WP6

DIGIT B1 - EP Pilot Project 645

Deliverable 3: General Reflection on the Experience of Performing the Code Reviews for European institutions

Specific contract n°226 under Framework Contract n° DI/07172 – ABCIII

October 2016

DIGIT Fossa WP6 – Governance and Quality of Software Code – Auditing of Free and Open Source Software.

Deliverable 3. General reflection on the Experience of Performing the Code Reviews for European institutions

Author:

everis
an NTT DATA Company

Disclaimer

Contents

List of Tables

List of Figures

ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
APR	Apache Portable Runtime
DG	Directorate General
EC	European Commission
EUI	European institutions
FOSS	Free and Open Source Software
FOSSA	Free and open Source Software Auditing
GUI	Graphic User Interface
IDE	Integrated Development Environment
SME	Subject Matter Expert
WP	Work Package

1 INTRODUCTION

1.1. Context

The security of the applications used nowadays has become a major concern for organisations, companies and citizens in general, as they are becoming a more common part of our daily lives, and are being used for business and leisure purposes alike. The information these applications manage has become an essential asset to protect, as it includes personal information, internal data, industrial property, etc.

From a security point of view, this new scenario presents many new challenges that need to be addressed in order to protect the integrity and confidentiality of the data managed by the applications and their users.

Furthermore, their exposure to the Internet has made them a prime target, due to the value that this private and internal information has.

One of the advantages of Free and Open-Source Software (FOSS) is that its source code is readily available for review by anyone, and therefore it virtually enables any user to check and provide new features and fixes, including security ones. Also, from a more professional point of view, it allows organisations to review the code completely and find the vulnerabilities or weaknesses that it presents, allowing for a refinement of their security and, in turn, a safer experience for all the users of the applications.

1.2. Objective

The objective of this document is to provide a reflection on the experience and knowledge gained during the Code Review process, providing a stepping ground for further improvements in the process and its continued evolution.

It covers all aspects and phases carried out during the execution of the code review by:

1. comparing the code review methodology developed in WP2 with the actual process that was executed;
2. analysing the points that could be optimised or improved in the form of Lessons Learnt;
3. providing recommendations so future upgrades can be added to the process.

1.3. Scope

This document covers the reflection of the code review experience, including all phases, tasks and activities that have been carried out during the code review process

1.4. Modification History

Date	Version	Author	Comments
21/10/2016	0.1	Juan Ortega Valiente Francisco de Borja González Carro Alberto Dominguez Magaly Estévez	Initial draft
25/10/2016	0.2	SP Landercy	75% completion quality review
27/10/2016	0.3	Juan Ortega Valiente Francisco de Borja González Carro Alberto Dominguez Magaly Estévez	Amendments after 75% quality review
02/11/2016	0.4	SP Landercy	100% completion quality review
02/11/2016	0.5	Juan Ortega Valiente Francisco de Borja González Carro Alberto Dominguez Magaly Estévez	Amendments after 100% quality review Final draft
04/11/2016	0.6	SP Landercy	Additional 100% completion quality review
07/11/2016	0.7	Juan Ortega Valiente Francisco de Borja González Carro Alberto Dominguez Magaly Estévez	Final version

1.5. Deliverables

WP2 - Deliverable 11: Design of the methods for performing the code reviews for the European institutions

2 SUMMARY OF THE EXPERIENCE PERFORMING THE CODE REVIEW

Conducting a code review entails gaining a deep understanding of the inner workings or the software application, library or segment analysed, understanding not only the functionalities it provides, but also the approach taken in order to make them available and even gain a little understanding of the authors' school of thought.

Information was collected during the code review process as the code reviewers encountered challenges, difficulties or different ways of doing the checks. All of this information was analysed, resulting in various sets of lessons learnt, and the respective recommendations developed that could further improve the code review process.

The following sections show in more detail the lessons learnt and recommendations for each of the sub-processes:

- Code Review Methodology
- Code Review Execution
- Code Review Results
- Code Review Effort Estimation

2.1. Code Review Methodology

The first set of lessons learnt was made regarding the methodology defined and used for the code review.

The following points indicate areas where improvement can be made:

Lesson Learnt 1	**General security controls are too generic, lack technical detail**. Generic controls need to apply to all languages (added or to be added in future release of the methodology), and should include specific details as to what to check in the code and how to identify issues.
Gap	○ lack of technical details in general controls
Quick win mitigation for code review process	○ around 10% of the general controls were updated to include more detail
Improvement	○ general controls need to be further refined, following an iterative process. This to ensure that they provide enough detail for code reviewers to check them, and that they are applicable regardless of the language in which the code is reviewed.

Lesson Learnt 2	**General security controls are mainly focused on interconnected applications, especially C and C++ applications for desktop[1] software. This is very useful for the code reviewed but lacks scalability if other types of applications are included in the review.**
Gap	○ security controls focused on interconnected desktop software
Quick win mitigation for code review process	○ N/A
Improvements	○ Upgrade the controls to cover a wider range of software, including mobile solutions, server applications, etc., as well as providing support for additional programming languages. ○ add templates for each software to increase scalability

[1] Applications installed directly on an OS in a computer, Workstation or Server.

Lesson Learnt 3	**Lack of controls covering sockets and their connections in the code review controls**
Gap	○ no controls covering socket implementations in general controls, nor in any language-specific controls
Quick win mitigation for code review process	○ N/A
Improvement	○ research possible controls to add for socket implementations, also covering different programming languages.

Lesson Learnt 4	**The number of controls that apply to a specific file is small compared to the total number of controls**
Gap	○ lack of efficiency when running large files on all potential controls.
Quick win mitigation for code review process	○ partial filtering performed by adding programming language tags to specific checks in general controls.
Improvement	○ further refine the list of controls and checks, adding tags, filters and references to quickly identify those that can apply and those that need to be overlooked.

Lesson Learnt 5	**The different methodology code review modes defined provide enough flexibility, but the steps defined are somewhat rigid**
Gap	○ code review modes are too rigid
Quick win mitigation for code review process	○ add flexibility to include mixed modes, including a manual mode aided by automated checks and/or focus on specific code sections.
Improvement	○ further analyse potential flexible modes to adapt in different applications to review, in line with time/resource restrictions.

2.2. Code Review Execution

The second set of lessons learnt was made during the actual code review process.
The following points indicate areas where improvement can be made:

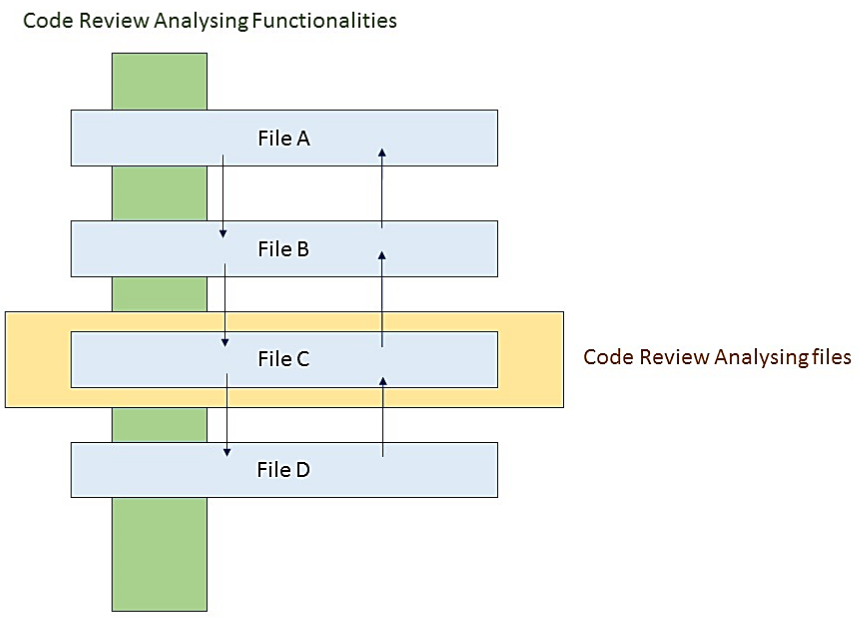
Lesson Learnt 6	**The defined document templates (text-based files) proved to be complex to use when registering the temporary results during the code review**
Gap	○ existing templates were adequate to present results but not for the collection of evidence during the code review.
Quick win mitigation for code review process	○ a spreadsheet template was generated to optimise the code review process, allowing reviewers to quickly review each control, provide the necessary result and evidence data, and later on provide a detailed assessment of the risk level of each finding. Code reviewers had to work in a separate spreadsheet, which was later merged into one for the final results.
Improvement	○ ideally, an application should be defined for this purpose, using a database to allow the concurrent work of multiple users (code reviewers), as well as providing an easy-to-use interface, help and guidelines and an easy way of providing an assessment. From the point of view of the report, this tool should also automate the process of generating and providing a final assessment, with corresponding graphs and indicators (such as CVE/CWE export data).

Lesson Learnt 7	**The code analysis in large applications proved to be very time-consuming, requiring a large number of resources and increased time allocation.**
Gap	○ the time needed for the manual code review stage was significant. See Section 2.4 for detailed information on the effort estimation.
Quick win mitigation for code review process	○ resource and queue optimisation to ensure the code review timing is respected. This was carried out by adding a procedure in the code review method.
Improvement	○ define an updated and optimised working queue that should allow the efficient processing of code reviews regardless of the software reviewed. Another solution contemplates the application of mixed modes, where a manual review is aided by automated tools

Lesson Learnt 8	**When analysing large applications, it was observed that it is necessary to modularise the code itself in order to allow multiple code reviewers to work in parallel**
Gap	○ need to modularise the code to process it
Quick win mitigation for code review process	○ the code was divided into modules (code libraries, etc.) and then in 'batches' (sets of files related to each other)
Improvement	○ need to define a structured process to modularise the code to be reviewed in large applications, allowing the possibility of multiple reviewers working in parallel and ensuring that the results are not negatively impacted, by avoiding overlaps .

Lesson Learnt 9	**An issue arose regarding the analysis of functionalities that use calls to functions in other files or libraries. Due to the segmentation of the code, and the limits of its scope, it is hard to analyse every single library added, such as those that are part of C, or external components of the software. Additionally, the code already analysed file by file should be examined following the hierarchy of calls, as shown in Figure 1: Code Review Analysing Functionalities.**
Gap	○ not all functions can be properly analysed to ensure that the calls made to 'third party' functions, or between files, are secure.
Quick win mitigation for code review process	○ calls made to functions within the code were double-checked whenever possible; external calls (OS; libraries not included in the scope, etc.) were not considered
Improvement	○ in future versions of the methodology, consider how to include the evaluation of the calls to that functions as well as the ones made to external libraries or OS functions, mainly in cases where specific versions are required (such as VS Runtime libraries).

Figure 1: Code Review Analysing Functionalities


Code Review Analysing Functionalities
File A
File B
File C
File D
Code Review Analysing files

2.3. Results of the Code Review

The last set of lessons learnt was made regarding the results of the code review

The following points indicate areas where improvement can be made:

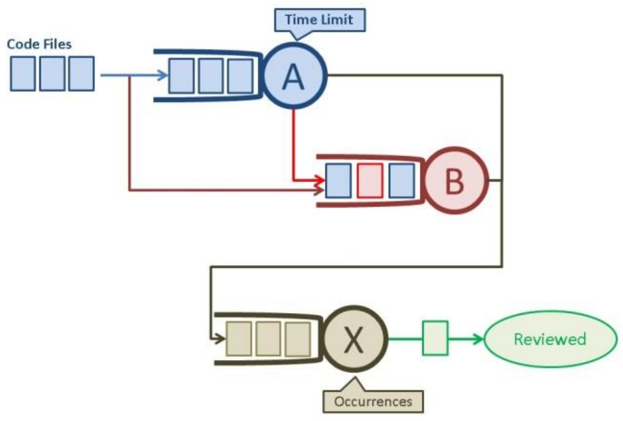
Lesson Learnt 10	**The evidence field has no template or indicator of the data it should contain, allowing each code reviewer to add the information that he/she considers appropriate**
Gap	○ lack of structure in the evidence field
Quick win mitigation for code review process	○ temporary templates and indications were given to the code reviewers to ensure that they provide enough evidence to validate the findings identified
Improvement	○ a structured format should be provided in order to ensure that the pieces of evidence are concrete, adequate, and to the point, unequivocally providing assurance that the findings are correct and proportionate.

Lesson Learnt 11	**It was observed that after the code review reports were given by the code reviewers, an additional Quality and Assurance (Q&A) stage had to be added to ensure that all results were adequate (findings position in the code, justification) and had the same format**
Gap	○ lack of standardised means of providing feedback.
Quick win mitigation for code review process	○ a Q&A stage was added to ensure that the results and evidence were correct and complete.
Improvement	○ ensure that the results and evidence are provided adequately with templates and guidelines. This also applies to the assessment of the findings (risk). If an application is defined, it would have to include specific templates and guidelines to ensure that code reviewers know exactly which information to provide and include in their reports.

Lesson Learnt 12	**The number of controls to include in the final report is excessive; for applications written in several languages this could mean 200+ controls**
Gap	○ need to optimise the final report to ensure that findings are properly identified and easily accessible
Quick win mitigation for code review process	○ controls that were not checked, or that passed successfully, were identified in the summary control table. However, their individual tables were not included
Improvement	○ define which control tables should appear on the final report; the consideration is to include only those that have any findings, or that have relevant evidence for the developers of the software reviewed

2.4. Lessons Learnt Resulting from the FOSS Communities Feedback

Feedback from the Apache HTTPD community resulted in the following areas of improvement:

Lesson Learnt 13	**The current code review workflow does not detect all occurrences of a finding**
Gap	○ The approach followed by the EU-FOSSA project aimed at reducing the code review time by sampling each finding, instead of listing every single occurrence in the code. To accomplish this, we prioritised the amount of code reviewed in a given period of time, instead of prioritising the number of occurrences of a finding. Since FOSS communities have expert knowledge in their software, this approach assumed that they would be the best candidates to detect all the occurrences in a limited sample. However, the FOSSA team did not take into account that FOSS communities do not necessarily have the required amount of resources for this purpose. It turned out that the Apache community is interested in all occurrences because their workflow is adapted for this kind of input.
Quick win mitigation for code review process	○ N/A
Improvement	○ The following modification of the code review procedure is proposed:  Time Limit Code Files A B X Reviewed Occurrences In the queue systems proposed in the procedure of the code review method (Annex 3 of *Deliverable 11: Design of the method for performing the code reviews for the European institutions*), a new queue (called ‘X’)

Lesson Learnt 13	**The current code review workflow does not detect all occurrences of a finding**
	is added, with the objective of finding all occurrences of a finding. While queues 'A' and 'B' work on detecting up to three occurrences of a finding, the queue 'X' will work in parallel conducting the following activities: • First Q&A review in order to discard false positives. • If it is not a false positive, find all occurrences of a finding in a code file. It is assumed that if a finding appears in several files, queues 'A' and 'B' will identify the code files affected, and queue 'X' will detect all occurrences of a finding in a code file. Following this new approach, the results will gather all occurrences of a finding with low impact in time (due to the queues working in parallel). However, the efforts and cost of the code review project will increase, as more code reviewers will be needed to work in queue 'X'.

Lesson Learnt 14	**The "FOSS version" of the code review report that is sent to the FOSS Communities should directly address the findings.**
Gap	○ the current report template includes information that might not be relevant to FOSS communities, as their interested is focused mainly on the findings. ○ Currently, the findings are added by controls (following the approach explained in lesson learnt 13) and presented in this way in the report.
Quick win mitigation for code review process	○ N/A
Improvement	○ Section 1 – Introduction should be eliminated ○ Section 3 – Methodology should be eliminated ○ To improve the presentation of the findings and expand on the information provided, a table should be included with the following information: the finding, a description, the associated failed control, and the location (file and line of code) of each occurrence. ○ To ensure the document integrity, each page should have a watermark, indicating that it is a draft, to be removed once the document is approved the FOSS community. This is a countermeasure to prevent information leakages before the FOSS community final approval, and to avoid misunderstandings about the results of the code review.

Lesson Learnt 15	**Find alternative methods to improve the communication with the FOSS communities**
Gap	○ The documentation should not be sent to email lists, as it is likely to be sent to thousands of people around the globe, causing misunderstandings and inconveniences. An email list might create confusion about the results, instead of being an efficient mechanism to interact with the FOSS communities.
Quick win mitigation for code review process	○ N/A
Improvement	○ European institutions should have a contact list for code review projects with several key points of contact (POCs) from the FOSS communities and the project owners. ○ These POCs should be engaged in the discussions about the code review results. ○ The list should be updated regularly, and the key contacts from FOSS communities should be willing to act as a contact point with the European institutions and be aware of the code review activities being executed by the European institutions on their software.

Lesson Learnt 16	**The selection of the project for code review should be based on the community size, as larger communities may already have their own code review processes and more resources to commit to this activity.**
Gap	○ The two code reviews conducted were considerably different in terms of lines of code, size of the community and communication method with the community to discuss the results. This could lead to the following: • A more challenging communication process with larger communities, as they may have more than one point of contact; • Longer discussion process to agree on the findings and the resolution process; • Long reports and not enough resources to attend them.
Quick win mitigation for code review process	○ N/A
Improvement	○ Select a project that does not have all the resources to conduct a security code review. ○ Prioritise smaller communities that are strategic for the European institutions. ○ If a larger community is selected, agree beforehand on the scope of the code review and the point of contact responsible from the community side. ○ Involve the community in the planning process, as per the Code Review methodology developed.

2.5. Effort Estimation to Conduct a Code Review

In order to be able to estimate the effort required to conduct a code review, the following activities were conducted

1. Analysis of the FOSSA project data when reviewing the code of Apache (C) and KeePass (C++):

LoC (Lines of Code)	1. around 61 000 LoC analysed in C 2. around 84 000 LoC analysed in C++
Code review team	1. three code reviewers analysing C code 2. four code reviewers analysing C++ code
Code review timeline	Four weeks
Number of controls	1. 160 code controls applied to C-based software: ○ 68 controls corresponding to the managed and defined modes (about 42.5 % of the total of controls) ○ 92 controls corresponding to the optimised mode (The remaining 57.5%). 2. 218 code controls applied to C++-based software: ○ 68 controls corresponding to the managed and defined modes (about 31% of the total of controls) ○ 150 controls corresponding to the optimised mode (the remaining 69%).

2. The above data resulted in
   - 145 000 LoC analysed
   - Team of 28 reviewers/week (4 reviewers x 4 weeks and 3 reviewers x 4 weeks).
   - 129.5 lines of code per reviewer and hour. This is the result of the following calculation:

145 000 lines of code / (28 reviewers/week x 5 days/week x 8 hours/day) =

129.5 lines of code/reviewer/hour

- The above result is calculated using the optimised mode (which by default includes the managed and defined modes)[2]

3. Managed and defined modes: a total of 136 controls analysed (68 applied to Apache and 68 applied to KeePass).
4. For the optimised mode, a total of 242 controls (92 applied to Apache and 150 applied to KeePass) were analysed.
3. Taking the above data into account:
   - around 36% of the analysed controls were from managed and defined modes
   - around 64 % of the controls were examined from the optimised mode.
   - This implies that, for each hour spent conducting a code review:
     i. One third of the time (22 minutes) was spent analysing controls from the managed and defined modes
     ii. Two thirds of the time (38 minutes) was spent analysing controls from the optimised mode.
4. Results:
   - Apache code review: eight failed controls (one from managed and defined modes, and seven from optimised controls). As a result, 12.5% of the failed controls come from the managed and defined modes, while the remaining 87.5% come from the optimised mode.
   - KeePass code review: 14 failed controls (four from managed and defined modes, and ten from optimised controls. As a result, 29% of the failed controls come from the managed and defined modes, while the remaining 71% come from the optimised mode.

To conclude, the FOSSA pilot project resulted in a total of five failed controls from the managed and defined modes in FOSSA project (about 23%), and 17 from the optimised mode (about 77%), **highlighting the importance of the optimised or manual code review.**

As final remarks, it is important to take into account:

1. The number of lines of code/reviewer/hour depends on the nature of the software, its type (web application, mobile application, etc.) and the programming language used in the software. Analysing a web application will most likely produce different statistics from a workstation software.

[2] See "Annex 2: Code Review Methodology –Code Review modes" for explanation of the code review modes.

2. The effort estimation is only for conducting the code review on the specified software, and as such does not include the Quality assessment, analysis and report writing tasks, which can take more time than the code review itself.
3. However, the effort estimation of the FOSSA code review can be used to plan future code review projects.

3 CONCLUSIONS AND FUTURE ACTIONS

To conclude, the results of the EU-FOSSA pilot project were satisfactory, with areas of improvement identified, and a set of future actions proposed for the European institutions to ensure the continuation of this initiative.

The future actions have been grouped in three sets, to target specific areas or groups, and to optimise the management of the Action Plans that should be created to ensure that the security of the critical software is continuously improved

3.1. Code Review Method

The current code review method can be improved to maximise the benefits using the knowledge gathered during the FOSSA project. The code review method is the essential element of the code review projects, where it impacts the project costs and duration, as well as the potential results.

1. Code review is a time-consuming activity that needs to be optimised to be feasible regarding costs and duration. Taking that into account, the FOSSA team has proposed a procedure where different working queues correspond to different code review levels. These levels have different time limits and knowledge to find security flaws in the code, as explained in *Deliverable 11: Design of the methods for performing the code reviews for the European institutions*. Therefore, the procedure of the code review should be refined according to the type of software analysed, to improve its efficiency.
2. The FOSSA team realised that it was required to perform a Q&A of the findings and their assessment, to integrate and normalise the information of the different code reviewers. This harmonises the results to make them more coherent. The process can be considered iterative, and during its development the methodology is continuously improved and updated, to include newer features in the languages supported, as well as adding support for other programming languages, systems and environments (e.g. mobile applications, web applications, etc.).
3. It is advisable for future actions to analyse the rest of Apache HTTP Server, as well as the rest of the libraries that it uses. This software is widely used as a web server and proxy server, and it should be analysed according to that fact.
4. The development or acquisition of an automated software tool to assist the code review process would be quite advisable to improve the efficiency of the code review process, as well as its feasibility. Possible features can be such as integration with automatic tools, centralised assessment and Q&A, etc.

3.2. Security Framework for Free and Open Source Software

Overall, it can be considered that the Code Review is just a phase in a bigger Security Framework concept, which includes a dynamic analysis, penetration testing, etc. This would provide the ideal revision of the security of an application, providing developers with the knowledge needed to ensure that their software is up-to-date in terms of security and allowing users to feel a bit more secure when using these solutions.

5. A single type of security audit (code review, penetration testing, etc.) will not detect all possible security issues in any software. This is why it is recommended to use several security tests to improve the security of the software;
6. Taking advantage of the knowledge acquired during the code review, security development guides and study material can be produced for software developers in order to improve the security of the European systems and applications;
7. Open wikis by security experts may be created to exchange information and best practices in software development, following the same philosophy as FOSS communities.

3.3. European institutions and FOSS Communities

Some FOSS software is a key part of the IT infrastructure of many European organisations, and such software provides a key value for the European society. Because of this, it is of utmost importance to develop mechanisms for collaboration between the European institutions and FOSS communities.

As other elements are part of the infrastructure needed for the member states, such as motorways and rail networks, some FOSS software might be considered as IT infrastructure. This means that the European public administration should promote and support FOSS software.

European institutions could collaborate with FOSS communities in the following ways:

8. Conduct security tests (Penetration testing, code analysis, etc.) on the FOSS, providing feedback and recommendations to FOSS communities;
9. Expand the scope of the security tests to include the solution of security issues detected during those tests;
10. Collaborate in the software development, either in an official way or allowing the developers of the European institutions to contribute to FOSS communities (sharing the code developed in the EUI, allocating time slots so developers can contribute to the FOSS communities development tasks, etc.).
11. Create documentation and guides for FOSS communities to improve software security;
12. Create forums or wikis to generate knowledge about software security, where experts can provide information;

13. Create webinars for developer about secure coding (per language), or about security in essential technologies used in software development (HTTP, etc.);
14. Develop subject matter experts (SMEs) in the areas of secure software and security in the software lifecycle, etc;
15. Promote the sponsorship of FOSS software regarded as IT infrastructure that is in a bad situation (at risk of being discontinued). However, this software at risk can be critical and used in many other projects (FOSS or proprietary software);
16. Promote the use of FOSS software within the European institutions by increasing its usage, by contributing to the development of FOSS software or by helping with the dissemination of that software.

4 ANNEXES

4.1. Annex 1: Future Actions

The Free and Open Source Software Audits are an important contribution to the European institutions open source strategy, and to ensure the reliability and security of the IT infrastructure we all rely on.

To ensure that this pilot project becomes an ongoing activity, a series of future actions are proposed, that should be analysed by the European institutions to ensure the continuation of this critical process.

Table 1 depicts in detail the future actions, grouped in three areas.

Table 1: EU-FOSSA pilot project - Future Actions

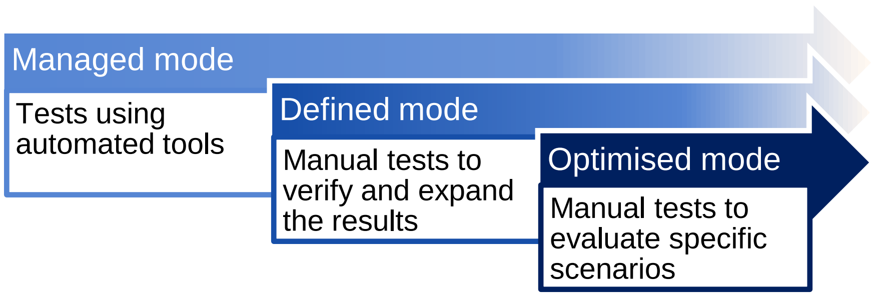
Area	Future Action
Code Review Method	1. The FOSSA team has proposed a process where different working queues correspond to different code review levels. These levels have different time limits and code reviewer know-how to find security flaws in the code, as explained in *Deliverable 11: Design of the methods for performing the code reviews for the European institutions*
	2. Perform a Q&A of the findings and their assessment, to integrate and normalise the information of the different code reviewers.
	3. Analyse the rest of Apache HTTP Server, as well as the rest of the libraries that it uses.
	4. Develop or acquire an automated software tool to assist in the code review process to improve its efficiency and feasibility.
Security Framework for Free and Open Source Software	5. Conduct different types of security audits, like code reviews, penetration testing, etc., to improve the security of the software.
	6. Create security development guides and study material for software developers in order to improve the security of the European systems and applications.
	7. Create, open wikis of security experts to exchange information and best practices in software development, following the same philosophy as the FOSS communities.

Area	Future Action
European institutions and FOSS Communities	8. Conduct security tests (Penetration testing, code analysis, etc.) on the FOSS, providing feedback and recommendations to FOSS communities.
	9. Expand the scope of the security test to include the solution of the security issues detected during those tests.
	10. Collaborate in the FOSS development, either in an official way or by allowing the developers of the European institutions to contribute to FOSS communities. This can be accomplished in several ways, such as: a. sharing the code developed in the EUI b. allocating time slots so developers can contribute to the FOSS communities development tasks.
	11. Create documentation and guides for FOSS communities to improve software security
	12. Create forums or wikis to generate knowledge about software security, where experts can provide information
	13. Develop subject matter experts (SMEs) in the areas of secure software and security in the software lifecycle, etc.
	14. Create webinars for developers about secure coding (per languages), or about security in essential technologies used in software development (HTTP, etc.)
	15. Promote the sponsorship of FOSS software regarded as IT infrastructure that is in a bad situation (at risk of being discontinued). However, this software at risk can be critical and used in many other projects (FOSS or proprietary software).
	16. Promote the use of FOSS software within the European institutions by increasing its usage, by contributing to the development of FOSS software or by helping with the dissemination of that software.

4.2. Annex 2: Code Review Methodology –Code Review modes

The execution process was divided into three sequential phases, each providing data as input for the next one, as depicted in Figure 2. All of them were carried out by the code review team, using both automated and manual tools.

Figure 2: Code review execution order


Managed mode
Tests using automated tools
Defined mode
Manual tests to verify and expand the results
Optimised mode
Manual tests to evaluate specific scenarios

Each phase corresponds to a code review mode, as explained below:

- **Managed mode**: covers the execution of the automated tools selected for the analysis of the code. The following categories were analysed:
  - *Data/Input Management (DIM)*: The data entry points of an application, service or library are one of the weak points that must be controlled against unexpected values. The subcategories covered are as follows:
    - File Input / Output Management (FIM)
    - Data stream management (DSM)
    - Character encoding management (CEM)
    - Input validation and sanitisation (IVS)
    - Sensitive Data Management (SDM)
    - Entry point validation (EPV)
    - XML schema validation (XSV)
  - *Authentication Controls (AUT)*: It covers any aspect related to the process during which the solution reviews and verifies the identity of another entity, such as a user. The subcategories covered are as follows:
    - Authentication verification (AUV)
    - Password policy usage (PPU)
    - Credential storage security (CST)
    - User account protection (UAP)
    - Password recovery process (PRP)

- *Session Management (SMG)*: It covers all parts of the protection and management of user sessions once they are authenticated against the solution. The subcategories covered are as follows:
  - Session creation (SCP)
  - Session ID management (SID)
  - Session lifecycle (SLC)
  - Session logout (LGP)
- *Authorisation Management (ATS)*: This process is designed to ensure that when a user or entity correctly authenticates against the application, s/he gets the proper privileges assigned to it. The subcategories covered are as follows:
  - Access control system (ACS)
  - Privilege revision (PRV)
- *Cryptography (CPT)*: Covers all aspects related to the protection via encryption of the information and data in transit and at rest. The subcategories covered are as follows:
  - Credential protection at rest (CPR)
  - Cryptographic configuration (CRC)
- *Error Handling/Information Leakage (EHI)*: The information provided by the application errors, page metadata and sample content must be filtered to avoid a leakage of sensitive information. The subcategories covered are as follows:
  - Information leakage (INL)
  - Sample files (SFL)
  - Error handling (EHD)
- *Software communications (COM)*: it comprises those functions that manage and control network connections, including sockets and protocol functions. The subcategories covered are as follows:
  - HTTP Secure Management (HSM)
- *Logging/Auditing (LOG)*: The logs generated by an application are a superb source of information about its contents, workings and potential weaknesses. The subcategories covered are as follows:
  - Log configuration management (CFG)
  - Log generation (GEN)
  - Log sensitive information (LSI)
- *Secure Code Design*: There are several aspects related to the application itself and the technologies and frameworks used for its implementation. The subcategories are as follows:

    - Framework requirements (FWK)
    - Variable types / operations (VTY)
    - Expressions/Methods (EXM)

- **Defined Mode**: once the managed mode is finished, the code review team will have a set of results generated from the automated tools. These results, together with the manual tests needed, are checked in order to fill the controls and checks that will provide the final results.

- **Optimised Mode**: focuses on those sections of the application that are found to be most at risk, alongside several more specific tests that require further evaluation. They are divided into the following subcategories:
    - Concurrency (CCR)
    - Denial of Service (DOS)
    - Memory and resource management (MRM)
    - Code Structure (COS)
    - Role-privilege matrix (RPM)

  The optimised mode covers the set of language-specific (C, C++, JAVA and PHP) controls, and other controls related to the code unique particularities. The language specific controls for C (CBC) are divided into the following subcategories:

    - Pre-processor (PRE)
    - Variable Management (VMG)
    - Memory Management (MEM)
    - File I/O Management (FIO)
    - Environment (ENV)
    - Signal and Error Handling (SEH)
    - Concurrency (CON)
    - Miscellaneous (MSC)