A Catalog of Security Architecture Weaknesses

Joanna C. S. Santos, Katy Tarrit, Mehdi Mirakhorli
Rochester Institute of Technology, USA
{jds5109, ktics, mxmvse}@rit.edu

***Abstract*—Secure by design is an approach to developing secure software systems from the ground up. In such approach, the alternate security tactics are first thought; among them, the best are selected and enforced by the architecture design, and then used as guiding principles for developers. Thus, design flaws in the architecture of a software system mean that successful attacks could result in enormous consequences. Therefore, secure by design shifts the main focus of software assurance from finding security bugs to identifying architectural flaws in the design. Current research in software security has been neglecting vulnerabilities which are caused by flaws in a software architecture design and/or deteriorations of the implementation of the architectural decisions. In this paper, we present the concept of Common Architectural Weakness Enumeration (CAWE), a catalog which enumerates common types of vulnerabilities rooted in the architecture of a software and provides mitigation techniques to address them. The CAWE catalog organizes the architectural flaws according to known security tactics. We developed an interactive web-based solution which helps designers and developers explore this catalog based on architectural choices made in their project. CAWE catalog contains 224 weaknesses related to security architecture. Through this catalog, we aim to promote the awareness of security architectural flaws and stimulate the security design thinking of developers, software engineers, and architects.**

I. Introduction

Secure by Design is more increasingly becoming the mainstream development approach to ensure security and privacy of software systems. In this approach, security is built in the system from the ground up and starts with a robust architecture design. Security architectural design decisions are often based on well-known security tactics, defined as reusable techniques for achieving specific quality concerns. Security tactics provide solutions for enforcing the necessary authentication, authorization, confidentiality, data integrity, privacy, accountability, availability, safety and non-repudiation requirements, even when the system is under attack [5].

In order to ensure the security of a software system, not only it is important to design a robust security architecture (intended) but also it is necessary to preserve the (implemented) architecture during software evolution.

Previous empirical studies [6] reported that about 50% of security problems are the result of software design weaknesses such as misunderstanding architecturally significant requirements, poor architectural implementation, violation of design principles in the source code and degradations of the security architecture. Weaknesses in the architecture of a software system can have a greater impact on various security concerns in the system and, as a result, give more space and flexibility for malicious users. Therefore, secure by design shifts the main focus of software assurance from finding security bugs to identifying flaws and weaknesses in the design.

There is a fundamental difference between *Architectural weaknesses* and *Bugs*. While the latter are more code-level, such as buffer overflows caused by miscalculations, the former are at a deeper level and much more subtle and sophisticated [4]. Although a software system will always have bugs, recent studies show that the security of many software applications is breached due to weaknesses in the architecture [4], [11]. An example of architectural weakness is the *Use of Client-Side Authentication*. In this example, a client/server product performs authentication within the client code, but not in the server code, allowing the authentication feature to be bypassed via a modified client which omits the authentication check. This design decision creates a weakness in the security architecture, which can be successfully exploited by an intruder with reverse-engineering skills. While this example occurs during the design process, some architectural weaknesses can be introduced during the implementation of architectural decisions. Examples of such weaknesses are inappropriate compartment of components forming sand-boxing or improper validation of certificate expiration in the implementation of authentication tactic.

In a recent effort, the *IEEE Center for Secure Design* launched by the *IEEE Computer Society* [4] encouraged a shift towards finding and promoting the awareness of common design weaknesses with the goal of helping architects and developers to learn from previous mistakes. With this respect, this center released a list of the Top 10 most common Design Flaws. However, as of today, only a few examples of such design weaknesses have been obtained or published to help architects and developers to learn and avoid such flaws.

Motivation of This Research: to obtain more insight about the types and characteristics of common security architecture weaknesses at both design and code level.

More specifically we will be investigating the following research questions:

RQ1 *How can one assess the quality of a security architecture by employing a conceptual framework?*
RQ2 *What are the known security weaknesses that are rooted in a software architecture?*
RQ3 *What security tactics are more likely to have associated vulnerabilities?*

Therefore, in this paper we present *Common Architec-*

978-1-5090-4793-2/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSAW.2017.25

tural Weakness Enumeration (CAWE), a catalog of known weaknesses rooted in design or implementation of a security architecture which can result in vulnerabilities. Our proposed CAWE catalog can be used as a reference guide (cheat-sheet) by developers/designers to reason about potential weaknesses in the design or implementation of their architecture (Providing an answer for RQ1). CAWE catalog was built from an existing list of known types of software vulnerabilities, documenting all the possible types of vulnerabilities in software system. [1]). This list did not distinguish between coding and tactical weaknesses. In our work, we categorize all known type of security weaknesses as tactical and non-tactical. This would enable us to answer research questions RQ2 and RQ3.

The remainder of this paper is organized as follows: section II presents an overview of security tactics and the types of weaknesses associated to them; in section III-A, we present the CAWE catalog; in section IV, we describe how it can be used; in section VI, we summarize related work in security architecture and finally in section VII we conclude the paper.

II. Security Tactics as the Building Blocks of a Security Architecture

Security tactics are the building blocks of a security architecture. The implementation of a combination of security tactics is required to deliver a secure system, in which legitimate users are properly authenticated; access controls are captured and enforced so that only authorized users can access their designated functionality; user activities are audited so recovering from a malicious activity is feasible; information integrity is preserved, and similar security characteristics are addressed.

While these tactics provide a well-formed solution to address various security concerns, if they are not adopted and implemented carefully, they can result in breaches in the security architecture. So, in the remaining of the paper, we discuss several ways in which a security architecture and its constituent tactics can be flawed or degraded.

Even if the designed architecture fulfills all security requirements appropriately, previous studies have shown that the architecture can erode as the software evolves or may be incorrectly implemented in the code [8], [15]. Consequently, based on these observations, we can classify architectural weaknesses into **Omission**, **Commission** and **Realization**. These three types of weaknesses are defined below:

- **Omission Weaknesses:** are caused by *missing* a security tactic when it is necessary. An example of an omission weakness is to store sensitive data in a file *without encryption*. In this flaw, the architect overlooks the need of protecting sensitive data from unauthorized users and assumes that attackers would never have access to the file, thereby considering that the password stored in plain text would not correspond to a compromise of the system.
- **Commission Weaknesses:** refer to incorrect choice of tactics which could result in undesirable consequences. An example of this weakness is the *Client side authentication* mentioned earlier in this paper. While architects have made a design decision to satisfy the requirement of authentication of entities, the weakness in this design will enable attackers to bypass the authentication by implementing a modified client which does not have the authentication check. Another example of such architectural weaknesses is *Using a Weak Cryptography for Passwords* to achieve better performance while maintaining data confidentiality. In this weakness, the passwords are stored with an obfuscation mechanism which is computationally less complex but easier for attackers to guess. Consequently, such improper design choice makes it possible for attackers to recover passwords via an exhaustive search.
- **Realization Weaknesses:** In this case, appropriate security tactics are adopted but are incorrectly implemented. For instance, a developer hard coded a URI in the implementation of the "Manage User Sessions" tactic, resulting in a session fixation flaw which enables a hacker to access a resource by following that URI [16].

[1]http://cwe.mitre.org/

III. Proposed Work to Answer Research Questions

To answer our research questions, we first describe the process followed to build the catalog of *Common Architectural Weakness Enumeration* (CAWE). We then show how this catalog, which is accessible via an interactive website, can be used to assess the quality of a security architecture. Subsequently, we answer our research questions regarding the characteristics and types of common architectural weaknesses using this catalog.

A. A Catalog of Architectural Weaknesses

The establishment of this catalog was sponsored by the US Department of Homeland Security. The CAWE catalog was built from an existing list of common types of vulnerabilities (the Common Software Weaknesses Enumeration). This list documents about 1,000 software weaknesses, but it does not clearly distinguish architectural weaknesses (i.e. security issues rooted in software architecture) from purely programming issues. Thus, we systematically categorized the entries in this list into *architectural weaknesses* and *coding bugs*. This allowed us to identify common architectural weaknesses and security tactics they could affect. Currently, our CAWE catalog has 224 architectural weaknesses categorized among 11 security tactics.

A CAWE entry describes an architectural weakness in a software system resulting in a vulnerability (see the full catalog at http://design.se.rit.edu/catalog). An instance of a CAWE entry is shown in Table I. This weakness results in the *Exposure of Data Element to Wrong Session* and is due to an incorrect implementation of the *Manage User Sessions* tactic [7]. Each CAWE entry is composed of several sections, such as impacted security tactic, type of flaw, textual description, source code examples, mitigation techniques and methods for detecting the flaw.

TABLE I
An Example of an Architectural Weakness from the Catalog

Impacted Tactic	Name: Manage User Sessions Description: Retains the information or status about each user and his/her access rights for the duration of multiple requests	**Impact Type**	Realization Flaw
Weakness	Title: Exposure of Data Element to Wrong Session (CWE-488) Description: The product does not sufficiently enforce boundaries between the states of different sessions, causing data to be provided to, or used by, the wrong session. **(...)** Common Consequences: • Confidentiality (Technical Impact: Read Application Data) Demonstrative Example: The following Servlet stores the value of a request parameter in a member field and then later echoes the parameter value to the response output stream. While this code will work perfectly in a single-user environment, if two users access the Servlet at approximately the same time, it is possible for the two request handler threads to interleave in the following way: Thread 1: assign "Dick" to name Thread 2: assign "Jane" to name Thread 1: print "Jane, thanks for visiting!" Thread 2: print ``Jane, thanks for visiting!". Thereby, showing the first user the second user's name. `public class GuestBook extends HttpServlet {` `String name;` `protected void doPost(HttpServletRequest req, HttpServletResponse res) {` `name = req.getParameter("name");` `...` `out.println(name + ", thanks for visiting!");` `}` `}` Potential Mitigations: • *Architecture and Design Phase:* Protect the application's sessions from information leakage. Make sure that a session's data is not used or visible by other sessions. In a multithreading environment, storing user data in Servlet member fields introduces a data access race condition. Do not use member fields to store information in the Servlet. • *Testing Phase:* Use a static analysis tool to scan the code for information leakage vulnerabilities (e.g. Singleton Member Field). Attack Patterns: • CAPEC-59 Session Credential Falsification through Prediction • CAPEC-60 Reusing Session IDs (aka Session Replay)		

B. Creation Process

To establish the CAWE catalog, we followed a *systematic* process to *classify* weaknesses from the common software weaknesses list (CWE) into *architectural weaknesses* and *bugs* as well as to categorize the *architectural weaknesses* per security tactic. This process followed two steps: first, we collected an extensive list of security tactics published in the community, extracted information about the solution proposed by the tactic and keywords which summarize the tactic [1], [7]. This collection helped to identify the building blocks of any security architecture. Second, we conducted a thorough analysis of the entries of the CWE list to verify which instances are caused by omission, commission or realization weaknesses.

Each entry in the CWE list can be of four types[2]: *View*, i.e. groups weaknesses in a given perspective, *Category*, i.e. types of weaknesses based on a common attribute, *Weakness*, i.e. an actual security issue, and *Compound Element*, i.e. a security issue due to the occurrence of other weaknesses in a time sequence. For each *Weakness* and *Compound Elements* types, the collection provides a set of related information, such as its description, mitigation techniques, code examples, and so forth. Hence, among 1,004 entries in the version 2.9 of the CWE list, we retrieved a subset of 727 entries which are of type *Weakness* or *Compound Elements* (*Categories* and *Views* entries were not included as they served more as a grouping of software weaknesses than an actual type of vulnerability). Then, we searched a tactic's name or keywords related to the tactic in the CWE name and description. The result of this search is a list of *potential* connections between security tactics and generic software vulnerability types (CWEs). Lastly, we manually analyzed provided data of all 727 collected instances (such as their descriptions, mitigation techniques, consequences and attack patterns) to confirm whether these potential connections indeed existed as well as verify whether there was not any missing connection. This way, once a CWE entry was considered to affect a security tactic, we classified it as an *architectural weakness* and annotated it with (i) the security tactic related to the weakness and (ii) the type of impact (commission, omission or realization flaw).

[2]The complete structure of the CWE collection can be found at MITRE's Website: https://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd.

To better illustrate how this classification happened, we consider the following example: the data of the CWE-354, "Improper Validation of Integrity Check Value", contains some of the keywords of the security tactic "Verify Message Integrity". Thus, this CWE is *potentially* related to that tactic. When we manually inspected this CWE instance, we noticed that it described a situation where a software fails to verify the checksums, i.e. extra data used to check the message integrity, which can make the software to potentially accept corrupted or modified messages. Hence, this CWE is actually related to the "Verify Message Integrity" tactic. Furthermore, if the CWE was tagged with the time of introduction as "Architecture and Design", we inspected if the CWE discussed that the issue occurred because of lack of a security tactic. For instance, the CWE-306 ("Missing Authentication for Critical Function") is caused by the absence of implementation of the "Authenticate Actors" tactic.

To minimize inherently biases in this manual analysis, four individuals worked independently over all these 727 entries to categorize them. Once they completed their analysis, results were double-checked and for entries with disagreements (84 in total), they discussed their rationale and defined what would be the appropriate classification result.

TABLE II
Total Number of Weaknesses per Security Tactics

Security Tactic	# CAWEs	Realization	Omission	Commission
Audit	6	3	1	2
Authenticate Actors	29	12	2	15
Authorize Actors	60	38	16	6
Cross Cutting	9	3	3	3
Encrypt Data	38	18	13	7
Identify Actors	12	10	2	0
Limit Access	8	7	0	1
Limit Exposure	6	6	0	0
Lock Computer	1	0	0	1
Manage User Sessions	6	5	0	1
Validate Inputs	39	35	4	0
Verify Message Integrity	10	6	4	0

C. Overview of the CAWE Catalog

The CAWE catalog currently contains 224 flaws which were categorized based on their impacts over 11 security tactics. To allow architects and developers browsing the catalog, we developed an interactive Web-based solution in which they can explore potential weaknesses based on architectural choices made in their project. In this solution, which is publicly available at http://design.se.rit.edu/catalog, users can select the security tactics of their project and visualize the associated weaknesses. For this purpose, this catalog could help architects and developers to learn how to avoid common architectural

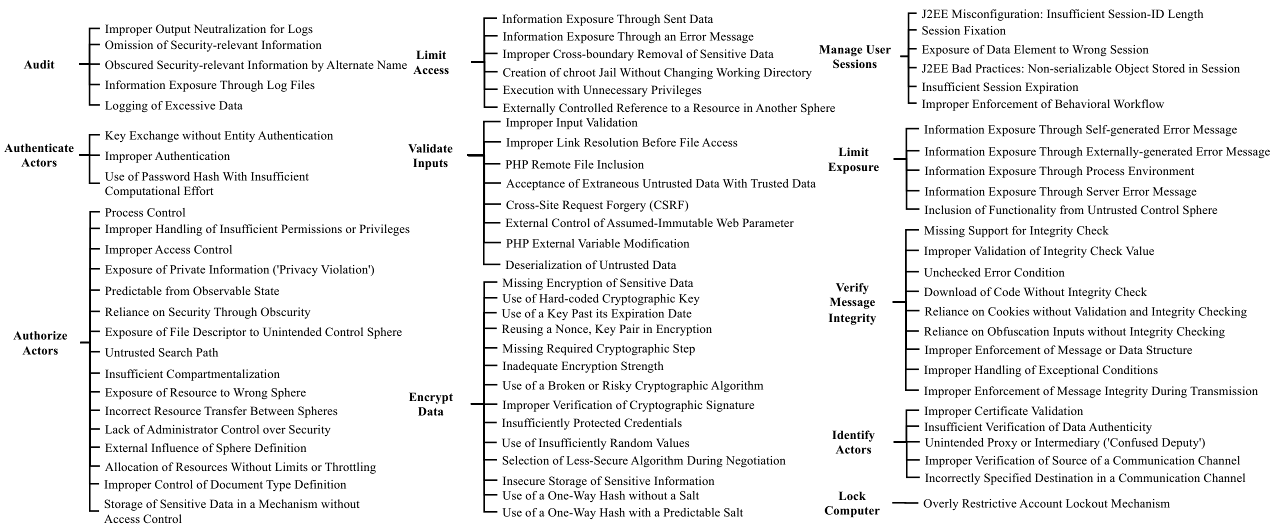

Audit
Improper Output Neutralization for Logs
Omission of Security-relevant Information
Obscured Security-relevant Information by Alternate Name
Information Exposure Through Log Files
Logging of Excessive Data
Authenticate Actors
Key Exchange without Entity Authentication
Improper Authentication
Use of Password Hash With Insufficient Computational Effort
Authorize Actors
Process Control
Improper Handling of Insufficient Permissions or Privileges
Improper Access Control
Exposure of Private Information ('Privacy Violation')
Predictable from Observable State
Reliance on Security Through Obscurity
Exposure of File Descriptor to Unintended Control Sphere
Untrusted Search Path
Insufficient Compartmentalization
Exposure of Resource to Wrong Sphere
Incorrect Resource Transfer Between Spheres
Lack of Administrator Control over Security
External Influence of Sphere Definition
Allocation of Resources Without Limits or Throttling
Improper Control of Document Type Definition
Storage of Sensitive Data in a Mechanism without Access Control
Limit Access
Information Exposure Through Sent Data
Information Exposure Through an Error Message
Improper Cross-boundary Removal of Sensitive Data
Creation of chroot Jail Without Changing Working Directory
Execution with Unnecessary Privileges
Externally Controlled Reference to a Resource in Another Sphere
Validate Inputs
Improper Input Validation
Improper Link Resolution Before File Access
PHP Remote File Inclusion
Acceptance of Extraneous Untrusted Data With Trusted Data
Cross-Site Request Forgery (CSRF)
External Control of Assumed-Immutable Web Parameter
PHP External Variable Modification
Deserialization of Untrusted Data
Encrypt Data
Missing Encryption of Sensitive Data
Use of Hard-coded Cryptographic Key
Use of a Key Past its Expiration Date
Reusing a Nonce, Key Pair in Encryption
Missing Required Cryptographic Step
Inadequate Encryption Strength
Use of a Broken or Risky Cryptographic Algorithm
Improper Verification of Cryptographic Signature
Insufficiently Protected Credentials
Use of Insufficiently Random Values
Selection of Less-Secure Algorithm During Negotiation
Insecure Storage of Sensitive Information
Use of a One-Way Hash without a Salt
Use of a One-Way Hash with a Predictable Salt
Manage User Sessions
J2EE Misconfiguration: Insufficient Session-ID Length
Session Fixation
Exposure of Data Element to Wrong Session
J2EE Bad Practices: Non-serializable Object Stored in Session
Insufficient Session Expiration
Improper Enforcement of Behavioral Workflow
Limit Exposure
Information Exposure Through Self-generated Error Message
Information Exposure Through Externally-generated Error Message
Information Exposure Through Process Environment
Information Exposure Through Server Error Message
Inclusion of Functionality from Untrusted Control Sphere
Verify Message Integrity
Missing Support for Integrity Check
Improper Validation of Integrity Check Value
Unchecked Error Condition
Download of Code Without Integrity Check
Reliance on Cookies without Validation and Integrity Checking
Reliance on Obfuscation Inputs without Integrity Checking
Improper Enforcement of Message or Data Structure
Improper Handling of Exceptional Conditions
Improper Enforcement of Message Integrity During Transmission
Identify Actors
Improper Certificate Validation
Insufficient Verification of Data Authenticity
Unintended Proxy or Intermediary ('Confused Deputy')
Improper Verification of Source of a Communication Channel
Incorrectly Specified Destination in a Communication Channel
Lock Computer
Overly Restrictive Account Lockout Mechanism

Fig. 1. High-Level Overview of the CAWE Catalog

issues in their software. Since the catalog is organized per security tactics, architects and developers can identify potential weaknesses related to a particular security tactic. For instance, in the case of the "Authenticate Actors" tactic, "Authentication Bypass" or "Client-side Authentication" flaws can occur in a system which implements this tactic. As another example, a common design decision is to apply "Encrypt Data" tactic to achieve data confidentiality. However, developers and architects may overlook the properties of encryption algorithms, making incorrect assumptions about their usage [11]. Since understanding the encryption algorithm is crucial to properly secure data, our catalog enumerates a set weaknesses which can guide them to obtain such knowledge. Examples of such weaknesses are: *CAWE-328 Reversible One-Way Hash* and *CAWE-780 Use of RSA Algorithm without Optimal Asymmetric Encryption Padding*.

D. Answer to the Research Questions

Once we established this catalog, we were able to answer the research questions presented in Section I.

RQ1: How can one assess the quality of a security architecture by employing a conceptual framework? Our proposed catalog is built based on classifying common known vulnerability types (weaknesses) into architectural and non-architectural. This catalog also connects architectural weaknesses to their underlying security tactic. Therefore, this catalog reflects the current knowledge of software vulnerabilities captured within the community. This catalog can be used as a framework to assess the quality of a security architecture. Designers/developers can use the catalog as a cheat-sheet, in which they can look up for a security tactic adopted in their project, and identify weaknesses that need to be investigated in their project. The catalog can facilitate security architecture inspection, guide code reviews and threat modeling activities. The catalog not only lists weaknesses associated to a security tactic, but also provides mitigation techniques to prevent or fix weaknesses.

RQ2: What Weaknesses are Rooted in the Security Architecture? From the catalog, we observed that among 727 software weaknesses we inspected from the CWE list, 224 known software weaknesses are rooted in the security architecture of a software. Figure 1 presents examples of these *architectural weaknesses* from the CAWE catalog for the collected tactics. Since some architectural weaknesses are a specialization of other entries (i.e. children of other entries), we present in this figure only the higher-level architectural weaknesses (i.e. we kept only the root elements due to space constraints).

RQ3: What Security Tactics are More Likely to Have Associated Vulnerabilities?

To answer this question, we computed the total number of flaws identified for each security tactic. Table II shows the number of weaknesses per tactic along with a breakdown by weakness type (omission, commission and realization weaknesses). In this table, we observed that the "Authorize Actors" tactic, which is used to ensure that only legitimate users can access data and/or resources, need to be implemented and tested more carefully. This tactic is subject to a higher number of known weaknesses if not implemented correctly. Similarly, "Validate Inputs" and "Encrypt Data" needs caution to avoid incorrect assumptions during its design and/or implementation.

IV. Examples of Architectural Weaknesses

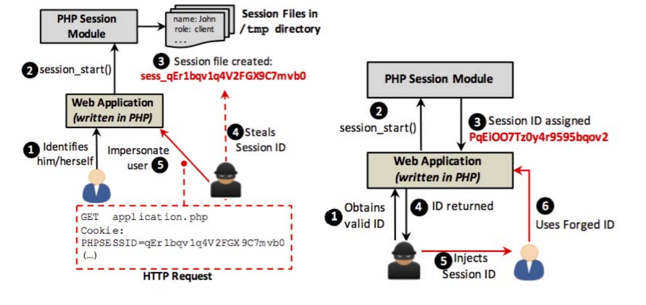
In this section, we scrutinize two instances of architectural weaknesses in PHP and provide a walk-through explanation of how they occurred and were exploited.

The *Manage User Sessions* tactic allows keeping track of who is using the system at a given time by managing a session object which contains all relevant data associated to the user and the session [14]. In this tactic, every user is assigned an exclusive identifier (*Session ID*), which is utilized for both identifying users and retrieving the user-related data. Since session IDs are sensitive information, this tactic may be affected by two main types of attacks [16]:

- *Session Hijacking*: an attacker impersonates a legitimate user through stealing or predicting a valid session ID.
- *Session Fixation*: an attacker has a valid session ID and forces the victim to use this ID.

• **Session Hijacking Attack**

The *Session Hijacking* can be facilitated by the architectural weakness of "not securing the storage of session identifiers". As reported in the CVE-2002-0121, the PHP project vesions 4.0 through 4.1.1 suffered from this architectural flaw because its original design stored each data session in plain textual files in a temporary directory without using a security tactic to store these session files in a secure way (such as encryption).



```
GET  application.php
Cookie:
PHPSESSID=qEr1bqv1q4V2FGX9C7mvb0
(...)
```

(a) (b)

Fig. 2. Examples of Architectural Weakness in PHP Facilitating the (a) Session Hijacking and (b) Session Fixation.

Figure 2(a) shows a scenario in which the weakness could be exploited. First, a legitimate user successfully identifies him/herself to the application. This causes the Web application written in PHP to start a session for the user through invoking the `session_start()` from the PHP's `session` module. Then, the `session` module in the PHP assigns a session ID for the user and creates a new file named as "sess_qEr1bqv1q4V2FGX9C7mvb0" to store the data about the user's session. At this point, the security of the application is compromised when an attacker observes the session file name and realizes that the user's session ID is equal to "qEr1bqv1q4V2FGX9C7mvb0". Subsequently, the attacker is able to impersonate the user by sending a cookie (`PHPSESSIONID`) in a HTTP request with this stolen Session ID. The Web application, after calling functions from the PHP's `session`, verifies that the session ID provided matches the user's data so, the application considers that the requests are being made by a legitimate user.

From this scenario, we can observe that such architectural weakness can lead to many consequences. First, if the user has an administrative role in the application, the attacker will be able to perform all the administrative tasks. Second, the attacker may be able to read the contents of the session file, thereby accessing data about the user, which may be sensitive, that the attacker is not supposed to have access to. It is important to highlight that such flaw affects not only the "Manage User Sessions" tactic, but also other security tactics such as "Authenticate Actors" and "Authorize Actors", which use the session data to perform authentication and users' access control.

• **Session Fixation Attack**

An example from PHP of an architectural weakness which facilitates the session fixation attack is discussed in the CVE-2011-4718. The root cause of this vulnerability was in the implementation of the "Manage User Sessions" tactic (i.e. there is a realization weakness). The main problem is that the session management implementation was accepting uninitialized session IDs before using it for authentication/authorization purposes. The function responsible for validating session IDs only verified whether the ID contains a valid charset and length, but did not check whether the ID actually exists and it is associated with the client performing the HTTP request. Failing to verify the existence of the ID exposes the system to session fixation attacks.

Figure 2(b) shows how this architectural vulnerability is exploited. It starts with the attacker establishing a valid session ID (steps 1 to 4). Next, the attacker induces the user to authenticate him/herself in the system using the attacker's session ID (steps 5 and 6). Since the forged ID (PqEiOO7Tz0y4r9595bqov2) is well-formed (valid size and charset), the PHP session accepts it as the ID for the user's session, even though the ID was not assigned to the user at first, but to the attacker account.

V. Planned Next Steps for this Work

As next step, we plan to evaluate the usage of the CAWE catalog in different development contexts:

- **Architectural Risk Analysis**: Architectural risk analysis is a systematic approach for evaluating design decisions against quality requirements. For architectural risk analysis to be effective, evaluators need to have a previous knowledge of architectural flaws based on the software context such as its requirements, architectural tactics applied, etc. Consequently, the CAWE catalog could be used as guidance when performing such assessment. This guidance is twofold: on one hand, the enumerated flaws of omission could be used as a road map for evaluators to identify missing relevant architectural decisions; on the other hand, the categorized flaws of commission and realization allows spotting issues in the current architecture of the system, being the first step to assess its impacts and risks. In a set of subject studies we will examine the usage of the catalog in this context.
- **Security Training**: Past experiences in industry lead to the creation of security-driven software development processes, which emphasize security concerns early in the software development lifecycle, such as CLASP (Comprehensive, Lightweight Application Security Process) and Microsoft's SDL (Security Development Lifecycle). A common aspect of these processes is the recommendation of providing proper *training* of the employees to develop a common background in software security [17]. With this respect, our catalog could be used to aid such training by promoting the awareness of the potential architectural issues that their systems may be exposed to due to common mistakes. We will conduct further

educational experiments to examine whether usage of catalog will help the students better understand architectural weaknesses.

- **Threat Modeling**: For modeling the threats of a software, an organization usually performs brainstorming sessions. In this context, the CAWE could be used in those sessions for obtaining insights. In fact, existing in experiences in the industry report the usage of threat libraries, built from a small subset of weaknesses from the CWE list, for aiding this threat modeling process [3]. We will compare our approach with similar techniques when it is used to facilitate threat modeling.

VI. Related Research

Currently, there are many research work and books focused on the identification, categorization and detailing of security tactics [1], [10], [13]. In this work, however, we take one step further towards observing how these tactics could be compromised when incorrectly implemented or inappropriately designed, thereby filling the existing gap by overcoming some security challenges from avoiding architectural flaws.

The usage of security knowledge bases to help developers and engineers in their daily activities have been discussed in the research community. In this matter, security ontologies which represent knowledge within the security domain, have been created to support some activities (e.g. requirements engineering and quantitative risk analysis). These ontologies, however, do not introduce architectural concepts [2]. In this context, similarly to a security ontology, Wu et al. [18] proposed *semantic templates*, which are a structured description of generic patterns of relationship between software components, faults and security consequences built on top of the CWE list and the vulnerabilities reported in the National Vulnerability Database. Again, these templates do not differentiate architectural concerns.

In addition to knowledge bases, some research work focused on methodological architecture-related activities for engineering secure software. For example, Pedraza-Garcia et al. [9] described activities, tools and notations to specify the security requirements of a software and guide the architect to select the architectural tactics which would better address the requirements, in order to minimize the impacts of design decisions being informally done and dependent on the architect's experience to select architectural choices. Another example is [12], where they proposed a vulnerability-oriented architectural analysis approach which reuses the knowledge from known vulnerabilities as a checklist when evaluating and designing an architecture. Unlike the proposed method, these efforts focused on helping to build a secure software through software development activities.

VII. Conclusion

This paper presented the new concept of CAWE (Common Architectural Weakness Enumeration), a catalog of common types of architectural weaknesses. This catalog constitutes an effort towards stimulating security thinking in developers to avoid fundamental design problems in both architectural design and implementation time and to help researchers to develop novel techniques to identify and mitigate such flaws. Currently, the catalog enumerates 224 architectural weaknesses, documenting how these weaknesses may affect security tactics. As a future work, we plan to evaluate our catalog with security experts, looking forward to expanding the catalog.

Acknowledgments

This work was partially funded by the US National Science Foundation under grant numbers CCF-1543176, CNS-1629810 and IIP-0968959 under funding from the S2ERC I/UCRC program and US Department of Homeland Security.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[2] C. Blanco, J. Lasheras, R. Valencia-García, E. Fernández-Medina, A. Toval, and M. Piattini. A systematic review and comparison of security ontologies. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 813–820. Ieee, 2008.

[3] D. Dhillon. Developer-driven threat modeling: Lessons learned in the trenches. *IEEE Security & Privacy*, (4):41–47, 2011.

[4] I. C. for Secure Design. Avoiding the top 10 software security design flaws. http://cybersecurity.ieee.org/center-for-secure-design/, 2015. (Accessed on 10/06/2016).

[5] M. Hafiz, P. Adamczyk, and R. E. Johnson. Growing a pattern language (for security). In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 139–158, New York, NY, USA, 2012. ACM.

[6] G. McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[7] J. C.-H. Mehdi Mirakhorli. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Trans. Software Eng.*, 2015.

[8] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic centric approach for automating traceability of quality concerns. In *International Conference on Software Engineering, ICSE (1)*, 2012.

[9] G. Pedraza-Garcia, H. Astudillo, and D. Correal. A methodological approach to apply security tactics in software architecture design. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–8. IEEE, 2014.

[10] C. Preschern. Catalog of security tactics linked to common criteria requirements. In *Proceedings of the 19th Conference on Pattern Languages of Programs*, page 7. The Hillside Group, 2012.

[11] S. Rehman and K. Mustafa. Research on software design level security vulnerabilities. *SIGSOFT Softw. Eng. Notes*, 34(6):1–5, Dec. 2009.

[12] J. Ryoo, R. Kazman, and P. Anand. Architectural analysis for security. *IEEE Security & Privacy*, (6):52–59, 2015.

[13] J. Ryoo, P. Laplante, and R. Kazman. Revising a security tactics hierarchy through decomposition, reclassification, and derivation. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pages 85–91. IEEE, 2012.

[14] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[15] J. van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.*, 17:277–306, July 2005.

[16] C. Visaggio. Session management vulnerabilities in today's web. *IEEE Security & Privacy*, (5):48–56, 2010.

[17] B. D. Win, R. Scandariato, K. Buyens, J. Grgoire, and W. Joosen. On the secure software development process: Clasp, {SDL} and touchpoints compared. *Information and Software Technology*, 51(7):1152 – 1171, 2009. Special Section: Software Engineering for Secure Systems Software Engineering for Secure Systems.

[18] Y. Wu, R. A. Gandhi, and H. Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, pages 22–28. ACM, 2010.