Sarch-Knows: A Knowledge Graph for Modeling Security Scenarios at the Software Architecture Level

Jeisson Vergara-Vargas[1,2][0000−0001−5498−6619], Felipe Restrepo-Calle[1][0000−0003−4226−1324], Salah Sadou[2][0000−0001−8961−3142], and Chouki Tibermacine[3][0000−0002−2063−0291]

[1] Universidad Nacional de Colombia, Bogotá, Colombia
`{javergarav, ferestrepoca}@unal.edu.co`
[2] IRISA & CNRS, Université Bretagne Sud, Vannes, France
`salah.sadou@irisa.fr`
[3] LIRMM & CNRS, Univ Montpellier, Montpellier, France
`chouki.tibermacine@lirmm.fr`

Abstract. Security, as a software quality attribute, needs to be addressed from different perspectives and at different levels of the software life-cycle. One of these perspectives is the one that focuses on design decisions at the highest level, that is, at the architectural level. This paper presents a knowledge graph, called "Sarch-Knows", that models security scenarios based on the architectural design of a software system. The knowledge graph is based on different paths called scenarios, where each scenario covers the fundamental elements to meet a security property and the architectural elements on which the properties fall. This knowledge graph is being implemented as a Neo4j database on which queries can be issued to extract aggregated knowledge about security and architecture. This knowledge is scattered over many sources of documentation, like NIST, MITRE, databases, books and papers; which is why this graph can be considered as a starting option to establish an ordered scheme of this knowledge.

Keywords: Software Architecture · Security · Modeling · Knowledge Graph · Sarch.

1 Introduction

The architecture of a software system is defined from a series of elements and relationships, which constitute the most important structures of the system, fundamental to reason about it [1, 13]. These structures are essential to ensure compliance with the functional and non-functional requirements of the system. However, from the non-functional point of view, these structures are essential when it comes to ensuring quality attributes [12]. Although there is a wide variety of quality attributes, there are some that are indisputably relevant to all types of software systems. One of these is security. Security is the ability of a software system to protect the elements of the system, including data, from unauthorized access [1, 18]. Likewise, it is the ability to provide access to the different system actors that are authorized (users, components and external systems). Security as a software quality attribute is covered by the same fundamental elements of cybersecurity, among which are threats, weakness, attacks and risks. It can be identified that some contributions have been made in the identification of specific elements, at the architecture level, that can affect the security of a software system, among them the classifications of architectural weaknesses and vulnerabilities related to the application of security tactics [14, 16]. Although some methodological proposals have been presented to support the secure software development process, as in [20], there is currently no comprehensive contribution that provides a transversal description of the essential security concepts as well as specific concepts related to the architecture of a software system. In this context, this paper presents a knowledge graph where it is possible to model a complete security scenario, involving elements associated with cybersecurity and elements associated with software architecture. This conceptual modeling approach makes it possible to identify the flow of a possible security risk, from the identification of the threat and the respective weakness, to the architectural elements that are subject to this risk.

The remainder of this paper is structured as follows. Section 2 describes the related work to the context of the proposed work. Section 3 specifies and details the characteristics of the knowledge graph proposed. In Section 4 the approach of security scenarios is presented. Section 5 analyzes the applicability of the proposed work. Finally, Section 6 presents the conclusions and future work.

2 Related Works

Software architecture, as a field of knowledge, poses different strategies when designing and building a software system. On the one hand, architecture is responsible for defining the structure of the system, that is, the elements that make it up and the way in which they are related [13]. On the other hand, the architecture is responsible for defining mechanisms to meet non-functional requirements, particularly quality attributes. Security, as a software quality attribute, is addressed from the architectural point of view through two fundamental concepts. In the first place, the use of architectural tactics has been proposed [4, 11], as sets of design decisions that seek to guarantee the quality attribute. In the case of security, architectural tactics have been classified in different ways; however, the taxonomy is highlighted where the moment on which the tactic acts in the system to deal with an attack is taken into account [8, 1]. In this case, this can be seen at the level of detection, resistance, reaction and recovery. Moreover, the use of architectural patterns is also proposed to address recurring design problems. For security, these patterns seek to specify a concrete solution on the architecture, through the implementation of a particular architectural tactic [1, 5, 19].

At the level of the basic elements of cybersecurity, several works have been proposed, in which a relationship between these elements and the elements of the architecture of a software system is considered [15, 17, 10, 14, 16]. In this case, the information registered by NIST in the National Vulnerability Database (NVD) is identified as one of the most relevant [9], where specific records can be found on vulnerabilities identified over time, associated with multiple types of computer systems, including software systems. In the same way, the information published by MITRE is highlighted, through the Common Weakness Enumeration (CWE) [6], where a particular mapping of architectural concepts is presented with a set of weaknesses identified over time, associated with poor implementation (or null) of an architectural tactic in a software system [7].

3 Sarch-Knows: Knowledge Graph

The proposed knowledge graph is modeled from two fundamental perspectives: the *field of knowledge* and the *level of detail*. In the first place, the perspective by field of knowledge divides the graph into two parts; the first that includes the elements associated with software architecture, and the second that includes the elements associated with cybersecurity.

On the other hand, the level of detail perspective divides the graph into two other parts; the first that includes the abstract elements (lower level of detail), and the second that includes specific elements, corresponding to instances of the abstract elements (higher level of detail).

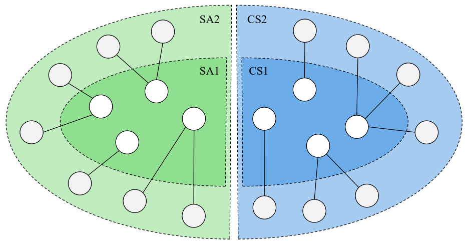

Fig. 1. Knowledge graph overview (SA: Software Architecture, CS: Cybersecurity).

Figure 1 presents a general overview of the knowledge graph structure, taking into account the two described perspectives. The green region corresponds to the field of knowledge: Software Architecture (SA), and the blue region corresponds to the field of knowledge: Cybersecurity (CS). Likewise, each region by field of knowledge presents the two levels of detail. SA1 and CS1 correspond to the minimum level of detail (abstract elements), and SA2 and CS2 correspond to the maximum level of detail (specific elements). The details of the graph are described below, based on the fields of knowledge. It is important to mention that both the abstract elements and the specific elements are presented in the graph as nodes. Likewise, the relationships between the abstract elements and specific elements are presented in the graph as edges/arcs. Abstract elements establish conceptual relationships between them. Abstract elements and specific elements establish instantiation relationships between them. While the specific elements establish security scenarios between them.

3.1 Software Architecture (SA)

The first part of the graph groups the relationships between different concepts related to the architecture of a software system. To comprehend the principles of software architecture, it is important to understand the generalities of the related abstract elements:

- *Architectural Element:* The fundamental unit of construction of a software system. Among its basic characteristics are: a set of responsibilities, a boundary, and a set of interfaces. These elements can include components, connectors, modules, layers, services, and messages [13].
- *System Structure:* A particular organization and arrangement of architectural elements within a software system. It can be considered as a set of architectural elements and their respective relationships [1, 13].
- *Component-and-Connector Structure:* structure of the system that groups those architectural elements that are present at runtime [2].
- *Component:* a computational element or data store that is present at runtime. [18, 2].
- *Connector:* a path of interaction at runtime between two or more components. [18, 2].
- *Architectural Tactic:* a design decision that influences the fulfillment of a quality attribute [8, 1].
- *Architectural Pattern:* an architectural solution to solve a recurring software design problem [18].

Figure 2 (a) presents the abstract elements associated with the software architecture perspective in the graph.

The software architecture perspective presents the basic idea of architecture responsibility. On the one hand, it presents the definition of the system structures, composed of a set of architectural elements and emphasizing the structure of components and connectors. On the other hand, it presents architectural tactics and architectural patterns, fundamental to achieving quality attributes; in this case, security.

3.2 Cybersecurity (CS)

The second part of the graph groups the relationships between the basic concepts of cybersecurity. These concepts are essential to understand and attend to the aspects related to the security quality attribute, from any point of view, including the architectural. Thus, to comprehend the principles of security, it is important to understand the generalities of the related abstract elements [9]:

- *Weakness:* a defect or deficiency in the design, construction or configuration of a software system.
- *Risk:* the possibility of an undesired occurring event or incident that has a negative impact on the security of a software system.
- *Attack:* a malicious attempt to compromise the security of a software system.
- *Threat:* any event, action or entity that has the potential to cause damage or compromise the security of a software system.
- *Countermeasure:* A measure or action taken to prevent, mitigate or neutralize an identified threat or risk. Its main objective is to protect a software system against possible attacks or security incidents.
- *Security Property:* a system's ability to protect the elements that compose the system, including data, from any event that may mainly generate a confidentiality, integrity and availability risk.

Figure 2 (a) presents the abstract elements associated with the cybersecurity perspective in the graph.

The security perspective presents the basic idea of treatment of the security quality attribute from the general perspective of cybersecurity. On the one hand, it presents the concept of weakness of a software system, which can affect a security property. This weakness can be exploited by an attack, and therefore can cause a risk. Attack that will always be associated with a threat. On the other, the concept of countermeasure is presented, as the element that can prevent the attack and therefore remedy the weakness so that the risk does not become effective.

3.3 SA-CS Connection

From the two perspectives presented (Software Architecture and Cybersecurity) the graph presents a main characteristic related to the connection point between the two fields of knowledge. Particularly, the concepts of architectural tactic and architectural pattern are taken, both for the security quality attribute, which support decisions and design solutions at the architectural level. In this way, both a tactic and a pattern can be considered as forms of implementation of countermeasures at the architecture level to guarantee security properties. Figure 3 presents the connection between the two perspectives and their respective elements.

This figure shows the relationship that exists between the two perspectives. Fundamentally, the relationship is given in terms of the concept of countermeasure, which is what will allow to remedy the vulnerability (a weakness instance) in the system, and therefore, mitigate the risk. Since the graph has a focus on the architecture of the system, in addition to providing the specification of

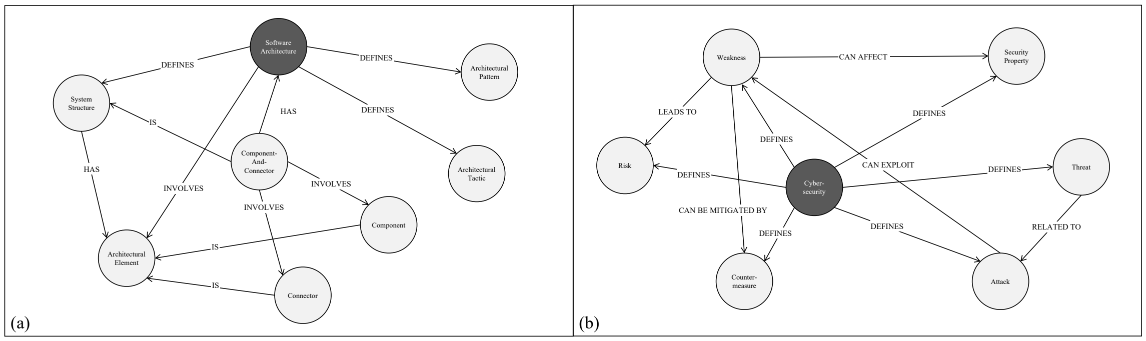

Fig. 2. First perspective: Software Architecture (a), and second perspective: Cybersecurity (b).

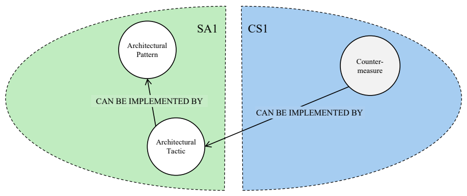

Fig. 3. Connection between the abstract elements of the two perspectives of the graph: Software Architecture (SA1) and Cybersecurity (CS1).

the architectural elements on which the possible risk falls, the graph also provides the definition of security tactics and security patterns that can be taken to address some security requirement. Thus, the graph works on the idea that tactics and patterns support the necessary countermeasures to address security requirements. In conclusion, the relationship between the two perspectives is created from the analysis of architectural tactics and patterns that can be taken at the design stage, in order to ensure a software system before and during its implementation.

4 Security Scenarios

4.1 Scenario Overview

Based on the general characteristics of the knowledge graph presented, the concept of *security scenario* is presented below. The security scenarios are based on the general scenario model for quality attributes presented by Bass et al. [1], which is composed of the following parts:

- **Source:** a *threat*.
- **Stimulus:** an *attack* that seeks to exploit a *vulnerability* (associated with a *weakness*).
- **Artifact:** the *system structure* (or part of it), composed by a set of *architectural elements*.
- **Environment:** normal execution of the system.
- **Response:** *Countermeasures* defined from *architectural tactics* and *architectural patterns*. Mechanisms used to control response.
- **Response Measure:** Evidence of the effectiveness of the applied *countermeasures*. Evidence that the *risk* became effective or not.

Based on the above, the knowledge graph presented allows the description of a security scenario from a subgraph composed of a initial set of nodes, associated with the abstract elements of the two perspectives: software architecture (SA1) and cybersecurity (CS1); and a second set of nodes, associated with the specific elements defined from the abstract elements. This last set of nodes represents a specific security case on the architecture of a software system (SA2, CS2). Specific elements are classified as *SASE* (Software Architecture Specific Elements) and *CCSK* (Current Common Security Knowledge).

4.2 Using Neo4j for Knowledge Modeling

With the purpose of making use of the knowledge graph, it has been modeled by means of a Neo4j database [4]. It is a database management system oriented to persistence and data query, through an approach of graph-based data model. Neo4j offers several advantages for graph databases. It provides efficient storage and retrieval of complex, interconnected data, enabling flexible, high-performance queries. With its native graph processing capabilities, Neo4j enables easy relationship traversal and analysis, making it ideal for applications involving knowledge graphs [3].

For the creation of the database, the characteristics of the knowledge graph described in Section 3 were taken into account. In this way, the database is made up of a set of nodes (and their respective relationships) associated with the abstract elements, both at the software architecture level and at the cybersecurity level. These nodes belong to a category (a label in Neo4j) called *abstract*. In addition, the complementary nodes of the database are created from the specific elements, that is, from the specific elements for each concept and that describe the security scenario. This means that a single node of an abstract element, can have multiple relationships with nodes that represent specific elements. These nodes belong to a category (a label in Neo4j) called *specific*.

In this way, a security scenario corresponds to a subgraph, formed from a logical relationship between specific elements of the software architecture and specific elements of cybersecurity. Thus, through the Neo4j query language (Cypher [5]) it is possible to filter a security scenario, and to obtain its respective subgraph, looking for all nodes corresponding to specific elements that have a logical relationship. A particular example of a security scenario is presented below. Table 1 summarizes the relationship between the abstract elements and the specific elements associated with the security scenario to be described.

The presented security scenario is related to a common weakness in different software systems designed as a Service-Oriented Front-End Architecture (SOFEA). This architecture (*system structure*) is generally composed of the following elements: a front-end component (presentation), a back-end component (business logic), a database component (data persistence), an HTTP connector for communicating a web browser with the front-end component, a REST connector for communicating the front-end component with the back-end component, and a database connector for communicating the back-end component with the database component. In this case, the weakness is related to the HTTP connector and refers to the fact that the protocol may not have a mechanism that allows verifying the integrity of the message that travels through that channel.

The *weakness*, called "Missing Support for Integrity Check" [6] is part of the Common Weakness Enumeration (CWE) published by MITRE, in its mapping on Architectural Concepts. In this case, the weakness falls on an *architectural element* of the architecture: the HTTP connector, which allows

[4] https://neo4j.com/

[5] https://neo4j.com/docs/getting-started/cypher-intro/

[6] https://cwe.mitre.org/data/definitions/353.html

Table 1. Example of a particular security scenario for a software system with a Service-Oriented Front-End Architecture (SOFEA).

Perspective	**Abstract Element(s)**	**Specific Element(s)**	
Software Architecture (SA)	System Structure	SOFEA (Service Oriented Front-End Architecture)	
	Architectural Element	HTTP connector between a web browser component and the Front-End component of the system	**sase1**
Cybersecurity (CS)	Weakness	CWE-353: Missing Support for Integrity Check	**ccsk1**
	Security Property	Integrity	**ccsk2**
	Attack	CAPEC-389: Content Spoofing Via Application API Manipulation	**ccsk3**
	Threat	Malicious User	**ccsk4**
	Risk	A08:2021 – Software and Data Integrity Failures	**ccsk5**
SA/CS	Countermeasure	Implement a Mechanism for Verifying Message Integrity	**ccsk6**
	Architectural Tactic	Detect Attacks >Verify Message Integrity	**sase2**
	Architectural Pattern	Intercepting Validator	**sase3**

communication between a web browser and the front-end component, connector in charge of sending messages coming from the client. Additionally, this scenario poses the *threat* of a malicious user attempting an *attack* called "Content Spoofing Via Application API Manipulation" [7] and which is part of the Common Attack Pattern Enumeration and Classification (CAPEC), also published by MITRE. The weakness leads to a *risk* called "Software and Data Integrity Failures" [8] mapped in the OWASP Top Ten classification.

Finally, the scenario presents an *architectural tactic* and an *architectural pattern* that serve as *countermeasures* to mitigate the risk generated by the weakness. In this case, the architectural tactic "Verify Message Integrity" proposes the use of techniques such as checksum and hash values to verify the integrity of the messages that travel through the HTTP connector. It is important to mention that this tactic is part of the "Detect Attacks" category. On the other hand, the architectural pattern "Intercepting Validator" is based on the addition of a new software element upfront the destination of messages (the front-end component), whose responsibility is to implement the described architectural tactic. Figure 4 presents the subgraph associated with the described security scenario, based on a query made on the Neo4j database. Here, the yellow nodes represent the abstract elements and the nodes with different colors represent the specific elements of the security scenario. See Table 1 for details of these elements.

The Cypher (Neo4j's graph query language) request made to return the described security scenario is:

MATCH p =
(a:sase {short_ name: "HTTP"})-[*]-(b:ccsk {name: "Missing Support for Integrity Check"})
RETURN p;

This means that the database is being searched the subgraph *(p)* containing all interrelated nodes that have the "HTTP" architectural element and the weakness "Missing Support for Integrity Check" in its path.

[7] https://capec.mitre.org/data/definitions/389.html
[8] https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/

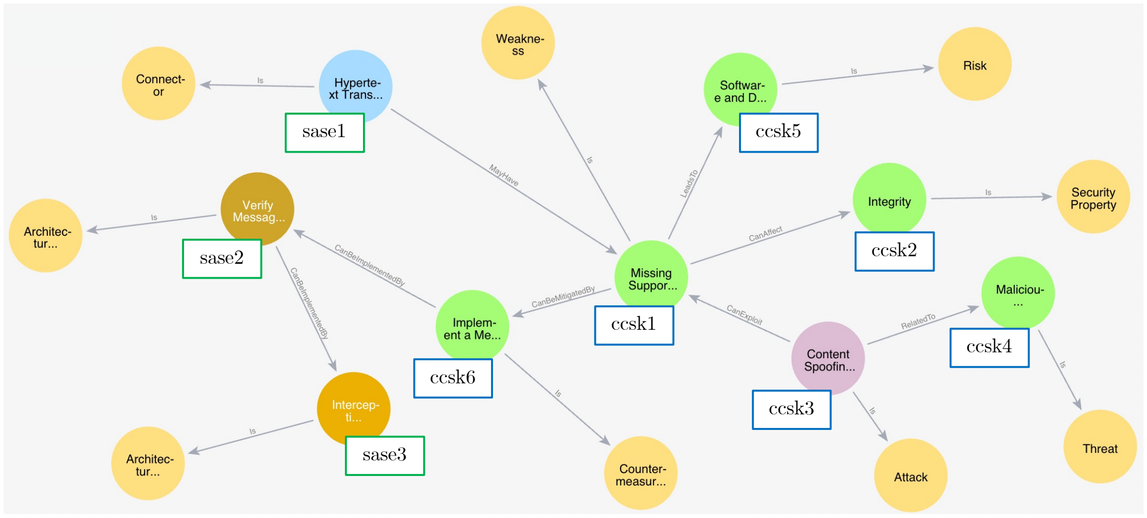

Fig. 4. Security scenario subgraph as a Neo4j database query.

5 Discussion

Guaranteeing the quality of a software system involves dealing with different quality attributes, among which security is one of the most important, and the architecture of the software system is essential to meet the related requirements. In this way, due to the complexity when dealing with this attribute and the number of possible scenarios that fall on a software system and where it is necessary to meet the security requirements, it is very important to have a source that synthesizes the fundamental elements. That contributes to the treatment of these security scenarios.

However, despite the fact that there are different sources of information where the elements that contribute to the description of a security scenario are related, there is no single resource that comprehensively covers and relates all the elements. For this reason, our knowledge graph proposal comprehensively conceives all the fundamental elements necessary to fully describe a scenario in which security in the architecture of a software system is sought to be addressed. The knowledge graph is implemented manually, from different sources of information, where the following stand out: the Common Weakness Enumeration (CWE) and the Common Attack Pattern Enumeration and Classification (CAPEC), provided by MITRE, the official documentation and the National Vulnerability Database (NVD) by NIST, different databases and the main bibliographical references on software architecture and cybersecurity.

At the implementation level, the base model of the graph is highlighted, which includes abstract elements of the two fields: software architecture and cybersecurity. This guarantees that all security scenarios are structured in the same way, keeping the formalism between the elements of the architecture and the solidity of the control scheme over the security of the software system. In the same way, each specific element is rigorously described, based on the identified sources of information and a corresponding analysis that allows the generation of logical relationships between each part of the scenario.

Based on the above, the applicability of the knowledge graph can be observed as follows. In the first place, the knowledge graph allows analyzing a security scenario at the level of the architecture of a software system. In analysis, it is based on the contribution of the graph when it comes to identifying vulnerabilities in a set of elements of the architecture. This can generate a security risk in the system, as well as the countermeasures that can be applied in the system to mitigate this risk. This is based on the point of view of a set of architectural tactics and patterns. In second place, all the specific elements modeled in the graph, based on the abstract elements, can have one or more security scenarios associated to them. This allows multiple security scenarios to be consulted, performing a filter by the scenario identifier, allowing multiple specific elements to be part of multiple security scenarios.

It is important to mention that vulnerabilities are not modeled in the knowledge graph since they are considered instances of weaknesses, that is, weaknesses identified or reported in real software systems.

The proposed knowledge graph can be used as a primary tool when carrying out the architectural design of a software system that requires meeting a set of security requirements, and therefore, serves as a basis to guide the construction of the system. The maintenance of the knowledge graph is based on the appearance of new reports of vulnerabilities and weaknesses in the sources of information taken as reference. The addition to the database is done using the Cypher query language, building the query from the abstract elements and the specific elements analyzed, equivalent to a new security scenario.

Finally, the works presented in Section 2 generally describe independent classifications when dealing with security. On the one hand, precise classifications of weaknesses at the architectural level are presented, but without a deep level of detail towards the architectural elements involved. On the other hand, works are presented that describe the different design decisions that can be taken to deal with security, but do not delve into the vulnerabilities that are sought to be remedied. For this reason, our knowledge graph proposes a joint perspective where, under the concept of security scenario, it is possible to detect vulnerabilities in a more precise way, thanks to the structure of the graph and the knowledge vocabulary that it incorporates.

6 Conclusions and Future Work

In this paper, we presented a knowledge graph for modeling security scenarios from the point of view of a software system architecture. This graph, implemented as a Neo4j database, models abstract elements in two fields: software architecture and cybersecurity, as well as specific elements that allow describing a security scenario in a software system architecture. The scenarios start from the weakness that can be exploited and that falls on a set of architectural elements, up to the countermeasures that can be applied to the system to mitigate the risk generated in terms of tactics and architectural patterns for security. The graph is created from different sources of information and allows a general overview of the elements involved when dealing with a security requirement at the software system architecture level.

As a future work, three paths are proposed. The first one is related to the definition of the strategy so that relevant security scenarios at the architectural level can be loaded into the database in a collaborative way, guaranteeing the rigor of the concepts involved, the guarantee of the sources of information and the precision of the new data. Secondly, it is pertinent to include a detailed and comparative evaluation of the proposed approach with other approaches, in order to analyze the real utility of this approach for decision-making at the architectural level, both in small and large software systems. This is due to the fact that in large software systems the number of security scenarios will grow exponentially and it is important to complement the proposal with data loading schemes and more automated data analysis, which make this tool very useful for a software architect. Finally, the possibility of extending this same knowledge graph idea to other quality attributes such as scalability, availability, among others, is raised.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, vol. 4th Edition (2022)
2. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures - Views and Beyonds, vol. 2nd Edition (2011)
3. Fernandes, D., Bernardino, J.: Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb (2018). https://doi.org/10.5220/0006910203730380
4. Fernandez, E.B., Astudillo, H., Pedraza-García, G.: Revisiting architectural tactics for security. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **9278**, 55–69 (10 2015). https://doi.org/10.1007/978-3-319-23727-5_5/COVER
5. Fernandez, E.B., Yoshioka, N., Washizaki, H.: Evaluating the degree of security of a system built using security patterns. ACM International Conference Proceeding Series (8 2018). https://doi.org/10.1145/3230833.3232821, https://dl.acm.org/doi/10.1145/3230833.3232821
6. MITRE: Common weakness enumeration (cwe), https://cwe.mitre.org/data/index.html
7. MITRE: Common weakness enumeration (cwe) - architectural concepts, https://cwe.mitre.org/data/definitions/1008.html
8. Márquez, G., Astudillo, H., Kazman, R.: Architectural tactics in software architecture: A systematic mapping study. Journal of Systems and Software **197**, 111558 (3 2023). https://doi.org/10.1016/J.JSS.2022.111558
9. NIST: Nvd - national vulnerability database, https://nvd.nist.gov/
10. Orellana, C., Villegas, M.M., Astudillo, H.: Mitigating security threats through the use of security tactics to design secure cyber-physical systems (cps). ACM International Conference Proceeding Series **2**, 109–115 (9 2019). https://doi.org/10.1145/3344948.3344994, https://dl.acm.org/doi/10.1145/3344948.3344994
11. Pedraza-Garcia, G., Astudillo, H., Correal, D.: A methodological approach to apply security tactics in software architecture design. 2014 IEEE Colombian Conference on Communications and Computing, COLCOM 2014 - Conference Proceedings (2014). https://doi.org/10.1109/COLCOMCON.2014.6860432
12. Richards, M., Ford, N.: Fundamentals of Software Architecture: an Engineering Approach (2020)
13. Rozanski, N., Woods, E.: Software Systems Architecture. Addison-Wesley, 2nd edn. (2012). https://doi.org/10.1017/CBO9781107415324.004
14. Santos, J.C., Peruma, A., Mirakhorli, M., Galstery, M., Vidal, J.V., Sejfia, A.: Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017 pp. 69–78 (5 2017). https://doi.org/10.1109/ICSA.2017.39
15. Santos, J.C., Suloglu, S., Ye, J., Mirakhorli, M.: Towards an automated approach for detecting architectural weaknesses in critical systems. Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020 pp. 250–253 (6 2020). https://doi.org/10.1145/3387940.3392222, https://dl.acm.org/doi/10.1145/3387940.3392222
16. Santos, J.C., Tarrit, K., Mirakhorli, M.: A catalog of security architecture weaknesses. Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings pp. 220–223 (6 2017). https://doi.org/10.1109/ICSAW.2017.25
17. Santos, J.C., Tarrit, K., Sejfia, A., Mirakhorli, M., Galster, M.: An empirical study of tactical vulnerabilities. Journal of Systems and Software **149**, 263–284 (3 2019). https://doi.org/10.1016/J.JSS.2018.10.030

18. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture - Foundations, Theory, and Practice. Wiley (2009)
19. That, M.T.T., Sadou, S., Oquendo, F.: Using architectural patterns to define architectural decisions. pp. 196–200 (2012). https://doi.org/10.1109/WICSA-ECSA.212.28
20. Uzunov, A.V., Fernandez, E.B., Falkner, K.: Assessing and improving the quality of security methodologies for distributed systems. Journal of Software: Evolution and Process **30**, e1980 (11 2018). https://doi.org/10.1002/SMR.1980, https://onlinelibrary.wiley.com/doi/full/10.1002/smr.1980 https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1980 https://onlinelibrary.wiley.com/doi/10.1002/smr.1980