Architectural Analysis for Security

Jungwoo Ryoo | Pennsylvania State University
Rick Kazman | University of Hawaii and Carnegie Mellon University Software Engineering Institute
Priya Anand | Pennsylvania State University

Existing research on systems security has focused on coding, providing little insight into how to create a secure architecture. Combining architectural analysis techniques based on tactics, patterns, and vulnerabilities will achieve the best outcomes.

Security is a quality attribute that has received little attention from the software architecture community. Most of the research, methods, and tools created to address security focus on secure coding. However, it's difficult to achieve a high level of system quality by focusing only on coding. Architectural issues can overwhelm even the most meticulous coding efforts, and ignoring such issues results in systems that are hard to maintain, insecure, buggy, and so on.

In this article, we present three techniques for analyzing security that together constitute a comprehensive architectural analysis method. The benefit of these techniques is that they are scalable: they can apply to systems, or even systems of systems, and be systematically and repeatedly applied.[1]

Architectural Analysis

Architectural analysis is a structured way to analyze the design decisions made (or not made) in a software-intensive system with respect to the stakeholders' quality attribute goals such as modifiability, performance, usability, and of course, security. The rationale behind architectural analysis is that discovering design problems during coding or maintenance is too late, because addressing these problems later in the life cycle is costly, risky, and disruptive to a project. A few established architecture analysis methods exist, the most well-known being the architecture tradeoff analysis method (www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm).

Possible Approaches

We propose three novel approaches to architectural analysis for security: *vulnerability-oriented architectural analysis* (VoAA), *pattern-oriented architectural analysis* (PoAA), and *tactic-oriented architectural analysis* (ToAA).

Vulnerability-Oriented Architectural Analysis

Vulnerabilities are weaknesses in a software-intensive system that attackers exploit to breach security. Software vulnerabilities consist of coding errors and design flaws. Many software security efforts focus on finding and fixing coding errors, as demonstrated in software vulnerability repositories such as the Common Vulnerabilities and Exposures (CVE) list (http://cve.mitre.org) and the Open Web Application Security Project vulnerability list (www.owasp.org/index.php/Category:OWASP_Top_Ten_Project). Today's diverse static and dynamic code analysis tools are evidence of the strong focus on finding and remediating software coding errors. Compared to coding errors, software design flaws have received less attention from research and practitioner communities.

Vulnerabilities are an important measure of software security. All other things being equal, the more vulnerabilities a piece of software has, the worse its security is. A software architect could, in theory, use a list of known vulnerabilities, such as the CVE list, as a checklist when designing or analyzing an architecture. However, the number of vulnerabilities in the CVE list is overwhelming.

 Copublished by the IEEE Computer and Reliability Societies

There are currently more than 70,000 CVE entries, and the list is constantly growing. The smaller Common Weakness Enumeration (CWE) list (http://cwe.mitre.org) is more manageable with 940 items, but is still too large to be practical. The numbers differ between the lists because the CWE features categories of weaknesses, whereas the CVE contains their instances. Each instance is associated with real-life security incidents and vendor reports as well as vulnerability information provided by sources such as security researchers.

Another obstacle to using such resources is their granularity. Most CWE vulnerabilities are directly mapped to coding errors, which makes them incompatible with the reasoning necessary for architectural analysis. One way to overcome this granularity problem is by linking the coding vulnerabilities to a more abstract concept related to architectural design. In the case of CWE-121—stack-based buffer overflow—the closest design concept is input validation. Using this approach, an architect can ask a question like, does the architecture contain facilities to handle input validation (associated with stack-based buffer overflow) systematically and consistently? This approach's advantage is that the architect can reason about the architecture in a way that's concrete and directly related to the coding vulnerability.

> **Architectural analysis is a structured way to analyze the design decisions made (or not made) in a software-intensive system.**

For VoAA to be truly useful, a categorization of known vulnerabilities is essential to help architects effectively navigate the vulnerability list and focus on a specific vulnerability. Although the CWE list offers some categories for this purpose, its categorization mixes design and implementation concepts, rendering it inappropriate for architectural analysis. For example, nonarchitectural design vulnerabilities stemming from implementation errors, such as null pointer dereference, are included. Hence, a new way of categorizing CWEs is necessary before such an enumeration can be used for VoAA.

Finally, the CWE list's sheer number of vulnerability items is problematic. Categorization helps, but a more fundamental solution would be to reduce the number of essential vulnerabilities that an architect must analyze. As part of our VoAA technique, we propose revising the CWE hierarchy so that an architect deals with fewer vulnerability types while still being able to drill down into more detail when necessary.

Pattern-Oriented Architectural Analysis

Design patterns are proven solutions to recurring problems[2] and appear as both architectural patterns and class- or code-level patterns. Architectural patterns address systemwide concerns, such as security, performance, or availability, whereas design patterns address local concerns. For example, the Factory pattern captures how to best instantiate an object from a class without exposing the instantiation logic.[2]

Design patterns might be architectural if they're consistently adopted throughout the software. For instance, the Factory pattern could be an architectural pattern if all developers were to use it uniformly and systemwide. Consistent adoption positively affects important quality attributes such as modifiability. Therefore, patterns can inform an architectural analysis by leading an analyst to look for evidence of architectural patterns that structure the entire system or for consistent, enforced use of design patterns.

Several pattern repositories could be used to support an architectural security analysis.[3,4] We conducted a literature survey and compiled our own list of 124 known security patterns that we use to guide our analyses.

Tactic-Oriented Architectural Analysis

Tactics are design fundamentals: the atomic building blocks of architectural patterns.[5] They are realizations of design intentions. For example, the fundamental categories of security tactics are *detect attacks*, *resist attacks*, *react to attacks*, and *recover from attacks*. These categories represent the most basic design intentions for an architect attempting to address security. They are further divided into more specific tactics to form a hierarchy (see Figure 1).

Because tactics catalog design intent, ToAA begins with an analyst interviewing an architect. At this stage, the interviewer probes the architectural approaches employed in the system, rather than delving into the specifics of how these approaches are realized. ToAA is much quicker than detailed design review or code assessment and takes advantage of the architect's knowledge of the system architecture. Because tactics cover the entire spectrum of design approaches available to the architect, they can efficiently function as a design checklist.

The analyst uses the tactics in Figure 1 to form interview questions such as the following:

- Does the system support intrusion detection? An example is comparing network traffic or service request patterns within a system to a set of signatures or known patterns of malicious behavior stored in a database.

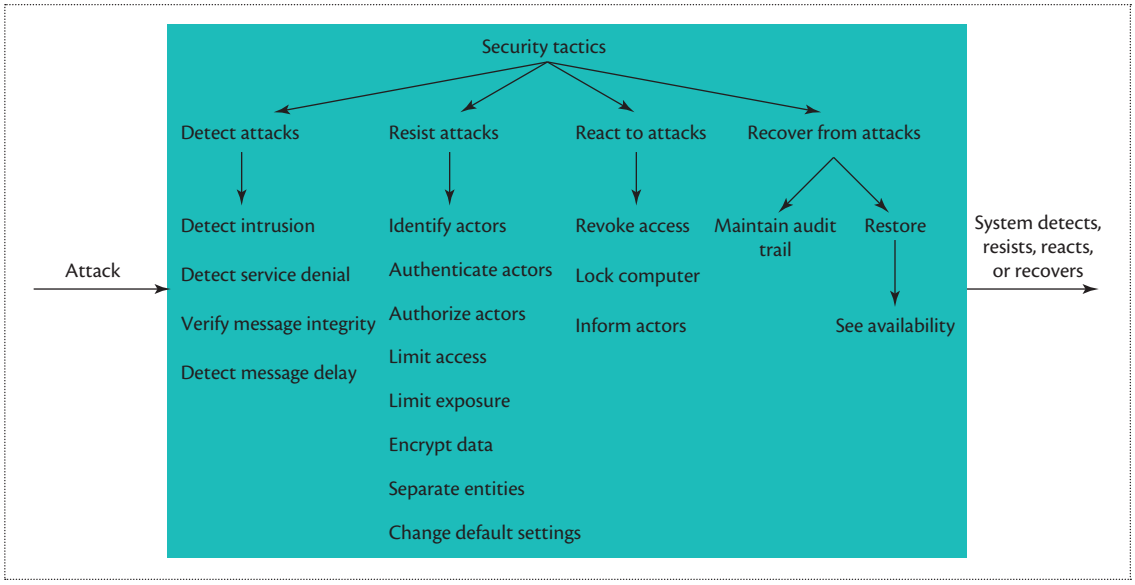

Figure 1. Security tactics. The fundamental categories of security tactics are detect attacks, resist attacks, react to attacks, and recover from attacks. These categories are further divided into more specific tactics.

- Does the system support message integrity verification? An example is the use of techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.
- Does the system support limiting exposure? An example is reducing the probability of a successful attack or restricting the amount of potential damage; for example, concealing facts about a system ("security by obscurity") or dividing and distributing critical resources ("don't put all your eggs in one basket").

For each question, the architect responds as to whether the system supports the tactic, detailing where and how the tactic is implemented and explaining any design rationale and assumptions involved in the tactic's realization. In this way, a detailed, comprehensive picture of the system's architectural approaches to security can be built in as little as one hour.

Proposed Method

Each of the approaches we described has strengths and weaknesses. VoAA offers comprehensiveness, but at the cost of usability: it's quite labor intensive because it's tied to coding concerns and there are many vulnerabilities to consider. PoAA dramatically shrinks the number of concerns to focus on—there are "only" 124 patterns in our catalog—but this is still a lot to manage. Furthermore, architects aren't always aware of the patterns being employed or how they are implemented, tailored, and combined. ToAA provides a relatively small number of concepts to consider and can be efficiently conducted in an interview, but it's relatively abstract and the connections to code might be distant.

To ameliorate each approach's weakness, we systematically combined them. Our blended method, called Architecture Analysis for Security (AAFS), consists of identifying security vulnerabilities, relating ToAA results to patterns, relating PoAA results to vulnerability categories, and analyzing and verifying source code. (For more information on related research, see the sidebar.)

Identify Security Vulnerabilities Using ToAA

An analyst interviews an architect and asks whether and how the system addresses each tactic type in Figure 1. Based on the architect's responses, the analyst ranks the tactics according to their importance and risk in the architecture. The goal is to develop a prioritized list of tactics. "Importance" combines the tactic's risk and contribution to achieving the system's business goals.

During this prioritization, the architect's input is critical: the architect is one of the few people who can authoritatively assess the degree to which each tactic has been adopted. Unlike a typical risk analysis scenario in which stakeholder input is crucial to determining what risk to address, accept, or transfer, this step doesn't focus on stakeholder-led decision making or prioritization. Rather, it focuses on capturing and presenting the system's current security state based on facts that are elicited, documented, or checked in the code base. The ranking of these tactics shows how critical each is to the system's architecture—according to the architect's understanding of the business goals—and determines the order in which they'll be analyzed for security. That

Related Research in Architectural Analysis of Security Methods

Numerous works on the architectural analysis of security exist, but they address only one or two aspects of our method. One of the Architecture Analysis for Security (AAFS) method's strengths is its comprehensiveness. AAFS addresses all aspects of a security-centric architectural analysis, including threat modeling, security pattern identification, vulnerability assessment, attack surfaces, risk analysis, and source code analysis. AAFS also provides guidance on leveraging these elements to optimize the effectiveness of a security analysis.

AAFS's customization and security focus also make it unique. Numerous general-purpose architectural analysis methods exist, each with its own goals and techniques.[1] But these methods have a wider scope than AAFS: they're concerned with all quality attributes and, hence, don't provide an in-depth architectural analysis of security.

Risk Analysis and Security Patterns

Spyros Halkidis and his colleagues proposed a method for performing a risk analysis based on security patterns.[2] Their method uses fuzzy logic and qualitatively determines how well a system is protected by its security patterns. This approach lacks AAFS's comprehensive coverage in four respects. First, it considers only seven security patterns and is therefore limited in its ability to help architects reason about security concerns and their corresponding countermeasures. Second, the risk analysis uses Microsoft's STRIDE model, which considers only six attack types (http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx). Third, its attack-based risk analysis is a less direct and fundamental path to improving software security than AAFS's tactic-, pattern-, and vulnerability-based risk analysis technique. That is, AAFS concentrates on vulnerabilities, which are a more direct and fundamental source of software security problems. Fourth, Halkidis and his colleagues don't consider tactics to identify vulnerabilities. AAFS's integration of tactics, patterns, and vulnerabilities into a single analysis framework provides more perspectives from which to analyze an architecture for security. In fact, although Halkidis and his colleagues perform a more in-depth risk analysis, AAFS subsumes their method.

Threat Modeling and Security Patterns

Eduardo Fernández explains how security patterns can be applied in various software development phases.[3] Munawar Hafiz and his colleagues developed a pattern language using the STRIDE threat model.[4] They classify security patterns according to STRIDE for easier and more systematic adoption. These works focus on pattern adoption, whereas AAFS concentrates on the assessment of system security.

Attack Surface and Security Vulnerabilities

Pratyusa Manadhata and Jeannette Wing attempted to quantify a software system's security by measuring its attack surface.[5] This approach differs from ours in that it focuses on the system's inherent properties, such as methods, channels, and data items. Attack surface measurements provide information on a system's vulnerabilities, but they don't reveal a causal relationship between architectural defects and security vulnerabilities.

References

1. D. Falessi et al., "Decision-Making Techniques for Software Architecture Design: A Comparative Survey," *ACM Computing Surveys*, vol. 43, no. 4, 2011, pp. 33:1–33:28.
2. S.T. Halkidis et al., "Architectural Risk Analysis of Software Systems Based on Security Patterns," *IEEE Trans. Dependable and Secure Computing*, vol. 5, no. 3, 2008, pp. 129–142.
3. E.B. Fernández, "A Methodology for Secure Software Design," *Proc. 2004 Int'l Conf. Software Engineering Research and Practice* (SERP 04), 2004, pp. 130–136.
4. M. Hafiz, P. Adamczyk, and R.E. Johnson, "Organizing Security Patterns," *IEEE Software*, vol. 24, no. 4, 2007, pp. 52–60.
5. P. Manadhata and J. Wing, "An Attack Surface Metric," *IEEE Trans Software Eng.* , vol. 37, no. 3. 2011, pp. 371–386.

is, the most important tactic will be the first one examined in the next phase, PoAA.

Because ToAA addresses the architectural level of abstraction, there should always be a follow-up PoAA analysis. To guide this subsequent analysis, ToAA aims to have the architect reflect on which security approaches have and haven't been implemented.

Relate ToAA Results to Patterns and Conduct PoAA

Once the riskiest tactics have been identified, the analyst, ideally in collaboration with the architect, reviews the patterns closely related to those tactics.

For example, let's assume that the first tactic to be investigated is "verify message integrity." A pattern commonly associated with this tactic is input validation, which is manifested via the intercepting validator pattern:

- The intercepting validator pattern verifies user inputs before they're used.[3] It performs filtering of all requests or user inputs according to a set of validation rules.
- Depending on the validation result, the intercepting validator will forward full, partial, or no input to the rest of the system.

Table 1. Example of pattern-oriented architectural analysis, which assesses the existence and consistent use of security patterns.

Pattern	Coverage	Priority
Intercepting validator	Partial	High
Message inspector	No	Low
Message interceptor gateway	No	Low

Table 2. Example of vulnerability-oriented architectural analysis, which yields a prioritized list of potential vulnerabilities.

CWE* no.	Description	Coverage	Priority
126	Buffer over-read	Full	High
127	Buffer under-read	No	High
785	Use of path ...	Partial	Low.

**CWE = Common Weakness Enumeration list.*

A pattern's effectiveness greatly diminishes if it's adopted by just a subset of the software. For instance, if only a portion of the software uses the client part of the intercepting validator pattern, the system as a whole will potentially suffer buffer overflow, SQL injection, and cross-site scripting (XSS) attacks. Adoption could be as simple as invoking a method to pass user input to the intercepting validator built into the architecture as a service.

An example of systemwide application of the intercepting validator pattern is an XML firewall architecture, which is used in the context of Web services and has the following security characteristics:

- Message inspector is a subcomponent of the XML firewall. It specializes in examining XML messages for their validity using mechanisms such as signatures, encryption, and tokens. It might additionally use an XML schema for input validation.
- Message interceptor gateway, another specialized subcomponent of the XML firewall, acts as a checkpoint that "encapsulates access to all target service endpoints."[4]

During PoAA, the analyst explores the evidence for systemwide adoption of security patterns. The architect is questioned about the existence or use of security patterns related to the tactics of interest. A pattern's use doesn't necessarily imply its faithful implementation, so the analyst must also question the architect about the implementation's completeness. By the end of PoAA, the analyst sufficiently understands the code to examine it during the next phase, VoAA. An example of such an assessment is shown in Table 1.

Relate PoAA Results to Vulnerability Categories and Conduct VoAA

This third phase builds on the evidence gathered in the previous phases. At this point, the analyst has a good idea of what source code to analyze but must strengthen the case by collecting evidence directly from the code. The analyst searches for weaknesses resulting from either not adopting patterns or not properly and thoroughly implementing those patterns.

For example, from the PoAA results, the analyst might conclude that the intercepting validator pattern exists in the system. But the analyst could still question the pattern integrity (how it was implemented) or coverage (how consistently it was adopted). Here, the analyst again needs guidance from the architect on where to begin examining the code. Thus, VoAA helps connect the architectural analyses begun in the ToAA and PoAA phases to the implementation.

Continuing the example, during VoAA, the analyst can expose the architect to well-known software vulnerabilities that can be addressed by the intercepting validator pattern—for example, buffer over-read (CWE 126), buffer under-read (CWE 127), and use of path manipulation function without maximum-sized buffer (CWE 785). The architect can then describe the use (or lack) of any systematic strategy, such as an architectural or design pattern, to handle these potential vulnerabilities. In addition, the architect is expected to provide evidence of the degree to which any such pattern is consistently implemented.

As with PoAA, the VoAA phase's output is a prioritized list of potential vulnerabilities. After undergoing a risk analysis exercise similar to that of the PoAA phase, the analyst describes each risk as fully, partially, or not covered. See Table 2 for an example.

Analyze Source Code Based on VoAA Results

The analysis isn't complete until the relevant source code is reviewed—or at least sampled if the amount of code is too large—to support the architect's coverage claims. The objective of this code-analysis phase is to verify the consistent, systemwide adoption of a pattern. If a pattern's adoption is partial, the software's security can't be guaranteed, which is almost as bad as not using a pattern.

Verify Code Analysis Results Using Security Testing Results and Incident Reports

In the best case scenario, security testing (static analyses, penetration tests, and so on) to objectively verify the effectiveness of VoAA activities will have already taken place. For example, if VoAA results imply that buffer under-read has no coverage and the security testing history or incident reports reveal attacks that

Table 3. Tactic-oriented architectural analysis interview with the architect of the Open Electronic Medical Record (OpenEMR) project.

Tactic	Realization of tactic (architect's response)
Detect intrusion	Use of logging.
	No way to detect specific intrusion attempts, such as SQL injection.
Verify message integrity	Partially supported by OpenEMR via standardized library function calls that sanitize user inputs.
Limit access	Implemented as part of the OpenEMR business logic.
	Use of database views and role-based access control.
Limit exposure	Not supported by OpenEMR.
	OpenEMR isn't modular; therefore, a compromise's impact can spread quickly throughout the system.

Table 4. Based on our Architectural Analysis for Security results, OpenEMR's architect identified these vulnerabilities as the project's highest priorities.

CWE no.	Description	Coverage	Priority
89	Improper neutralization of special elements used in a SQL command	Partial	High
79	Improper neutralization of input during webpage generation	Partial	High
80	Improper neutralization of script-related HTML tags in a Web page	Partial	High
87	Improper neutralization of alternate XSS syntax	No	High

exploited the vulnerabilities exposed by VoAA, it's safe to conclude that the vulnerability actually exists and that the security gap must be bridged. Therefore, although optional, the last step of our method is verifying the VoAA results using security testing or incident reports.

Although we described our method in a top-down fashion, we can also use it in a bottom-up manner. That is, security testing results or incident reports can be traced back to defects or oversights in the architectural design decisions made thus far. This traceability between code-level vulnerabilities and design defects is another important benefit of the method.

Case Study: Open Electronic Medical Record Project

Here, we describe how we used our AAFS method to analyze the security of the Open Electronic Medical Record (OpenEMR) project (www.open-emr.org). OpenEMR is an application that medical practices use to manage their daily operations, including the use of electronic medical records. OpenEMR attempts to be comprehensive: it contains features for running a medical practice, such as patient scheduling, patient portals, patient demographics, prescriptions, reporting, medical billing, and so on. The first software version was released in June 2001.

Applying ToAA, PoAA, and VoAA to OpenEMR

Our goal in this case study was to apply AAFS to OpenEMR. To make this more than just a proof-of-concept exercise, however, we wanted to validate AAFS's feasibility. We did this by first identifying OpenEMR's vulnerabilities and mapping them to the results of one or more external vulnerability assessments.

In the ToAA phase, we interviewed OpenEMR's lead architect. The architect explained that security wasn't a main concern when the OpenEMR project started and that substantial legacy code in OpenEMR has security vulnerabilities. Some of the architect's answers regarding security tactics appear in Table 3.

Note that the *verify message integrity* tactic revealed a weakness in sanitizing user inputs. The subsequent PoAA phase mapped this weakness to the intercepting validator pattern, which should have been implemented systemwide. This strategy also turned out to be one of the architect's top priorities.

There are 43 vulnerabilities associated with the intercepting validator pattern. The architect scrutinized this list and identified the vulnerabilities shown in Table 4 as the project's highest priorities, based on project history.

We first examined how OpenEMR addressed CWE 89. OpenEMR mandates the use of "bind variables" to prevent SQL injection attempts. The naive way of

doing this is to pass user input directly to a SQL statement, for example:

```
sqlStatement = "DELETE FROM immu-
nizations WHERE id = $_GET['id']";
```

Instead, the OpenEMR SQL injection countermeasure puts the potentially malicious user input into a temporary variable, the content of which is then sanitized before being bound to a placeholder (question mark in PHP) in the same SQL statement:

```
sqlState-
ment = ("DELETE FROM immuniza-
tions WHERE id = ? LIMIT 1",
array($_GET['id']));
```

> **AAFS's integration of tactics, patterns, and vulnerabilities into a single analysis framework provides more perspectives from which to analyze software security.**

This countermeasure has eight variants, which are explained on the OpenEMR wiki site (www.open-emr.org/wiki/index.php/Codebase_Security). When interviewed, one of the OpenEMR architects stated that the SQL injection countermeasures were imperfect and reactive, not proactive. According to this architect, the existence of eight countermeasure variants indicates that additional countermeasures were added as new vulnerabilities were discovered. The architect estimated that 20 percent of the code doesn't use the recommended countermeasures. This incomplete CWE 89 coverage indicates that, despite the project's efforts to fix them, SQL injection vulnerabilities still exist in OpenEMR.

We also explored the SQL injection countermeasures implemented in various software frameworks. PHP provides a built-in sanitization feature (www.php.net/manual/en/filter.filters.sanitize.php), which OpenEMR doesn't use as of this writing. A newly developed PHP security framework (PHPSEC; www.owasp.org/index.php/OWASP_PHP_Security_Project; http://phpseclib.com) is also available. Zend is another PHP framework featuring security countermeasures (http://framework.zend.com). Currently, OpenEMR doesn't use any of these frameworks.

We compared these framework countermeasures with what is available natively through the OpenEMR project. This comparison provided insight on how comprehensive or limited the OpenEMR SQL injection countermeasures were. All these framework-based countermeasures revolve around two mechanisms: *parameterization* and *escaping*. Parameterization isolates elements of a SQL statement as independent units and verifies that each corresponds to database table components (attributes, table names, and so on). Escaping refers to adding a backslash before a special character, such as ' , to force its recognition as a regular string instead of a reserved keyword for SQL. OpenEMR adopts both parameterization and escaping, comparable to the framework-based countermeasures.

Finally, we looked into the effectiveness of the OpenEMR SQL injection countermeasures; that is, how faithfully these countermeasures are enforced in the project. Countermeasure enforcement has two adoption levels. The lower level uses a secure coding standard and code inspections, which is problematic because the standard's implementation is up to individual developers. Furthermore, code inspections aren't guaranteed to catch all instances of nonconforming code. The higher and more desirable adoption level involves a systematic, architectural approach, such as the use of security frameworks. Our code reviews and interviews with the lead architect revealed that the OpenEMR project is in transition from the lower to the higher adoption level.

Based on these observations, it's probable that OpenEMR will continue to experience SQL injection vulnerabilities until more radical and architectural solutions are introduced.

We also examined CWE 87 (XSS). Unlike the case of SQL injection vulnerabilities, the OpenEMR team has taken virtually no systematic action against XSS vulnerability. Some XSS vulnerabilities have been patched on a case-by-case basis, but no secure coding standard has been introduced and no systematic code inspection specifically for XSS has been conducted. Therefore, we believe that the extent of XSS vulnerabilities is much more serious than that of SQL injection.

Validation

Various parties have conducted vulnerability assessments of OpenEMR.[6] These reports have a recurring theme: lack of or insufficient input validation. This result is consistent with our architectural analysis. More specifically, SQL injection and XSS vulnerabilities seem to be among the most difficult problems to solve.

We also conducted our own vulnerability assessments. In particular, we used two versions of OpenEMR, 3.1.0 and 4.1.2, to compare the changes in the number of SQL injection and XSS vulnerabilities. As other vulnerability assessment reports have revealed, we discovered that SQL injection vulnerabilities still exist in OpenEMR version 4.1.2: there were 96 SQL

injection vulnerabilities reported for version 3.1.0, and 12 for version 4.1.2. In addition, 65 XSS vulnerabilities were reported in version 3.1.0, and 61 in version 4.1.2.

The good news is that the number of SQL injection vulnerabilities decreased significantly, from 96 to 12. Considering the increased LOC in version 4.1.2, this change becomes even more obvious. In version 3.1.0, the vulnerability density was 4.57 per 10,000 LOC. In version 4.1.2, it dropped significantly to 0.23. The density of XSS vulnerabilities also dropped from 3.1 to 1.14.

Thus, although the OpenEMR team's SQL injection intervention is working, the effort remains insufficient. In the case of XSS vulnerabilities, the situation is worse. The raw number of XSS vulnerabilities has stayed relatively static, and although the density dropped, it remains unacceptably high. This clearly illustrates the difference that a systematic intervention can make.

OpenEMR's incomplete and inconsistent coverage of the intercepting validator pattern demonstrates a dire need for a more systematic, systemwide architectural approach. Existing OpenEMR intercepting validator implementations rely too heavily on developer conformance to secure coding standards and code inspections that are limiting and reactive at best. A better approach would be to use a framework—either homegrown or third party—in which various intercepting validator patterns are already implemented, heavily tested, and ready to be reused by developers. We strongly recommend using security frameworks that address not only OpenEMR's intercepting validator implementation needs but also other potential security requirements.

In future work, we plan to further evaluate and fine-tune the AAFS approach through more case studies. We are also planning additional empirical studies to better establish AAFS's precision, accuracy, and efficiency. Finally, we intend to contribute to the revision of the CWE-1000: Research Concepts catalog to enhance its suitability for AAFS. ■

Acknowledgments

We thank Jung-Woo Sohn, who helped with the code analysis of OpenEMR with respect to SQL injection vulnerabilities.

References

1. R. Kazman, M. Gagliardi, and W. Wood, "Scaling Up Software Architecture Analysis," *J. Systems and Software*, vol. 85, no. 7, 2012, pp. 1511–1519.
2. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.
3. C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2012.
4. D.M. Kienzle et al., "Security Patterns Repository," version 1.0, 2006; www.scrypt.net/~celer/securitypatterns/final%20report.pdf.
5. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley Professional, 2012.
6. A. Austin, B. Smith, and L. Williams, "Towards Improved Security Criteria for Certification of Electronic Health Record Systems," *Proc. 2010 ICSE Workshop on Software Engineering in Health Care* (SEHC 10), 2010, pp. 68–73.

Jungwoo Ryoo is an associate professor of information sciences and technology at the Pennsylvania State University–Altoona. His research interests include information assurance and security, software engineering, and computer networking. Ryoo received a PhD in computer science from the University of Kansas. He's a member of IEEE and ACM and a technical editor of *IEEE Communications Magazine*. Contact him at jryoo@psu.edu.

Rick Kazman is a professor at the University of Hawaii and principal researcher at Carnegie Mellon University's Software Engineering Institute. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. Kazman received a PhD in computational linguistics from Carnegie Mellon University. He's the coauthor of several books, including *Software Architecture in Practice*, *Evaluating Software Architectures: Methods and Case Studies*, and *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Kazman is a Senior Member of IEEE. Contact him at kazman@hawaii.edu.

Priya Anand is a visiting instructor at Pennsylvania State University–Altoona and a PhD student in the College of Information Sciences and Technology at Pennsylvania State University. Her research interests include software architecture and software security. Contact her at axg36@ist.psu.edu.