mixed impact on the update process.

They provided several suggestions for addressing the identified challenges, such as a consolidated, standard centralized repository for vulnerability information and active notification, standardizing update information (such as severity) and avoiding bundling of security patches with regular feature updates. They also suggest conducting usability studies into tools. Additionally, the study recommended that management should recognize the importance of expedient updates (particularly for security issues) as that would help administrators perform their jobs more successfully.

IV. Conclusion

A. Current Status of and Risks for OSS

The scientific papers reviewed in this field reveal some important insights into the current status of, and risks associated with, the incorporation of OSS into commercial systems. They suggest that:

1) Security patching is a niche activity with typically only a small proportion of the development team taking part. A possible interpretation is that security-patching is an inherently-difficult, specialist task; This is an interpretation that is supported by the vast majority of the literature reviewed;
2) OSS has become an integral part of modern software development, but it also poses a significant security risk. Studies suggest that 95% of systems incorporate open source components and that they frequently (80%) contain vulnerabilities, of which about half are high-risk;
3) The open nature of OSS projects, the public visibility of their development and maintenance (including information related to bugs and vulnerabilities) is such that it may expose them to the activities of “bad actors”. A study by Li and Paxson [6] found that 21.2% of vulnerabilities were not fixed when they were publicly disclosed;
4) These vulnerabilities often persist in code bases for prolonged periods of time: their median life-span is over a year, and a quarter of them persist for over three years;
5) A specific issue is where a component (A) relies on an older, vulnerable version of another component (B), but a newer version of component (B) already has a patch. The solution here is to ”backport” the fix in the new-version of component B to the old-version of component B. But that seems to seldom happen. In the Linux kernel, for example, only 8% of patches in the newest version have been back-ported to previous versions;
6) Recursive, or transitive, dependencies are less visible to non-comprehensive security tools and audits. Practitioners should use tools or processes that can identify and audit all the dependencies in an application.

B. Suggestions for Practitioners

Based on these risks/status, this section proposes several community-of-practice and tooling guidelines derived from the literature. In terms of communities-of-practice:

1) Repositories that leverage the power of the crowd (like PatchManagement.org for Windows OS) are helpful resources, identifying work-arounds, categorizing the priority of patches, determining ripple effects of fixes, and facilitating tool selection. Indeed these repositories are also useful vehicles for prompting vendors to react to newly discovered vulnerabilities. Given the product-specific nature of PatchManagement.org, this literature review suggests that other such repositories, directed at different products, should be created.
2) Important databases in the field like CVE and NVD can provide a valuable resource for practitioners. They identify consistent patches for 89% of the vulnerabilities they store and, even though the other 11% are inconsistent across these databases, the patches they propose still seem effective.
3) Building on these databases towards systems that can proactively notify subscribers of new vulnerabilities and patches would seem like the logical next step in the evolution of these databases;
4) Strong economic incentives should be provided to motivate software providers to issue patches (at least for disclosed vulnerabilities);
5) It’s important that older versions of packages are considered (especially those that are still widely used) when critical fixes are issued and that fixes are “backported” where possible.

In terms of tooling:

1) And again with reference to CVE and NVD, ”Tracer” like tools that digest and augment the information they contain should be developed. Such tools can present users with a consolidated view of the relative quality of proposed patches (from various aggregated sources) and present a consolidated discussion space for the vulnerability.

2) It is important that tooling can cope with transitive dependencies, where components rely on components that in turn rely on other components
3) Tools should also support fine-grained analysis of source code to see if the application under study makes use of specific code fragments where the vulnerability exists, not just the associated overall component.

References

[1] Mitre. (2022) Common vulnerabilities and exposures. [Online]. Available: https://cve.mitre.org/

[2] NIST. (2022) National vulnerability database. [Online]. Available: https://nvd.nist.gov/

[3] Synopsys. (2022) Open source security and risk analysis report. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf

[4] Sonatype. (2022) 8th annual state of the software supply chain. [Online]. Available: https://www.sonatype.com/state-of-the-software-supply-chain/introduction

[5] V. M. Igure and R. D. Williams, "Taxonomies of attacks and vulnerabilities in computer systems," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 1, pp. 6–19, 2008.

[6] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[7] X. Tan, Y. Zhang, J. Cao, K. Sun, M. Zhang, and M. Yang, "Understanding the practice of security patch management across multiple branches in oss projects," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 767–777.

[8] C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, "Tracking patches for open source software vulnerabilities," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 860–871.

[9] G. Schryen, "A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors," in *2009 Fifth International Conference on IT Security Incident Management and IT Forensics*. IEEE, 2009, pp. 153–168.

[10] G. Schryen and E. Rich, "Increasing software security through open source or closed source development? empirics suggest that we have asked the wrong question," in *2010 43rd Hawaii International Conference on System Sciences*. IEEE, 2010, pp. 1–10.

[11] O. Foundation. (2023) Owasp dependency-check. [Online]. Available: https://owasp.org/www-project-dependency-check/

[12] Various. (2013) The victims project. [Online]. Available: https://github.com/victims

[13] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 411–420.

[14] Z. Shen and S. Chen, "A survey of automatic software vulnerability detection, program repair, and defect prediction techniques," *Security and Communication Networks*, vol. 2020, pp. 1–16, 2020.

[15] C. Islam, V. Prokhorenko, and M. A. Babar, "Runtime software patching: Taxonomy, survey and future directions," *arXiv preprint arXiv:2203.12132*, 2022.

[16] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.

[17] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the past – analysing backporting practices in package dependency networks," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4087–4099, 2022.

[18] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 633–645. [Online]. Available: https://doi.org/10.1145/3460319.3464821

[19] A. Jenkins, P. Kalligeros, K. Vaniea, and M. K. Wolters, ""anyone else seeing this error?": Community, system administrators, and patch information," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 105–119.

[20] S. Wang and N. Nagappan, "Characterizing and understanding software developer networks in security development," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 534–545.

[21] N. Dissanayake, M. Zahedi, A. Jayatilaka, and M. A. Babar, "A grounded theory of the role of coordination in software security patch management," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 793–805.

[22] F. Li, L. Rogers, A. Mathur, N. Malkin, and M. Chetty, "Keepers of the machines: Examining how system administrators manage software updates for multiple machines," in *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, 2019, pp. 273–288.