

Security Evaluation Criteria of Open-Source Libraries

Vivian Mills and Sergey Butakov(✉)

Concordia University of Edmonton, Ada Boulevard, Edmonton, AB 7128, Canada
{1vivian.mills,sergey.butakov}@concordia.ab.ca

Abstract. The use of freely available, open-source code to reduce the time needed to create new software or add functionality to existing software is a common practice. With analysis of recent high-profile cases of open-source software packages being corrupted by the original developer, or the introduction of remote back-door functionality by malicious actors, it has been shown that there is much that can be done to help with simplifying the decision-making process of using any open-source code. This paper provides the basis for a simple-to-use checklist that can be used to quickly analyze open-source libraries for its suitability within an individual's or organization's code base. Fourteen projects were selected at random from a popular code hosting site that made use of specific biometric security libraries. The conclusions derived from the use of the checklist and the analysis of the selected projects will help with simplifying the decision-making process of using open-source code for software projects.

Keywords: Software security · Risk management · Supply chain attacks · Modern code review · Vulnerability management

1 Introduction

The number of public announcements regarding the corruption of popular open-source source code (OSSC) projects [1], or the discovery of back-doors [2] in popular packages has become more of a worry for developers and organizations [3, 4] that rely on OSSC to do their business. For example, in [1], the developer appears to have intentionally introduced issues to the libraries with the reasons not being clear as to why and having theories ranging from monetary compensation to a hacker protest. Shortly after that incident that particular project repository was forked and the development was taken over by a team of maintainers [5]. For [2], it is a case of a commercial company, that develops themes for a popular content management system (CMS), whose own systems were compromised at some point. From there their themes and plugins for that CMS were corrupted with code to allow attackers with backdoor access to the sites that use these source code objects. It could be argued that a theme or plugin should not be a high-security concern but any time you place code into a system it should be reviewed. While WordPress (WP) has had a number of security issues in the past, it was suggested that the main reason that these themes were able to do what they did, could be due to

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022 Workshops, LNCS 13381, pp. 422–435, 2022.
https://doi.org/10.1007/978-3-031-10548-7_31

any number of reasons like the use of older plugins, the misconfiguration of the instance permissions on the server, or not having a method of monitoring the instance files and configuration for changes [2]. These, along with regular audits of any third-party plugins have been recommended to help secure WP instances [6, 7].

Many users of free source code packages are not large companies that have dedicated software development teams, nor do they generally have the capability to scan the code for security issues. Scanning source code for security issues is a specialized skillset that requires vast amounts of time, or money for the specialized software to automate those tasks, or both. It was for these reasons that a different approach was developed to assist in reducing the amount of effort needed to begin the evaluation process of OSSC projects. The target audience of the proposed checklist is intended to be developers who may not be trained in security analysis of code or quality assurance teams that need to develop a code review tool tailored to the specific needs of the company. The research focused on reviewing OSSC library projects, found on GitHub, for the iOS local authentication Application Programming Interface (API). Specifically, it is focused on the Touch ID and Face ID, biometric authentication features of the Operating System (OS) and devices that support associated biometrics hardware.

The scope of the research was specifically chosen to match what could be reasonably expected of a developer to focus on when looking for a library to use. Specifically, for this project, the Local Authentication API for iOS was selected. There were a number of criteria that this particular feature set met, those included Biometric security, mobile devices, and it had to be an OSSC project found on GitHub. These requirements were chosen to help limit the number of project repositories that could be found on GitHub, but also to bring a level of real-world similarity to what an organization, or single developer, would be doing when starting to look for library projects that met their needs for a given task. The list that was eventually selected for this paper was not exhaustive, nor completely free of issues. For example, during the selection process it was discovered that a number of project repositories were forks of another project already in the list, with minor changes from the original project repository. Given the relatively small scale of this research project, its scope was limited to a small number of repositories that were found on GitHub that claimed the use of the iOS API criteria outlined earlier. From that starting point, using the software developed to help speed up the evaluation process, the repositories were analyzed using the criteria outlined in the following sections of this document.

The rest of the paper is organized into the following sections, Sect. 2 further reviews the previous work on analyzing OSSC Biometric API support. In Sect. 3, the non-code analysis part of the checklist is further defined and with justification provided. The results of the gathered research data are shown, where it is analyzed and broken down to show the patterns of what the checklist can expose when given evaluation criteria. In Sect. 4 we conclude the paper and suggestions for future research directions.

2 Related Works

The amount of work done in the field of code security and secure development methodologies is significant. With numerous organizations and researchers writing whitepapers or frameworks that can be used to help provide direction with security analysis or

supporting the decision-making process with respect to Risk Management in software development.

The National Institute of Standards and Technology (NIST) has developed documents such as Recommendations for Mitigating the Risk of Software Vulnerabilities (NIST SP 800-218) [8], detailing the Secure Software Development Framework (SSDF), or the Security and Privacy Controls for Information Systems and Organizations (NIST SP 800-53) [9].

Primarily, the SSDF framework is intended to help with the risk management factors of business risk mitigation. Both NIST standards can be used by independent developers but generally takes a lot of effort to make the best use of them given the size and scope of the frameworks. The SSDF does have a guideline that details a "subset of what an organization may need to do" [8]. The items in the guideline provide reference referrals to other documents like NIST SP 800-53 (SA-1, SA-8, SA-15, and SR-3). That these items lack are specific directions that can be modified or adjusted to specific needs. For example, in Table 1 of the SSDF, the first task is "Identify and document all security requirements for the organizations software development infrastructures and processes and maintain the requirements over time." Further on in the Practices column is the item "Reuse Existing, Well-Secured Software When Feasible Instead of Duplicating Functionality" where it essentially recommends that the user "Acquire and maintain well-secured software components … from commercial, open-source, and other third-party developers for use by the organization's software" [8]. Most smaller development departments do not maintain their third-party code themselves as they do not typically have the resources or ability to do so, except by continuing to acquire newer updated copies of the code on a regular basis.

By design, the NIST frameworks and the task items found in them are developed to be very generic in the guidance and direction for any organization that has the ability to look at and understand those frameworks. The SSDF is an exhaustive framework and implementing it would require a set of dedicated resources that most single developers, or even small development shops, would not have access to. It makes recommendations and provides reasoning for what it is suggesting with defined examples for every task, but because of the number of tasks and references to other frameworks or documents it makes it very challenging for someone to implement and follow on a regular basis.

There has been other research done using similar criteria as those defined in the evaluation criteria proposed in part 1 of the methodology. In [10], Thompson and Wagner, also used GitHub to review OSSC projects but from the perspective of the second part of the checklist, code reviews, where they attempted to create better policies for conducting code reviews for the code itself. As such the focus of Thompson and Wagner was more on the security aspect of using OSSC as a practical concept of the code review process [10]. While they focused on the actual code, their use of the GitHub APIs to look for correlations between the different statistics of a project like the numbers of stars, pull requests, contributors, and issues to try and identify what they could mean when compared to the quality and security of the code. Specifically, they focused on the relationship between code review coverage with the number of issues and issues that are security bugs. In [15] the focus was on the analysis of the literature available that discussed ways to evaluate the OSSC and models for measuring the quality of the

code. This is different than the approach taken by this paper in that the outcome of this paper should be a customizable and measurable way to analyze OSSC projects to cover quality, security, and usability of the code. For [16] the approach on the surface appears to be similar to the developed checklist for this paper, but their outcome was to help with the selection of existing evaluation frameworks to select open-source applications not libraries. This once again differs from this paper since the focus of this paper is to produce a workable checklist that can be used to analyze OSSC libraries for inclusion in a software project, rather than the selection of an application.

The goal of the research being presented in this paper is to provide a practical method that helps bridge a gap for individuals or small organizations who may not have the same resources yet still leave the thresholds for usability up to the team using it.

3 Proposed Approach

3.1 Methodology

The evaluation process methodology developed in this paper consists of 2 main parts with multiple items in each section. The first part is based on the evaluation of code repositories from a management and risk assessment point of view for characteristics such as freshness, number of contributors, and responsiveness of the developer(s). The research, data gathering, and evaluation consists of a number of smaller steps.

1. GitHub REST API – Researching what API calls were available and what data was publicly available from those calls.
2. Criteria evaluation rules – The rules used to determine the pass/fail values that the software tool would use to evaluate the selected repositories.
3. Software tool – Creation of a software tool that used the API calls and the evaluation criteria to gather the data for evaluation and comparison.
4. Build a list of target repositories – The repositories were filtered and selected based on the criteria of iOS, Touch ID/Face ID, and a library for other developers to use.
5. Gather the data on the repositories – The use of the developed software was key in gathering the data on the selected repositories.
6. Dataset evaluation – The task of evaluating the data and determination of the overall pass/fail of a given repository, to be completed by the user of the tool.

The second part of the evaluation process, the actual code review, was left for the reason that it had already been covered in other research, like that found in [10, 11] and further enhancements were out of scope for this project. Given that the initial intent to develop a checklist was more focused on the iOS Touch ID and Face ID API calls and how they were being used, the research could be expanded to any OS security API call, but for the sake of brevity the Touch ID and Face ID API calls were chosen. While the original method was designed to have a good balance between the steps, it was discovered that there was an imbalance in the original weightings of Step 1 and Step 2. One of the repositories reviewed was found to have an issue where it was no longer working on a later version of the iOS software, that the bug was more than a year old,

and that it was still open. The original weightings did not allow for the repository to be immediately disqualified for use.

The underlying basis for the checklist is to provide a measure of confidence for the evaluation of OSSC projects that are under consideration for inclusion in application software being developed by an individual or organization. Given the criteria that is found in NIST SP 800-218 the checklist is a useful companion methodology for initial evaluation of risk management.

The SSDF tasks listed in Table 1, is not a complete list of items found within the SSDF that the checklist covers. However, it does highlight why the checklist, when used, can help to provide a non-technical evaluator to a decision that can be aligned within a Risk Management framework if needed.

Table 1. Mapping of Secure Software Development Framework (SSDF) items to the CHECKLIST STEPs.

SSDF task	Checklist	Details
PO.1.1 PO.1.2	Evaluation criteria for the checklist	Review all security requirements at least annually. With every OSSC project update, security needs to be reviewed
PS.1.3	Step 1e, Step 1g	When defining the evaluation criteria that is acceptable for evaluating OSSC projects, the end-user is setting boundaries for attributes like documentation requirements and vulnerability disclosures
PO.3.1	Step 1i	Use of automated security tools and review of results
PO.3.3	Results of using the checklist	Use existing tools to create an audit trail of secure development-related actions. By keeping track of the results of the checklist an audit trail can be kept between updates of the OSSC projects
PO.4.1	Evaluation criteria for the checklist	By defining the evaluation criteria, the risk appetite is defined. When the checklist is used with the evaluation criteria it is a step in managing that risk
PS.1.1	Step 1b, Step 1d	Overtime, the continued use of the checklist to monitor the contributors and code releases of the OSSC project, and with thorough auditing of the results to possibly identify patterns where a project might be in trouble or not adequate to the risk management requirements
PW.1.1	Step 1g	Reviewing the open issues of the OSSC project can highlight issues that may be of concern to using the code
PW.4.1	Step 1c, Step 1e, Step 1h	Evaluating a OSSC project, attributes like provenance needs to be considered. Other characteristics like is it a forked project, has it been archived and superseded, and is it up to date and compatible for what it's intended use is (iOS 12 vs iOS 15)

3.2 Evaluation Criteria

This is the first part of the non-code analysis checklist and should be considered the initial sniff test of a library project. Any project that cannot standup to the initial scrutiny of the following items should not be considered for use in a project that is being built for production use. The following items are not in any predetermined order of importance. This part of the analysis process should be considered more of a holistic view of what should be considered for further analysis if it passes the preliminary test. There may also be the possibility for this section to be automated using a REST API through the GitHub website, further research will be required.

Items **a** to **i** of Step 1 in checklist:

a. **Age of the Project** – The age of the project repository will be a good indicating factor of longevity and stability. The more mature the code base the greater the complexity of the code conversely, less maturity will mean it is easier to understand the code as there is less cruft [12].
b. **Code Releases** – This means that active development is occurring, and the likelihood of bugs being fixed is higher than a stale project. This should be considered a critical section when being considered with the results of the Project Freshness. The speed of releases and time between releases can also be significant [13].
c. **Project Freshness** – The freshness of a project repository can be gauged by the support for the latest versions of the OS or APIs that is supported. A stale project will be a couple of versions behind the current production version or programming language version. This should be considered a critical section when combined with Code Releases. If the Code Releases and the Project Freshness is too far out of date or the targeted OS version is more than a major release version apart this should be a failure for the entire analysis section. It might be mitigated by the Code Analysis section if the API calls being wrapped by the library has not changed between major release versions. Currently there is not a good, automated way of determining this with any accuracy with an automated tool. This should be completed by the user of the checklist.
d. **Contributor Strength** – If the library project is supported by multiple developers, then the project is likely to survive loss of team members. If the project is supported by only a single developer, then the likelihood of it becoming abandoned is greater. This does have a negative side in that the more contributors there are the more likely that something could be slipped into the code as well [13].
e. **Documentation** – Documentation does not have to be written pages but may include code examples on how to use the library. Similarly, to Item C, there is currently no good, automated way of determining this with any accuracy with an automated tool. This should be completed by the user of the checklist.
f. **Popularity** – While popularity is not something that should be used by itself, when combined with the other analysis items of this section, it can be useful measurement of the project whereby the more popular the library is, the less likely it is to have glaring errors [10]. At least ones that have not already been discovered and listed in the issues of the project's repository
g. **Open Issues** – The number of open issues registered on the project may be an indicator of the state of the project. Any decently sized project may have open

issues, but the age of the open issues and the responsiveness of the developer on those issues will need to be considered when evaluating the library. This should be considered a critical section depending on the age and severity of the issues. Failing this section should be a failure for the entire analysis [14].

h. **Repository State** – If the repository is marked as archived then it should be an immediate failure for any use in future projects.
i. **Automated Code Analysis** – One of the new features introduced by GitHub is the ability for any public repository to look for known code issues like those found on the Common Weakness Enumerations website (https://cwe.mitre.org). If this is available for a given repository it should be considered a critical part of the analysis that should being considered. If the repository doesn't have this turned on, then it should be considered a failure.

The above checklist items are comparatively similar to items proposed in the research of Thompson and Wagner [10] for filtering their initial dataset. In their paper they examine the code reviews for OSSC on the GitHub platform and the filtering criteria they considered regarding the repositories that they were including as a part of their review process included items like the creation date when combined with the last push date to determine if they had an actively developed project or a just a place holder. Others included a minimum number of issues (>5), contributor strength (>4), and other items such as the project age and the repository language details [10]. While some of these items are not strictly related to programming, Thomson and Wagner showed that their repository filtering rules for applicability to their paper were similar to how the use of the items in the checklist can be used. When used to filter repositories for evaluation that are being considered for inclusion into a software project. Since their paper has to do with code reviews and software security their work and the work provided by this checklist could benefit each other when used together.

3.3 Analysis

Table 2 details the specific evaluation criteria used in determining the pass/fail rate of each checklist item for the library projects under review. While the factors listed here are what was used during the testing process for this paper, it should be noted that everything on the checklist should be adjusted to suit the needs of the organization or individual using it, ultimately basing it on their risk appetite and other factors that would be unique to them. By allowing the flexibility of defining the evaluation criteria based on risk appetite it allows the user to change the way that the criteria can be considered. For example, if the OS that is being targeted by the code has not had a major change in more than a year, the number of updates to the repository may not meet the criteria as it was defined, for example, in this paper. By changing the focus of that step to having a longer time frame, it decreases the weighting on that step which then increases it on other steps.

Table 2. Specific evaluation criteria used in the checklist analysis.

Checklist item	Additional testing criteria	Explanation
Step 1a	Within the last 5 years	Limit the project repositories that were no older than 5 years. Ozment and Schechter found that the number of foundational vulnerabilities decreased with age, and that new vulnerabilities were reported and code was added. "The average age of foundational vulnerabilities have a median lifetime of 2.6 years." [12]
Step 1b	Within the last 12 months	Limit to actively developed/updated repositories. The more active a project is with code churn, the more likely it is to have issues occur [13]
Step 1c	Not assessed (Not automated)	This item left to the end user of the checklist to determine
Step 1d	More than 1	The more developers that actively work on a project the more likely it is to survive a developer leaving it. For single developer projects the likelihood that the project would become abandon-ware is significantly higher. It also can increase the number and quality of the code reviews and architectural decisions [10]
Step 1e	Not assessed (Not automated)	This item left to the end user of the checklist to determine
Step 1f	More than 15 stars	The higher the number of stars a project has the more likely it is to continue to be actively developed. It also means the more people who would report issues [10]
Step 1g	Open issue count/project age is less than 100	The number of issues is not a clean target but the number of issues when combined with the age of the oldest and newest issues and the volume of issues brings a clearer picture of projects that work to maintain and control issues
Step 1h	(Pass/Fail)	Any archived project should be considered an automatic failure
Step 1i	Automated testing in use	Since GitHub.com has several fully automated tests that can scan a repository for common vulnerabilities each repo should consider turning that feature on. Currently, the major issue with this feature is that Github.com does support more than a half dozen specific languages for such scanning and none of the repos that were chosen for evaluation used one of the supported languages

While it may seem that injecting additional testing requirements to the checklist items should not be necessary, it must be noted that any risk analysis that an organization undertakes for any system that they use, will always be unique to that organization. Thus, it makes sense that defining of testing criteria for the first part of the checklist should also have those same steps taken when considering the use of OSSC for inclusion on internal projects. It may not be common for a single developer to do a risk analysis, based on NIST 800-218, of someone else's code prior to using it. Evaluation software having user definable levels of acceptance criteria would help to highlight the possible need for individuals to make use of it.

Appendix B provides a summary output of the application that produces Part 1 of the non-code analysis portion of the checklist that was generated when the analysis was run on the fourteen repositories selected for evaluation.

All fourteen repositories were evaluated using the application developed. The method of selecting the projects that were used came from finding public projects that made use of the Face ID and Touch ID security API features offered by iOS devices. The application was then used to evaluate all the repositories which then generated the raw data found in Appendix B.

For example, a single project repository, ***suxinde2009/touchidkit,*** it passed five of the seven test cases. When manually reviewing the two test cases that were not automated, it passed an additional item to bring its total up to six of nine. Based on the outcome of the checklist and the evaluation criteria used during the research for this paper, this project repository is a nearly successful candidate for moving on to the second part, or code review, of the checklist.

By reviewing the results of each checklist item for this project we can see the overall details of the ***touchidkit*** project:

Step 1a – The project creation date was February 26, 2020, this is within the past five years. (Passed)
Step 1b – The project had a check-in on May 30, 2021, this fell within the 12-month timeframe that was set. (Passed)
Step 1c – The project does not list what version of the programming language or OS version it is intended for in the repository details. (Failed, manually examined)
Step 1d – It appears as though there has not been another contributor to this project beyond the initial creation and check-in. (Failed)
Step 1e – The developer provided both documentation and some sample code. (Passed, manually examined)
Step 1f – Twenty stars compared to the required fifteen stars. (Passed)
Step 1g – There were no open issues, with 1 closed. (Passed)
Step 1h – The project repository is marked as Active. (Passed)
Step 1i – There is no apparent automated testing setup on GitHub. Simply due to the fact that Swift is not currently a supported programming language. (Failed)

One mark against this project that is not as well highlighted is the fact that there has not been a code update for over two years. This is where the flexibility in the design of the checklist comes into play. Not having a code check-in for so long should have affected its Code Release (Step 1b) score but it did not. The reason for this is due to

the fact that the readme.md file for the repository was updated and counts as a check-in. However, since there are no outstanding issues and the popularity is high enough to meet the evaluation criteria for the project, it should be considered for the code review process. That failure to identify a non-code check-in could be corrected in a future version of the software developed for this research.

3.4 Discussion

In their paper, Thompson and Wagner talk about the Modern code review as being the heavyweight process of formal code inspection [10], and it is essential that the code review is done on any OSSC that is to be brought into a project even if that code is worked on directly by the individual or team that is looking to use it. To get to that point they setup and used several evaluation criteria similar to the checklist which they used to generate a base list of 48,612 candidate repositories, from that list they randomly selected 5000. They then used a second set of evaluation criteria to further refine their dataset down to 3,126 repositories. All of the filtering they used can be matched onto similar items of the checklist. What stands out from their list of filtering criteria, that differs from the checklist, are the attributes of the repositories like popularity (Stars), or that a project may have been Archived. While that may not have mattered to them for the purposes of their research it points to a type of blind spot that should not be overlooked. The checklist explicitly uses these items to try and raise the awareness that using a project that has been Archived by the maintainer may be problematic in the future. Similarly, a project that does not have any interest from the rest of the development community may not have the same forward momentum to continue to be developed by the maintainer. Either item could highlight a possible issue for use of the OSSC in the future. By using several of the same criteria, in a similar fashion to the steps being proposed in the methodology of this paper, it demonstrates that there can be useful datapoints extracted that can help to winnow out OSSC projects that should be avoided without starting with the code review first.

The summarized values in Table 3 does not paint a great picture of the fourteen OSSC projects that were selected for evaluation. In two of the nine checklist items we can see that the number of repositories that passed the evaluation criteria was one or less and that in most of the checklist items less than half of the repositories were considered to have passed the testing. The summarized numbers, while they look significant, are meaningless without a greater understanding of what the checklist items evaluation criteria were, and that they need to be further broken down to gain a full understanding of what the numbers mean.

For Step 1a, thirteen of fourteen projects passed, but to determine if the chosen criteria was valid would depend on what the requirements are for the end-user. Choose a cutoff date that is too short and the results will only show very young projects. Does that mean that there could be an inherent bias against mature projects or vice versa? There is no simple answer to that question since it will differ between projects, as well as organizations, using the checklist. When looking at the results for Step 1b, where only four of the fourteen projects passed, it could be inferred that the majority of the projects may have been abandoned since the last check-in was outside of the established criteria. This inferred result can also be misleading in multiple ways. For example, the project

that was reviewed in Sect. 3.2 also did not have a code check-in within the criteria period, yet it was marked as having passed. This is an established issue within the developed software used but could be corrected by looking for code changes vs any other type of changed document. It leads to the question of what would be considered a valid code check-in. Should the evaluation criteria also look for specific source code file changes and then the number of lines modified to match a minimum requirement? What if the only lines that have changed are comments? It could be viewed that any modifications show that the code is still being maintained and thus could meet some evaluation criteria of the organization.

The rest of results in Table 3 can also be viewed in the same manor, but what can be extrapolated from those results is that the evaluation criteria are critical to the use of the checklist and that the amount of effort spent in defining those criteria is going to pay off when used appropriately. Further to the effort of properly defining the evaluation criteria is the weighting of every item in the checklist to each other item. If a project passes six of nine items, but fails on what is considered a critical item does that failure negate the six passed items? What each user, or organization, who uses the checklist must understand is the weighting of each item and balance that with their risk appetite and risk mitigation strategies. If the team is willing to accept that risk then there is no reason that a single failed, heavily weighted, item should immediately negate a project evaluation.

Table 3. Checklist item evaluation rate as defined by the evaluation criteria.

Checklist items	Pass rate
Step 1a	13/14 (92.9%)
Step 1b	4/14 (28.6%)
Step 1c	Not assessed
Step 1d	1/14 (7.1%)
Step 1e	Not assessed
Step 1f	5/14 (35.7%)
Step 1g	9/14 (64.3%)
Step 1h	14/14 (100%)
Step 1i	0/14 (0%)

4 Conclusion

In view of the general availability of open-source source code library projects, the use of such code becomes more likely in the projects of a single developer or a small team of developers. The issues that arise from relying on such code are numerous. With corruption of the source by the developer or from a malicious third-party actor to a lack of on-going maintenance it becomes important to have a methodology that can be used to evaluate every third-party open-source library. The proposed checklist should be looked

at as a tool that needs to be adjusted and refined by the user of the tool to reflect the risk appetite, and development practices that are acceptable to the development team. The target audience is not strictly intended to be the developers themselves, but anyone who needs to evaluate source code for inclusion into any project.

The checklist, as it is proposed, should be considered as a part of a larger risk management process. In both the NIST SP 800-53 and SP 800-218 frameworks, there are multiple steps of risk management from the analysis and appetite determination to the mitigation processes. The creation of the checklist was originally intended for software development, but the general use for such a system is not limited to the examination of software. It could be universally extended and developed for any kind of system where an evaluation by an individual not trained or experienced in doing such an analysis could be developed and implemented.

When the checklist is used in conjunction with a proper modern code review process for software development, the overall number of OSSC project repositories to consider should be reduced. This should have an overall positive effect on the time to evaluate OSSC repositories and risk level of using them. It is important that the determination of the categories and fundamental testing criteria are predetermined to match the needs of the implementors of the checklist to get the most benefit. There are at least two possible future research paths to improve the checklist, one would be to further refine it for the use in software development such that it could be applied to commercial software source code. The second would be to look at creating a generalized checklist for use in other industries that utilize open source approach to the development.

Appendix A: List of Repositories Used in Analysis

See Table A.1.

Table A.1. List of repositories used in analysis.

Framework	Description
https://github.com/g212gs/LocalAuthentication	iOS API example
https://github.com/roninprogrammer/BiometricAuthentication	iOS API example
https://github.com/gabrieltodaro/BiometricExamples	iOS code example. Additional code for storing user id/password in local keychain system
https://github.com/AppliedRecognition/Face-Template-Utility-Apple	This application template uses the following project library. It also demonstrates how to use the library
https://github.com/AppliedRecognition/Ver-ID-UI-iOS	Advanced middleware security library. Providing more features that the base API

(*continued*)

Table A.1. (*continued*)

Framework	Description
https://github.com/expo/expo	Entire framework that can build apps on either iOS or Android using React
https://github.com/rushisangani/BiometricAuthentication	Security middleware library
https://github.com/vasilenkoigor/BiometricAuth	Security middleware library
https://github.com/tejas-ardeshna/TJBioAuthentication	Security middleware library
https://github.com/yankodimitrov/SwiftPasscodeLock	Security middleware library
https://github.com/Meniny/BioAuthenticator	Security middleware library
https://github.com/BabyShung/SplashWindow	Security middleware library
https://github.com/suxinde2009/TouchIDKit	Security middleware library
https://danielsogl.github.io/capacitor-face-id/	Security middleware library

Appendix B: Summary Results of Repository Analysis

Table B.1. Summary results of repository analysis.

Repository	Step 1a	Step 1b	Step 1c	Step 1d	Step 1e	Step 1f	Step 1g	Step 1h	Step 1i
g212gs/localAuthentication	Pass	Fail	n/a	Fail	n/a	Fail	Pass	Pass	Fail
roninprogrammer/biometricAuthentication	Pass	Fail	n/a	Fail	n/a	Fail	Pass	Pass	Fail
gabrieltodaro/biometricExamples	Pass	Fail	n/a	Fail	n/a	Fail	Pass	Pass	Fail
appliedRecognition/face-Template-Utility-Apple	Pass	Fail	n/a	Fail	n/a	Fail	Pass	Pass	Fail
appliedRecognition/Ver-ID-UI-iOS	Pass	Pass	n/a	Fail	n/a	Fail	Pass	Pass	Fail
expo/expo	Pass	Pass	n/a	Pass	n/a	Pass	Fail	Pass	Fail
rushisangani/biometricAuthentication	Pass	Pass	n/a	Fail	n/a	Pass	Fail	Pass	Fail
vasilenkoigor/biometricAuth	Pass	Fail	n/a	Fail	n/a	Fail	Pass	Pass	Fail
tejas-ardeshna/TJBioAuthentication	Pass	Fail	n/a	Fail	n/a	Fail	Fail	Pass	Fail
Yankodimitrov/SwiftPasscodeLock	Fail	Fail	n/a	Fail	n/a	Pass	Fail	Pass	Fail
Meniny/BioAuthenticator	Pass	Fail	n/a	Fail	n/a	Fail	Pass	Pass	Fail
BabyShung/SplashWindow	Pass	Fail	n/a	Fail	n/a	Fail	Fail	Pass	Fail
suxinde2009/TouchIDKit	Pass	Pass	n/a	Fail	n/a	Pass	Pass	Pass	Fail
danielsogl.github.io/capacitor-face-id	Pass	Pass	n/a	Fail	n/a	Pass	Pass	Pass	Fail

References

1. Roth, E.: Open source developer corrupts widely-used libraries, affecting tons of projects - The Verge, The Verge, 9 January 2022. https://www.theverge.com/2022/1/9/22874949/developer-corrupts-open-source-libraries-projects-affected. Accessed 18 Jan 2022
2. Martin, B.: AccessPress Themes Hit with Targeted Supply Chain Attack, Sucuri, 20 January 2022. https://blog.sucuri.net/2022/01/accesspress-themes-hit-with-targeted-supply-chain-attack.html. Accessed 9 Feb 2022
3. Microsoft: Third-party notices|Microsoft Legal, Microsoft (2022). https://www.microsoft.com/en-us/legal/third-party-notices. Accessed 18 Jan 2022
4. Microsoft: "Open Source|Microsoft Legal," Microsoft (2022). https://www.microsoft.com/en-us/legal/intellectualproperty/open-source. Accessed 18 Jan 2022
5. An update from the Faker team, 14 January 2022. https://fakerjs.dev/update.html. Accessed 8 Mar 2022
6. Eckert, J.: The Ultimate WordPress Security Guide for 2022, Parachute Design, 14 January 2022. Accessed April 2022
7. Koskinen, T., Ihantola, P., Karavirta, P.: Quality of WordPress Plug-Ins: an overview of security and user ratings. In: 2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Conference on Social Computing (2012)
8. Souppaya, M., Scarfone, K., Dodson, D.: SP 800-218, Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities, February 2022. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218.pdf. Accessed 22 Feb 2022
9. NIST Joint Task Force: SP 800-53, Security and Privacy Controls for Information Systems and Organizations, September 2020. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf. Accessed 22 Feb 2022
10. Thompson, C., Wager, D.: A large-scale study of modern code review and security in open source projects. In: 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), New York (2017)
11. Davila, N., Nunes, I.: A systematic literature review and taxonomy of modern code review. J. Syst. Softw. **177**(0164–1212), 110951 (2021)
12. Ozment, A., Schechter, S.E.: Milk or wine: does software security improve with age? In: 15th USENIX Security Symposium (USENIX Security 06), Vancouver (2006)
13. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: 2005 Proceedings. 27th International Conference on Software Engineering, ICSE 2005 (2005)
14. Aljedaani, W., Javed, Y., Alenezi, M.: Open source systems bug reports: meta-analysis. In: Proceedings of the 2020 the 3rd International Conference on Big Data and Education (ICBDE 2020), New York (2020)
15. Adewumi, A., Misra, S., Omoregbe, N., Crawford, B., Soto, R.: A systematic literature review of open source software quality assessment models. Springerplus **5**(1), 1–13 (2016). https://doi.org/10.1186/s40064-016-3612-4
16. Adewumi, A., Misra, S., Omoregbe, N., Sanz, L.F.: FOSSES: framework for open-source software evaluation and selection. Softw. Pract. Experience **49**(5), 780–812 (2019)