Experimental Evaluation of Security Requirements Engineering Benefits

Jaouad Boutahar[1], Ilham Maskani[2], Souhaïl El Ghazi El Houssaïni[3]
[1,3] Systems, architectures and networks Team
EHTP, Casablanca, Morocco
[2] LISER Laboratory
ENSEM, Hassan II University
Casablanca, Morocco

***Abstract*—Security Requirements Engineering (SRE) approaches are designed to improve information system security by thinking about security requirements at the beginning of the software development lifecycle. This paper is a quantitative evaluation of the benefits of applying such an SRE approach. The followed methodology was to develop two versions of the same web application, with and without using SRE, then comparing the level of security in each version by running different test tools. The subsequent results clearly support the benefits of the early use of SRE with a 38% security improvement in the secure version of the application. This security benefit reaches 67% for high severity vulnerabilities, leaving only non-critical and easy-to-fix vulnerabilities.**

***Keywords*—Software security; security requirements engineering; security evaluation; security testing**

I. Introduction

Security Requirements Engineering (SRE) is the discipline that integrates security to Requirements Engineering, the very first step in the Software Development Life Cycle (SDLC). By adding security requirements to other system requirements during requirements engineering, a big improvement can be made in term of security vulnerabilities, software maintenance efforts and development costs. Moreover, OWASP , the leading organization in web application security, recommends focusing a big part of security flaws detecting efforts on the requirements engineering phase and the design phase[1]. There is related work which proves that it is critical to address security issues at the earliest phase, but few works try to measure just how much improvement can be obtained from applying an SRE approach. The goal of this paper is to make such a quantitative evaluation by developing two versions of the same web application, with and without using an SRE approach and evaluating their levels of security. The SRE approach that will be used is CompaSRE, a proprietary approach detailed in previous work[2]. For evaluation purposes, there's a plethora of testing methods and tools that could be used. A proper benchmark is needed to select the most appropriate. This paper is structured as follows. First, related work is discussed. Then, in the second section, the discipline of SRE is presented, along with definitions of its most important concepts, and the CompaSRE approach is explained. Then, in third section, the followed methodology is explained, along with the scope of the web application that will be developed for tests, and the selected test method and tools. Finally, the testing results and their variables are discussed in the fourth section.

II. Related Work

There is an abundance of security requirements engineering approaches. But when it comes to evaluating their performance, to the best of our knowledge, no source calculates how much is security improved by a certain SRE approach. Magnusson et al. tried to show how IT security investments can create value[3]. They studied models for return on investment on IT security in general. One of the models, developed by MIT, focused on proving the return of investment on secure software development, and showed that the earliest the security is addressed, the highest the benefit. This benefit was estimated at 21%. As reported in another paper [4], finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase. This is an evaluation of the financial cost of not thinking about security at the requirements engineering phase, which is their first number one recommendation on how to reduce software defect.

III. Security Requirements Engineering

This section presents the main SRE concepts, their definition and use in SRE. It also presents the CompASRE approach used to elicit and model security requirements for this experiment.

A. SRE approaches

An SRE approach refers to any method or process or framework that sets clear steps in order to elicit security requirements for a system to be at the Requirements Engineering phase. In a previous study, 9 approaches were studied. They go about eliciting requirements from different starting points: goals, users, or risks. But, ultimately, any SRE approach uses a different set of the same concepts. These concepts are drawn from both the fields of security and requirements engineering. All 9 approaches include identifying "goals", 7 of them identify threats, 6 of them identify stakeholders and 4 of them identify assets and risks. Table 1 offers a definition of these concepts, which is based on the ISO/IEC 27000:2016 vocabulary[5].

TABLE I. SRE CONCEPTS DEFINITIONS

Concept	Definition	Alternate labels
Stakeholder	Person or organization that can affect, be affected by, or perceive themselves to be affected by a decision or activity. Some approaches include other systems that have an interest in the IS. Are also included internal software agents to whom a goal will be assigned.	Actor, client, agent
Asset	Anything that has value to the organization, its business operations and their continuity, including Information resources that support the organization's mission (Data).	Information, Resource, Object
Goal	A Security objective that must be achieved by the system to be	Objective
Risk	Potential that threats will exploit vulnerabilities of an information asset or group of information assets and thereby cause harm to an organization	
Requirement	Need or expectation that is stated, generally implied or obligatory. Requirements are low level details of goals.	Goal, objective

B. CompASRE

The CompaSRE approach is the result of a personal previous work. It was designed as a comprehensive approach, incorporating the strengths and best practices found in existing approaches, and filling the gaps between them. It's based on the previous definitions and will be used in this experiment to elicit and model requirements. CompASRE, as illustrated in Fig. 1. below, is structured in five phases, each phase contains a set of activities to perform.

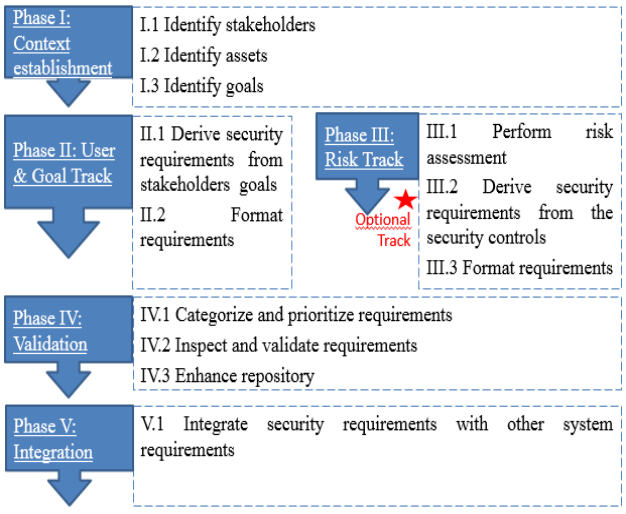

Fig. 1. CompASRE Steps.

The first phase "context establishment" aims to identify all common elements that are necessary to perform security requirements engineering in later phases. Then, the second phase "User & Goal track" aims to elicit requirements from the earlier identified goals. Security goals, as expressed by the stakeholders, are detailed and refined until reaching requirements. The third track is about deriving security requirement by doing a risk assessment, which implies analyzing threats and vulnerabilities. Risk assessment can be time and work consuming and was featured in only 4 out of the 9 studied SRE approaches. But assessing risks leads to thinking about security controls which might lead to new security requirements. Therefore, it was chosen to include this but keep it optional. The choice to perform it or not will depend on the type and size of the project. The more complex a project is, the more necessary it is to conduct a risk assessment. Once requirements are elicited (through phase 2 or 3), they must be modeled. The model created to be used with CompASRE or other SRE approaches is an extension of SysML requirements diagrams[6] and was presented in detail in previous paper [7]. In phase 4, elicited requirements are categorized and prioritized, then inspected for validation to resolve conflicts and eliminate redundancies. When an organization keeps a repository of requirements, this repository is to be updated. Finally, in phase 5, the security requirements are added to all other system requirements to complete the RE phase of the SDLC. Further validation might be necessary by the RE team.

IV. METHODOLOGY & TESTING

In this section, the methodology followed to conduct the study, the web app used as a test subject, the tests that were performed and the tools that were used are presented.

A. Methodology

The aim is to evaluate the positive impact of SRE on reducing system vulnerabilities, using specifically the CompASRE approach to elicit security requirements. To achieve that aim, first, the same web application was developed using 2 different software development lifecycles, resulting in 2 levels of security. The first version of the web app, the "*No Secure"* version, was developed following a classical waterfall lifecycle. This lifecycle was chosen because the app's functional perimeter is relatively small and unchanging. As for security, the way it was incorporated is the it's typically done in software projects where security is either not addressed at all (vulnerabilities will be patched after release) or is only addressed during the test phase. For this case, some minimal testing was done, in addition to correcting for the most obvious vulnerability "SQL injection". For the second *"Secure"* version, the same lifecycle was applied, but will be complemented by CompASRE. It means that, during the "Requirements Engineering" phase which is the first phase, CompASRE will be applied to elicit security requirements. Other later phases will be carried normally. Both versions were developed using the same language (JEE framework/java), same database management system and same development tools. Then, upon development completion, they were hosted on Microsoft's cloud solution Azure.

Finally, once both versions of the web app were hosted, security tests were conducted from a hacker's perspective. Quantitative results were obtained on vulnerabilities found in each version.

Both versions of the application are publicly available for fellow researchers on the following links: https://appgestionschool.azurewebsites.net/ for the secure

version; https://appgestionschoolnosecure.azurewebsites.net/ for the no secure version.

B. Test Web Application

The web application that will be used as a test subject is a grade management system for an engineering school. The primary criterion of choice was that the app must be security sensitive, which is the case here since it manages security sensitive information such as students' grades and their personal information. The web app's functional perimeter includes security problematic features such as authentication, filling forms, uploading and downloading files. But the perimeter was kept small on purpose because auditing the two versions of the web app and comparing their security levels is the main goal, not web app's complexity. The app offers 3 menus for:

1) teachers to enter students' grades

2) students to view grades, download records, submit a claim and review individual contractor lecturers

3) administrative staff to manage grades, claims and reviews and upload program's data.

C. Testing Method

Empirically proving SRE benefits relies on obtaining quantitative results on the vulnerabilities found and their levels of severity. To obtain such results, security tests can be conducted in 3 manners:

1) Static Application Security Testing (SAST): It's a white box testing method where the testers have access to the system's code. The code is scanned to systematically detect and eliminate security vulnerabilities

2) Dynamic Application Security Testing (DAST): It's a black box testing method applied on running applications from the outside.

3) Penetration Testing: Manuel conducting of an application penetration scenario to target a specific asset or vulnerability that require human intervention.

Table 2 summarizes the pros and cons of using each test method in relation to this experiment.

TABLE II. COMPARISON OF TESTING METHODS

Testing method	Pros	Cons
SAST	• Finds vulnerabilities sooner during SDLC (before deployment)	• No quantitative report on found vulnerabilities • Tests conducted from the inside • Focus on code
DAST	• Tests conducted from the outside • Automated repetitive tests • Quantitative results • Used post-development • No previous knowledge of the app is needed	• Web app had to be deployed on an internet facing server
Pen Testing	• Allows more targeted tests requiring human intervention • Allows analyzing and exploiting other system components such as OS and hardware	• Tests and results can't be reproduced for both versions • Costly in term of time and human resources

From this comparison, the DAST testing method was chosen because, tests must be conducted from a hacker approach (i.e. a malicious outsider seeking harm), rather than a developer or tester approach (i.e. a development team member seeking to improve security). Furthermore, DAST's automated and repetitive tests will give better quantitative results to compare security in each version such as the number of vulnerabilities.

D. Testing Tools

Many DAST tools are available to conduct tests. These tools work by executing predefined attack scripts that send a request to the web app. The web app's response to the tool is analyzed to determine the existence of a vulnerability. Each tool has its own scripts, and its own parameters to configure security tests[8]. Choosing and using only one tool would give biased data as a result. For this reason, it was decided to use different tools to gather extensive data. The criteria for choosing these tools were:

1) oriented towards application vulnerabilities rather than network vulnerabilities

2) not only targeting a certain type of vulnerabilities

3) detailed results: vulnerability severity, page where found, …

4) available installation and use documentation

5) available user interface

6) Free install or extended free trial

After applying these selection criteria, 3 tools were chosen: OWASP ZAP 2.7.0 [9], Vega 1.0[10] and Acunetix trial version 12.0.180911136 [11]. For each one of these tools, both versions of the web app were tested with the same tool parameters, to guarantee reliable results.

V. RESULTS

In this section, comparison results are discussed as obtained by the testing tools, along with the variables that could influence them. Remaining vulnerabilities are examined.

A. Results Discussion

Fig. 2. shows the number of security alerts reported by each tool. A security alert arises when one vulnerability is detected in a certain location of the web app (a location can be a page, a field within the page, an embedded resource …). So, if the same vulnerability is detected in many locations, it would rise as many alerts as the locations where it was found. For each tool, the benefit was calculated as the percentage of reduction in the number of alerts (1).

$$security\ benefit = 1 - \frac{nbr\ of\ alerts\ in\ secure\ version}{nbr\ of\ alerts\ in\ non\ secure\ version} \quad (1)$$

Every tool reported a decrease in the number of alerts, with a security benefit average of 38%. But benefits varied greatly between tools, with Vega reporting the highest benefit (68%), while Acunetix reporting the lowest (18%). As for the number of alerts, they were quite close to the average with an average number of alerts of 20,66 for the "non-secure" version, and 11,66 for the "secure" version.

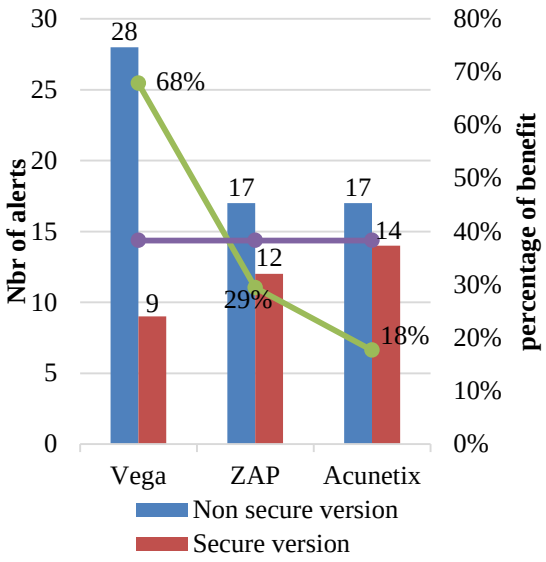

Fig. 2. Number of alerts and benefit per tool.

TABLE III. BENEFIT PER TOOLAND PER SEVERITY

Severity	Vega			ZAP			Acunetix			Average benefit
	No secure	*Secure*	*benefit*	*No secure*	*Secure*	*benefit*	*No secure*	*Secure*	*benefit*	
High	16	4	75 %	4	0	100 %	4	3	25 %	67%
Med	1	1	0%	4	3	25 %	8	8	0%	8%
Low	4	0	100 %	9	9	0%	3	2	33 %	44%
Info	7	4	43 %	0	0		2	1	50 %	46%

To get into the detail of alerts severity, Table 3 shows the number of security alerts, ranked by severity, as reported by each tool. Benefits are calculated by severity level. As shown in the diagram, the best benefits were obtained for high severity alerts with a 69% decrease. Indeed, high severity vulnerabilities are among those targeted early on during the SRE phase. As a result, the secure version of the web app is built with embedded security measures against those vulnerabilities.

As for the nature of the vulnerabilities that were found, table 4 presents the reported vulnerabilities for each version, along with the number of occurrences of each one. In the secure version, some vulnerabilities have disappeared (i.e. cross site scripting XSS), but some persisted, sometimes with fewer occurrences. This persistence of vulnerabilities can be explained, in some cases, by the fact that no requirement has been expressed against that vulnerability. In other cases, a requirement has been expressed against that vulnerability, but wasn't implemented during development (i.e. verbose error output). This is true in software development projects when a requirement is abandoned for time or cost reasons, or because of requirements mismanagement. There are also cases when a requirement is badly implemented, or implemented in only a few locations of the application, leaving some pages vulnerable. Regarding SQL injection, it's a high severity vulnerability that was considered during SRE, and all measures against it had been taken, but it was still reported by one tool in the secure version. Further manual tests could not confirm the vulnerability in the indicated location, so it's considered it as a false positive due to the tool itself.

TABLE IV. NUMBER OF ALERTS PER VULNERABILITY AND PER TOOL

Discovered Vulnerabilities	No secure version			Secure version		
	Vega	ZAP	Acunetix	Vega	ZAP	Acunetix
XSS Cross-site scripting	1		1			0
Integer Overflow	2					
SQL Injection		4	1			1
Page Fingerprint Differential Detected	12			3		
Session Cookie Without Secure Flag	1			1		
Apache tomcat infromation diclosure			2			2
Verbose Java error output	1		7	1		7
HTLM form without CSRF protection			1			1
String format error		1				
Javascript inter-domain sourcefile inclusion		3			3	
Autocomplete enabled in password field	4					
HTTP cache-control Header not set	1	3	1		3	
X-Content-Type-Options Header not set		3			3	
Lack of pretection against password brute force attack			1			1
Cookie Without Secure Flag	1		1	1		
X-Frame-Options-Header not set	3	3	1	1	3	1
Character Set Not Specified	1			1		
Blank Body Detected	1			1		
Possible sensitive information disclosure			1			1
TOTAL	**28**	**17**	**17**	**9**	**12**	**14**

B. Remaining Vulnerabilities

Even after applying the SRE approach, the secure version of the application still has some vulnerabilities. Indeed, no SRE approach claims to be able to eliminate all vulnerabilities. Furthermore, other phases of the SDLC play a big role in how secure a system would be. In the case of this application, the remaining vulnerabilities are not critical and can be corrected with a minimum of effort. More importantly, none of them comes from a design flaw which means that no redesign of the application will be necessary. Some could argue that, since SRE isn't failproof, all vulnerabilities could be left to be discovered and corrected at the end of the SDLC. This could be

true for applications with a simple scope. But, for more complex systems, correcting vulnerabilities at the end can either be too costly, too cumbersome (impacting quality) or sometimes downright impossible because of design constraints. So, even if applying SRE has its own cost and isn't failproof, it still delivers better built-in security and quality.

C. Variables

It's noticeable that the nature of the discovered vulnerabilities and the numbers of their occurrences vary a lot from tool to tool. There are many variables to consider when interpreting these results. Any change in these variables would influence the results. The greatest variable is the testing tool itself. It's true that the tools work in a similar way: they crawl the website to find URLs, they attack said URLs with proprietary scripts and malicious input, then they analyze how the web app responds.

But where they differ is: how deep do they crawl? what vulnerabilities are tested for? what script/input is used to detect the vulnerability? The answers to these questions can lead to big differences detection efficiency, leading to false negatives (existing vulnerabilities that go undetected), or false positives (vulnerabilities reported but don't exist)[8]. This analysis by severity is also biased by the fact that the same vulnerability can be considered with different levels of severity (i.e. verbose error output, which is like apache tomcat information disclosure, is high severity for Acunetix and medium severity for Vega). Tools also differ in their settings and parameters. ZAP offered more advanced parameters such as creating contexts, authenticated attacks, adjusting crawling depth… It was found that the same parameters couldn't be applied to all tools, but for each tool, the parameters were the same in each version. Last but not least, if the tests had been done at development phase, prior to deployment, SAST tools would have been used, giving different results.

The second big variable is due to the web application used as a test subject. The more complex an application is, the more locations there are to find vulnerabilities. Security also depends on the technology used for the application. It was noted that applications in truly compiled application languages (i.e. C, C++) are more secure (in terms of (regarding OWASP Top 10) than general-purpose bytecode languages (i.e. Java, .NET) while scripting (i.e. PHP, ASP) are even less secure[12].

Furthermore, the type of application (social network, e-commerce ...) and industry (finance, e-gov ...) also influences what vulnerabilities would be found[13].

Finally, the SRE approach used to elicit requirements is another variable. Each approach has a different set of steps to follow, and various approaches work differently for different projects[14]. The security requirements elicited may also vary, for the same approach, depending on how correctly the approach was applied.

VI. Conclusion & Perspectives

The aim of the paper was to quantify the benefit of using an SRE approach. To achieve that, two versions of the same web application were developed. The contribution of this research is the evaluation of the security level of each version, proving the benefit of SRE. It was found that the second version was 38% more secure than the first. High severity vulnerabilities are the more impacted and were decreased by 67%. Vulnerabilities that persisted were either overlooked during SRE, or mechanisms against them were poorly developed or not developed at all. Remaining vulnerabilities are not critical and easy to correct. These results depend on many variables related to the testing tools, the web application subjected to the tests, and the SRE approach that was applied.

Plans for future work are to further investigate all discovered vulnerabilities to detect false positives and determine how that may have influenced the results. Tests could be done on other types of applications of different technologies to mitigate the effect of these variables. It's also planned to improve CompASRE's efficiency after a detailed study of its results, challenges and lessons learned.

References

[1] "Testing Guide Introduction - OWASP." [Online]. Available: https://www.owasp.org/index.php/Testing_Guide_Introduction. [Accessed: 13-Oct-2016].

[2] I. Maskani, J. Boutahar, and S. El Ghazi El Houssaïni, "Analysis of Security Requirements Engineering: Towards a Comprehensive Approach," IJACSA Int. J. Adv. Comput. Sci. Appl., vol. 7, no. 11, pp. 39–45, Nov. 2016.

[3] H. S. Venter and Information Security South Africa, Eds., Peer-reviewed proceedings of the ISSA 2004 enabling tomorrow conference. ISSA, 2004.

[4] B. Boehm and V. R. Basili, "Top 10 list [software development]," Computer, vol. 34, no. 1, pp. 135–137, 2001.

[5] "ISO/IEC 27000:2016 - Information technology -- Security techniques -- Information security management systems -- Overview and vocabulary," ISO. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=66435. [Accessed: 20-Oct-2016].

[6] "What is SysML? | OMG SysML." [Online]. Available: http://www.omgsysml.org/what-is-sysml.htm. [Accessed: 14-Nov-2017].

[7] I. Maskani, J. Boutahar, and S. El Ghazi El Houssaïni, "Modeling Security Requirements: Extending SysML with Security Requirements Engineering Concepts," Int. J. Appl. Inf. Syst., vol. 12, no. 9, pp. 30–36, Dec. 2017.

[8] A. Doupé, M. Cova, and G. Vigna, "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners," in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2010, pp. 111–131.

[9] "OWASP Zed Attack Proxy Project - OWASP." [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. [Accessed: 01-Nov-2018].

[10] "Vega Vulnerability Scanner." [Online]. Available: https://subgraph.com/vega/. [Accessed: 01-Nov-2018].

[11] "Acunetix 14 Day Trial," Acunetix. .

[12] Veracode, "State of Software Security Report, Supplement to Volume 6: Focus on Application Development," 2015.

[13] Veracode, "State of Software Security Report, Volume 6: Focus on Industry Verticals," 2015.

[14] D. Mellado, C. Blanco, L. E. Sánchez, and E. Fernández-Medina, "A systematic review of security requirements engineering," Comput. Stand. Interfaces, vol. 32, no. 4, pp. 153–165, Jun. 2010.