A Practical Experience Applying Security Audit Techniques in an Industrial e-Health System Which Uses an Open Source ERP

Julián Gómez[a], Miguel Á. Olivero[b], J. A. García-García[c] and María J. Escalona[d]
Web Engineering and Early Testing (IWT2, Ingeniería Web y Testing Temprano), University of Seville, Spain

Keywords: Audit, Cybersecurity, Odoo, Healthcare, Pentest, Pentesting, Security.

Abstract: Healthcare institutions is an ever-innovative field, in which modernization is moving forward taking giant steps. This modernization, so called "digitization", brings up some concerns that should be carefully considered. Currently, the most sensible concerning in this field is the management of Electronic Health Record and patients' data privacy. Health-related data in healthcare systems are under strict regulations, such as the EU's General Data Protection Regulation (GDPR), whose non-compliance imposes huge penalties and fines. Cybersecurity in healthcare plays an important role at protecting these sensitive data, which are highly valuable for criminals. Security experts follow already existing security frameworks to orchestrate the security assessment process, so that the auditing process is as complete and as organized as possible. This study extends the lifecycle of a security assessment framework and conducts an exploitation and vulnerabilities' analysis on an actual industrial scenario. The results of this security audit shows that even if the system is heavily fortified, there can be still some vulnerabilities.

1 INTRODUCTION

Cybersecurity is a field that is located within computer science, but is related to other disciplines such as law, legislation, and security and control forces. According to ISO/IEC 27032:2012 (ISO, 2012), cybersecurity is "the preservation of confidentiality, integrity and availability of information in cyberspace. In addition, other properties such as authenticity, accountability, non-repudiation, and reliability may also be involved.". In the end, cybersecurity is in charge of defending the information of computer systems and electronic communication.

Personal data is always sensitive information, but even more so in the case of healthcare systems. In order to manage and successfully conduct audits, a set of steps or phases should be followed. It is with a structured set of step that the security audit is complete and thorough. Such steps or phases are established through the security frameworks.

However, a gap has been identified. The current frameworks have a set of structured phases that aim at ensuring that the security audit is as complete as possible. But they lack on analysis on source code. This study extends the auditing process by including an additional stage consisting on performing analysis to the source code of a system. The inclusion of this phase leads to the strengthen of the auditing process. Besides the additional phase, this study also shows how a security audit of an actual Electronic Health Record (EHR)-related system has been performed through an industrial case scenario.

After following a set of phases established by the frameworks, a new phase consisting in applying techniques of static analysis is proposed. This new phase adds a security layer in the process of auditing a system, needed in critical systems such as the ones in healthcare.

Since the audited system is a live system of a industrial healthcare institution, its name shall not be disclosed due to privacy reasons. The remaining of this paper is organized as follows: Section 2 and 3 introduces the background and summarizes some related work. Section 4, presents the execution of the security audit and its results. This section has been organized in 5 subsections, each one corresponding to one of the phases given by the audit frameworks. Finally, the last section states a set of conclusions and suggests some research lines that as future work.

[a] https://orcid.org/0000-0002-3157-1469
[b] https://orcid.org/0000-0002-6627-3699
[c] https://orcid.org/0000-0003-2680-1327
[d] https://orcid.org/0000-0002-6435-1497

Gómez, J., Olivero, M., García-García, J. and Escalona, M.
A Practical Experience Applying Security Audit Techniques in an Industrial e-Health System Which Uses an Open Source ERP.
DOI: 10.5220/0010714500003058
In *Proceedings of the 17th International Conference on Web Information Systems and Technologies (WEBIST 2021)*, pages 482-489
ISBN: 978-989-758-536-4; ISSN: 2184-3252

2 BACKGROUND

Every year, ISACA publishes a summary of the state of the art, i.e. the state of the field and the latest innovations in cybersecurity. The report can be found on the official ISACA website (ISACA, 2021).

The 2021 report gets the data after doing a macro survey in the last quarter of 2020 to professionals in the field, of different nationalities, ages and with different years of experience. Demand is on the rise and more and more professionals are needed. More than 60% of respondents describe the situation in their company's cybersecurity department as "significantly understaffed". The report points out that business understood from the traditional point of view does not work well. The cybersecurity workforce is scarce and, in the future, will probably continue to be scarce, because it takes a long time to train a professional from a theoretical point of view.

To conclude, ISACA expects that 2021 will be the year in which companies will start hiring as many professionals as there are vacancies to fill. In addition, it is suggested not to overestimate the effect of "digitized classes" in education, since the skills that are most lacking are the interpersonal ones: the soft-skills.

3 SECURITY AUDIT FRAMEWORK

This section will look at how the security audit itself has been managed. The first step to take when choosing a methodology is to know which ones there are and what each of them consists of in brief. The five most popular ones are the following (Gkoutzamanis, 2020): OSSTMM, OWASP Web Security Testing Guide, NIST SP 800-115, PTES and ISSAF.

3.1 Standardized Frameworks

Regarding the security audit framework itself, OSSTMM and OWASP Web Security Testing Guide have been chosen. These two frameworks are *de facto* standards in the field of cybersecurity. Unlike PTES or ISSAF, the first ones on the list are very popular and are still being updated today. In the case of OSSTMM, it is particularly effective, because within the existing general and broad methodologies, it is the most comprehensive of all, considering aspects that other guides do not consider, such as the human aspect. Regarding the OWASP Web Security Testing Guide, it is a very popular guide to perform security audits on web applications. Since the system that will be audited is a web system, this guide is ideal. For all these reasons, these are the two frameworks that the study will be working with.

In addition to the phases, there are different types of audits. Each of the phases will fall into one of the three kinds here explained:

- **Black Box.** This audit is the process that would be followed by someone totally external to the system, who has no prior ideas about how the system is developed from the inside. This type of audit simulates the state that a cybercriminal who is going to attack a system would start from and has to gradually gather information.
- **White Box.** This type of audit is the complete opposite of the black box audit, as the auditor now has all the information on how the system is developed inside and can see the source code. However, he has no prior knowledge of what attack vectors he is going to test yet, nor what vulnerabilities there may be.
- **Grey Box.** This type of audit relies on the auditor having partial knowledge of the details of the system. In fact, it is the type of audit that most auditors start from: they know what type of system they are going to audit in broad strokes. For example, in the case of the system of this project, the only previous information before the security audit was that the system uses an Odoo 11 web system.

These three types of auditing are not mutually exclusive and are actually performed all at once. Generally, a passive scan of the application without knowing anything (black box) is what goes first and then goes the specific details (the source code, configuration files, platform where the application is deployed...).

After looking at the types of audit, it will be explained which are the phases that will be developed during the security audit. These phases have been established after combining the OSSTMM framework and the OWASP framework. In addition to that combination, a new phase is proposed. Such phase receives the name of "Static analysis" and it aims at analyzing the source code of the system in order to discover new vulnerabilities.

The visual representation of the flow of a security audit can be seen on Figure 1, being "Static analysis" the highlighted in blue. The phases are:

1. **Scope.** Definition of the scope and objectives of the audit. See section 4.1.
2. **Social Engineering.** Social engineering techniques will be used to try to breach the system

starting from the weakest link in the chain: the users. In this paper, this phase will not be discussed due to length restrictions. See section 5.

3. **Vulnerability Analysis.** The vulnerabilities will be analyzed if they are corrected. In addition, it will be checked if the system is vulnerable to the vulnerabilities published in the OWASP Top 10. See section 4.2.
4. **Vulnerability Exploitation.** Some of the vulnerabilities found will be exploited. See section 4.3
5. **Static Analysis**. The source code will be passed through code analyzers to find memory bugs and bad *input sanitization*. This is the phase proposed by this study. See section 4.4.
6. **Writing the Report.** An executive report that contains all of the knowledge that has been generated. See section 4.5.

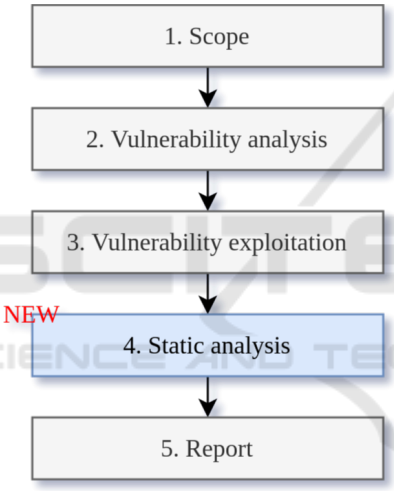

Figure 1: Phases of the security audit.

3.2 Phase Addition Proposal

Static code analysis is done on the code without executing it. The main difference from the programs used in the other sections (4.2), is that with those programs the analysis was performed on the live application (so-called dynamic analysis), while for this static analysis only the source code is needed.

Static analysis makes it possible to detect vulnerabilities that otherwise would not be easily detected because the functionality will not be executed and will need a trigger or because a piece of code is not exposed to users.

This static analysis of code is not usually done in audits that follow step by step the frameworks, perhaps because they are more associated with being run on the Continuous Integration tool pipeline when code is uploaded to a repository. However, the power and utility that these analyses provide should not be underestimated, because on many occasions security analysis programs miss vulnerabilities that without analyzing the code would be impossible to detect.

It is assumed that a cybercriminal would not have access to the source code of an application in any case, but that would be relying too much on the fact that such access is impossible to obtain. In fact, cybercriminals can make use of social engineering techniques in order to gain access to source code. It is this distrust that leads us to improve the source code of our system, even when it is difficult - not impossible - for a cybercriminal to gain access.

4 EXECUTION OF THE SECURITY AUDIT

Each of the proposed phases of the security audit will be covered in this section, except the social engineering phase, due to limitations in length.

4.1 Scope

This phase is of the kind “Grey box”.

The first phase of a security audit is to define the scope. *Scope* refers to exactly what parts of the system are to be audited and under what conditions, not to be confused with the scope of the paper itself, which is the performance of a security audit and the proposal of a new phase in the auditing process.

To know what alternatives there are to audit, the best option is to consult checklists to provide a scope as complete as possible. One such checklist is the OWASP ASVS checklist (OWASP, 2019). It provides steps to follow to investigate a system in depth. Although this list is already quite comprehensive, it is good practice to consult other lists to make the scope even more complete. Briefly summarizing,it is as follows:

- Process abuse and social engineering.
- Application architecture identification.
- Evaluate which parts of the system are critical.
- Authentication.
- Authorization.
- Validation of inputs.
- Bugs in application logic.
- Web Server and Framework.
- Context-Dependent Encounters.

Once the alternatives that exist to audit are known, a meeting with the client to define the scope needs to take place. Everything that was going to be audited was also discussed. The following list reflects what is included in the scope:

- This project's entire system: Login page, Internal page, Custom modules, Known vulnerabilities of Odoo 11.
- Basis security protocols analysis such as SSL and HTTPS.
- If the following can be obtained or modified: Configuration of any kind, Directory browsing, .htaccess

4.2 Vulnerability Analysis

This phase is of the kind "Black box".

Vulnerability analysis is one of the most important phases to be performed in a security audit. These vulnerabilities are present in information systems of all kinds: bugs in application logic, user inputs that are not properly escaped or sanitized, redirects.... The list is long. This section explores how vulnerability scanning can be done, what tools and techniques are available, and what it is for.

To make vulnerability scanning as complete as possible, there are numerous guides with lists of vulnerabilities to check, both manually and with an automated tool. One such guide is the OWASP top 10 (OWASP, 2017). However, if a more exhaustive list is needed, the OWASP ASVS (OWASP, 2019) can be checked. The OWASP Top 10 has the following as a summary:

1. Injection: Injection flaws, such as SQL, NoSQL, OS, and LDAP injection.
2. Broken Authentication: Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens.
3. Sensitive Data Exposure: Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII.
4. XML External Entities (XXE): Many older or poorly configured XML processors evaluate external entity references within XML documents.
5. Broken Access Control: Restrictions on what authenticated users are allowed to do are often not properly enforced.
6. Security Misconfiguration: Misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations and verbose error messages containing sensitive information.
7. Cross-Site Scripting XSS: XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping.
8. Insecure Deserialization: Insecure deserialization often leads to remote code execution.
9. Using Components with Known Vulnerabilities: Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application.
10. Insufficient Logging & Monitoring: Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems and maintain persistence

As with other areas of software engineering, the discovery of these vulnerabilities is not a complete process. When an application is in the testing phase and there is no warning in the tool used to test the application, it does not mean that it is known with certainty that the application will not have flaws, but that the tool does not find more and that what has been tested does not give flaws. It does not mean that there cannot be others. Usually, these tools use an *oracle*, which evaluates based on parameters that are specified if there are bugs in a part of the code or not. This is why it is said that the *testing* phase is not complete. All the bugs in an application cannot be discovered and it will never be certain that a piece of software is free of bugs.

The same is true for security auditing. When using an auditing application or even auditing manually without the help of tools, certainty that the system will not have more vulnerabilities is not a given. The program will test for known vulnerabilities and will try to be as complete as possible, but there may be vulnerabilities that it cannot discover, for example, because they are of the type **zero day**, vulnerabilities that have not been published and that the person who knows about the vulnerability can exploit. This is why it is so difficult to determine whether an audit is complete or not. The only way to do so is to trust that the security auditor, with the experience and knowledge he has, will report as many vulnerabilities as he finds and will always try to do as complete an audit job as possible.

Knowing that a tool cannot discover all of the vulnerabilities of a system, the following sections are presented.

Vulnerability scanning tools compete in a niche market that is hotly contested. There are few free programs, and those that do offer free versions lack many of the features that are really needed. These licenses can cost as much as 3600 euros per year, as is the case with Nessus, or as affordable as 360 euros per year, as is the case of Burp suite. In fact, the best strategy is to combine various tools as they often have different ways of reporting vulnerabilities.

After comparing Nesus, SolarWinds MSP, OWASP ZAP, Burp Suite and Rapid7 InsightVM, two of the most widespread tools in the cybersecurity field have been used when scanning for vulnerabilities in a system: OWASP ZAP and Burp Suite. These two programs are dynamic scanners, i.e. they analyze the running system and take it as if it were a black box without seeing its source code, which automate the vulnerability discovery process.

4.2.1 OWASP ZAP

The configuration of the tools is not a trivial process, because to give an example, the application to be audited has a login system whose access has to be described in the tool.

The tool is configured by adding to the "scope" the URL to be audited. With respect to this login, the tool needs an anti CSRF token, which briefly, is a token that prevents a user from executing in their web browser malicious content from the web page where they are authenticated.

The tool has two scanners. The passive scanner makes no changes to the web application, i.e. it only makes GET requests. The passive scanner is also called *spider* and collects information from the application bit by bit. There is another type of scan: the active scan. This scan makes changes to the application, i.e. it will be able to execute more HTTP commands besides GET, such as POST, PUT, DELETE.... It is a much more aggressive scan, since it will try to exploit as many security flaws as it finds. It should not be run on production systems as it will try to crash the system. In fact, during the execution of this type of scan, the URL that was provided to me with the copy of the production system crashed due to the high number of requests. If this were to happen on a production system, it could be catastrophic.

The tool, after being run for a couple of hours, finds as many vulnerabilities as it can and then generates a report with all vulnerabilities, their severity and their solution. Perhaps this report is the most important part of running the tool because it is what the development team uses to fix the vulnerabilities.

4.2.2 Burp Suite

The configuration of Burp suite is significantly simpler than that of ZAP. For login, Burp suite opens a browser window and copies the clicks and keyboard inputs and then plays them back. There is only one type of scan that takes out all the vulnerabilities it finds. However, it does not have the HUD, so the security audit is not as interactive.

As with ZAP, Burp Suite generates a report with all the vulnerabilities it finds. Example of this report can be found in (Burp Suite, 2019).

4.3 Vulnerability Exploitation

This phase is of the kind "Black box".

This section aims to further investigate vulnerabilities that have been discovered in the previous sections, as well as to consider as false positives some vulnerabilities that in fact are not. No new tools have been used to investigate vulnerabilities, except those already used (OWASP ZAP and Burp suite), because all tests have been done manually to eliminate false positives.

The analysis tools give the severity and confidence of the vulnerability. Their severity indicates how high a priority they should be given to fix, and the confidence indicates how certain the vulnerability is present or if it is a false positive. Both tools produce tables that indicate the severity of the vulnerability, and in the case of Burp suite, it also indicates how confident Burp suite is regarding a false positive.

		Confidence			
		Certain	Firm	Tentative	Total
Severity	High	1	1	9	11
	Medium	0	4	0	4
	Low	16	6	1	23
	Information	24	8	4	36

Figure 2: Table generated by Burp Suite.

The OWASP ZAP report has reported 16 alerts, with a total of 395 instances. Burp suite has reported 74 vulnerabilities and they are represented in the figure 2. As it can be seen, this table is insightful for the auditor since it gives the auditor what to audit first: those vulnerabilities with higher risk and confidence.

In total there are more than 450 vulnerabilities to be investigated. Normally in an enterprise environment an estimate of how long it would take to investigate all vulnerabilities is usually given based on the number found. Due to limitations in length of the presented work, two of the most critical vulnerabilities

reported by OWASP ZAP will be further investigated and other two by Burp suite.

4.3.1 Vulnerability 1: Redirects

This first vulnerability is present in two forms: as *External redirect* and as *XSS Reflected*. They have high severity and medium confidence. If we look at the url, the common path of the two vulnerabilities is.

```
https://o****n.com/web/session/logout?redirect=
```

This path itself is very dangerous because it is vulnerable to an XSS and redirect attack. An XSS attack is an attack that relies on an attacker being able to inject Javascript code into a web page. In this case, an attacker could inject code to steal users' cookie information, for example.

If the URL is further investigated, the logout page shows a new aspect. There are, in fact, two hidden fields: one with an anti CSRF token and another with the name *redirect*, which corresponds to the *redirect* of the URL marked as vulnerable. It is therefore concluded that the login page also has this vulnerability. If an attacker wanted to exploit it, he would only have to put in a URL a redirect parameter as follows:

```
https://o***n.com/web/login?redirect=https://
github.com/
```

and a URL to which you would like to redirect. This way, after the users enter their data, they will be redirected to *github.com* (in this example).

To fix this vulnerability, it is recommended that the development team try to do the *login* and *logout* redirects without relying on hidden fields in forms, but instead do it with session *cookies* or in an internal way.

4.3.2 Vulnerability 2: DOM XSS

This vulnerability actually has nine instances. They have high severity and tentative confidence. The exploitability of these vulnerabilities, if the notes of the report is looked at, depends on jQuery libraries. These two vulnerabilities are, in effect, the same and occur because of the multiple vulnerabilities that are present in jQuery version 1.11.1. It is recommended that the company upgrades to the latest version of jQuery and always keeps its dependencies up to date.

4.3.3 Vulnerability 3: SQL Injection

Path traversals (also called Directory traversals) are vulnerabilities that allow attackers to break out of the *var/www* folder where the web is hosted on the server and be able to query and modify files on the server. This vulnerability has been reported by OWASP ZAP with eight instances that have high severity and low confidence.

There is a total of eight declared instances, which have different URLs. However, in all URLs the parameter that could be changed to an escape path is the last one. For example, in the first instance, it is:

```
https://o***n.com/web/dataset/call_kw
/mail.message/load_views
```

This should change *load_views* to an escape path. To exploit this vulnerability, escape paths of the kind *../../../etc/passwd* and with encoding such as *..%252f..%252f..%252fetc/passwd* were used.

If the server is vulnerable it should return the file *etc/passwd*. However, after testing the escape paths listed in the snippet above, no way to escape to parent folders containing other files has been found, so it is declared as a false positive.

4.4 Static Analysis

This phase is of the kind "White box". With this phase, it is expected to discover new vulnerabilities not previously discovered. This phase is proposed by this study. This phase adds completeness to the security audit, since some vulnerabilities that cannot be discovered without exploring the source code of a system, can be discovered with the use of static analyzers.

This phase consists in performing a static analysis, which consists in analyzing the source code of an application to discover new vulnerabilities that classical dynamic analysis tools do not find. This extension enhances the completeness of the audit since new vulnerabilities that could not have been found are now discovered, thus resulting in a more complete audit.

The static code analysis is done on the code without executing it. The main difference from the programs used in the other sections, is that with those programs the analysis was done on the live application, while for this static analysis only the source code is needed. Static analysis makes it possible to detect vulnerabilities that otherwise would not be easily detected because the functionality will not be executed and will need encouragement or because a piece of code is not exposed to users.

We believe that this phase is key in the process of auditing because it discovers new vulnerabilities. A security audit aims to be as complete and sound as possible. Static analysis is yet another field to make progress at, because there are few static tools that focus on security and not performance o type-checking, for instance.

Code parsers usually have underneath them an Abstract Syntax Tree (AST). These ASTs represent

a program hierarchically, whose representation is the same as the compilers do. They are said to be of abstract syntax because the representation does not depend on the language itself that is being used. Thus, a function would have the same representation in C, Java or Python. In ASTs, the inner nodes represent operators and the last nodes, called leaves, represent variables. To identify vulnerabilities, static analyzers using this technique traverse the tree looking for flaws in a function's logic. One visual representation of such AST of a simple PHP function can be seen on Figure 3.

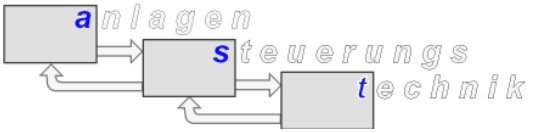

Figure 3: Sample AST of a PHP function.

There are multiple lists that compile the static tools that are open source. An example of one of the most complete list is (Analysis-Tools-Dev, 2021). After obtaining security tools from the previously commented list, a comparison between the following tools took place: py_find_injection, Sonarqube, Dlint, AttackFlow, pyre and Bandit. The tools selected for the project were: Banding, Dlint and Sonarqube.

4.4.1 Bandit

Bandit is a tool from PyCQA. According to its own GitHub description it is "a tool designed to find common security problems in Python code. To do this Bandit processes each file, builds an AST from it, and runs the appropriate plugins against the AST nodes. Once Bandit has finished scanning all files, it generates a report." The installation of the tool is very simple and execution is automatic. As an example, the following fragment shows a vulnerability discovered by Bandit.

```
>> Issue: [B110:try_except_pass] Try, Except,
Pass detected.
   Severity: Low   Confidence: High
   Location: ./src/addons/imedea_andrology
   /models/andrology_episode.py:317
   More Info: https://bandit.readthedocs.io/en
   /latest/plugins/b110_try_except_pass.html
```

4.4.2 Dlint

Dlint is a static analysis tool of Dlint-py. As with Bandit, its goal is to detect security flaws in Python code. Its installation and configuration are straightforward. As an example, the following fragment shows a vulnerability discovered by Dlint.

```
./src/addons/imedea_project/models/cycle_items
/defrosting.py:3:1: DUO107 insecure use of XML
modules, prefer "defusedxml"
./src/addons/imedea_project/models/cycle_items
/embryonic_study.py:3:1: DUO107 insecure use of
XML modules, prefer "defusedxml"
```

4.4.3 Sonarqube

Sonarqube is a platform that performs static code analysis. Sonarqube has several versions: the community version (open source) and the developer, enterprise and data center versions (paid). One of the most useful features of Sonarqube is the generation of reports. Unfortunately reports can only be generated natively in the paid versions and in the community version you have to rely on a plugin developed by the community (Cnescatlab, 2021). This plugin only supports up to version 8.2 of Sonarqube, from February 2020, when the latest is 8.9, from June 2021. That is why Sonarqube has been run twice in total, once with 8.2 to generate the report that will go later in the Executive Report, and another in which it is run without report. In fact, the two versions report totally different vulnerabilities and bugs, so it is useful to run both versions.

4.5 Executive Report

The Executive Report is a document that is given to the different departments of a company so that all staff can understand the work that has been done and the vulnerabilities in it. For confidentiality reasons, the report that has been generated for the present project cannot be publicly released.

This report is the part that a security auditor usually likes the least because it is based on creating documentation and not on investigating vulnerabilities. However, this is perhaps the most crucial part of the audit. During the previous sections there has not really been any material produced that would be of any benefit. The importance of this report is highly emphasized.

There are two main readers of this report: senior management, who are interested in the general high-level details, and the developers who have to fix the vulnerabilities through a series of technical steps, as well as some ideas to fix the vulnerabilities.

5 CONCLUSIONS AND FUTURE WORK

Companies around the world are undergoing digitization, which is driving them to use more and more

healthcare systems. These healthcare systems are not without security holes, which cybercriminals can exploit. Hence the need to perform security audits, especially in the audited system, because being a system in production in a health institution, the data it contains are very sensitive. In this project an audit has been done on a system in production implemented in an industrial scenario, more precisely in a healthcare system currently in production. The system uses Odoo 11 ERP to store their patients data.

This study reviews security audit orchestration frameworks. After having identified a gap where frameworks could be more complete, an extension is proposed by adding a phase to the auditing process.

This phase consists in performing a static analysis, which consists in analyzing the source code of an application to discover new vulnerabilities that classical dynamic analysis tools do not find. This extension enhances the completeness of the audit since new vulnerabilities that could not have been found are now discovered, thus resulting in a more complete audit.

Besides the addition of the new phase, the study shows how a real security audit on a live, industrial healthcare system is performed. More precisely, first, in the section of the scope, it is stated what can be audited. In the section of the analysis (4.2) tools are executed that identify computer vulnerabilities and in the section of the exploitation (4.3) it is tried to deepen in the most critical vulnerabilities. Another analysis is done in the static analysis section (4.4). This time, the tools are run only on the custom code of the modules. Finally, all the knowledge generated is collected in the Executive Report (4.5).

In light of the results, it can be concluded that critical and highly confidential systems, it should be subject of periodic security audits, in order to protect such confidentiality. In addition to that fact, the new proposed phase succeeds in discovering vulnerabilities that classic dynamic analysis tools cannot discover.

Future work might be oriented in various directions according to the examined perspectives and identified gaps, such as the social engineering one. A social engineering audit, which explores the human factor of a system, may be as important as a technical audit of the system, since there is no point in having a technically fortified system if employees later reveal sensitive information. The work could be expanded in regard to the phase of testing the source code by, for instance, giving specific step to achieve the desired completeness of testing the source code. The methods and comparison of different tools can also be expanded. Due to time and length constraints, more experimental results are needed.

Finally, it could be also investigated the option of including artificial intelligence to improve the analysis and perhaps the tools that use artificial intelligence can find vulnerabilities that others cannot.

ACKNOWLEDGMENTS

This work has been partially supported by the NICO project (PID2019-105455GB-C31) of the Spanish the Ministry of Science, Economy and University and NDT4.0 (US-1251532) of the Andalusian Regional Ministry of Economy and Knowledge.

REFERENCES

Analysis-Tools-Dev (2021). github.com/analysis-tools-dev/static-analysis. github. Online; Accessed on June 2021.

Burp Suite (2019). Burp suite scanner sample report. https://portswigger.net/burp /samplereport/burpscannersamplereport. Online; Accessed on July 2021.

Cnescatlab (2021). cnescatlab/sonar-cnes-report. github. https://github.com/cnescatlab/sonar-cnes-report. Online; Accessed on June 2021.

Gkoutzamanis, D. (2020). Five penetration testing frameworks and methodologies. cisotimes. https://cisotimes.com/five-top-penetration-testing-frameworks-and-methodologies/. Online; Accessed on March 2021.

ISACA (2021). State of cybersecurity 2021. https://www.isaca.org/go/state-of-cybersecurity-2021. Online; Accessed on June 2021.

ISO (2012). Information technology — security techniques — guidelines for cybersecurity. https://www.iso.org/standard/44375.html. Online; Accessed on June 2021.

OWASP (2017). Owasp top ten. https://owasp.org/www-project-top-ten/. Online; Accessed on June 2021.

OWASP (2019). Owasp application security verification standard. owasp. https://owasp.org/www-project-application-security-verification-standard/. Online; Accessed on June 2021.