A Contextual Guidance Approach to Software Security

Philipp Schugerl[1], David Walsh[1], Juergen Rilling[1], Philippe Charland[2]
[1]Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{p_schuge, da_wals, rilling}@cse.concordia.ca
[2]System of Systems Section
Defence R&D Canada – Valcartier
Québec, Canada
philippe.charland@drdc-rddc.gc.ca

Abstract

With the ongoing trend towards the globalization of software systems and their development, components in these systems might not only work together, but may end up evolving independently from each other. Modern IDEs have started to incorporate support for these highly distributed environments, by adding new collaborative features. As a result, assessing and controlling system quality (e.g., security concerns) during system evolution in these highly distributed systems become a major challenge. In this research, we introduce a unified ontological representation that integrates best security practices in a context-aware tool implementation. As part of our approach, we integrate information from traditional static source code analysis with semantic rich structural information in a unified ontological representation. We illustrate through several use cases how our approach can support the evolvability of software systems from a security quality perspective.

Keywords: Context-awareness, security concerns

1. Introduction

The constant changes in our society and in technology do not only affect our daily lives, but also software systems and the requirements these systems have to satisfy. The Internet is an enabling technology that formed the basis for a trend towards the globalization of software development through multi-site development and collaborative workspaces. Along with such technical and cultural changes, new challenges for requirements assessment arise. Components in these global systems might no longer evolve together, making the assessment, validation and control of the implementation of requirements a major challenge. Non-functional requirements (NFR), also often referred to as the qualities of a system, can be divided into two main categories: (1) Execution qualities, such as performance and usability, which are observable at run time; and (2) evolution qualities, such as testability, maintainability, extensibility, and scalability, which are embodied in the static structure of the software system. In this paper, we focus on the second category of NFRs, i.e., the evolvability of software systems. Given these large, complex and global software systems, security concerns are becoming an essential quality aspect of them and of their development processes. Most of the existing work has focused on reactive or postmortem activities that involve security fixes either at the end of the deployment phase or after delivering the systems. More recently, security patterns have been promoted to integrate security concerns early on during the software development process. Security patterns, similar to other design patterns, represent existing (security) solutions in a structured way and allow knowledge from these already existing solutions to be reused. In [1], security patterns have been classified in two main categories: (1) *Available System* patterns, which facilitate the construction of systems in order to provide predictable secure access to services and resources they offer to users and (2) *Protected System* patterns that facilitate the construction of systems to protect valuable resources.

Addressing security concerns in evolving software systems is an essential and highly specialized task, which often requires maintainers to deal with inconsistent or even non-existing software artifacts (e.g., documentation, bug reports). This lack of requirements and artifact traceability results in situations where addressing security concerns becomes an inherently difficult task. In this paper, we present a novel approach that applies reusable security and dependability solutions stored in security patterns and rules for Java programs. The objective is to provide an environment that guides maintainers in identifying security concerns in software systems based on their current maintenance task and available resource context. Our approach combines a more traditional rule-based static analysis with a higher level, semantic rich ontological representation of the source code to support the identification of security concerns at various abstraction levels.

This research is a continuation of our previous work on semantic modeling of software artifacts and process modeling [2]. We extend our existing semantic

software engineering environment (SE-Advisor) to provide maintainers with both context-aware and semantic rich information to guide them while addressing various security concerns.

Relevant background and use cases illustrating the motivation of our research are presented in Section 2 and 3. Section 4 introduces our approach to context-awareness followed by implementation details of our SE-Advisor in Section 5. Application examples and threats to validity are presented respectively in Section 6 and 7. Related work (Section 8) and conclusions (Section 9) complete the article.

2. Related Work

A key challenge in system evolution is to ensure that a system is secure. This does not only increase stakeholders trust in the system. It is also perceived as an indicator of higher quality.

2.1 Security Patterns and Concerns

In order to address current and future security concerns, expertise related to such threats and their potential solutions have to be formulated and made available to maintainers. One approach is the creation of libraries that contain rules and patterns describing security concerns and their potential solutions. Traditional security patterns address network and hardware related issues. For example, one of the purposes of the RAID storage architecture is replication to ensure the possible file recovery in case of failure. More recently, security patterns have emerged as a mean to thwart attacks against applications by malicious users. The concern is especially important in the context of online applications whose unrestricted availability makes eventual vulnerabilities exploitable by anyone. These patterns describe known security concerns and provide proven solutions for addressing security risks. There exists a significant body of work on describing [3] and modeling [4] security patterns. Security patterns, which are specific to an application context, have been discussed for Web applications [5]. New security vulnerabilities are constantly exposed which lead to additional security patterns being added to these collections. However, knowledge related to security concerns often end up being disconnected from each other, making the analysis and correction of security concerns an even more challenging task.

Another approach for gathering security related information is by assessing the implementation of well-known and widely accepted best coding practices [6]. Code reviews through static analysis tools are the easiest and most straightforward way of identifying potential security concerns. Basic lexical analysis, the approach taken by static analysis tools such as ITS4 and Flawfinder, is achieved by tokenizing the source files and then matching the resulting token stream against a library of vulnerable constructs. By building an abstract syntax tree (AST) from source code, such tools can take into account the basic semantics of the program under scrutiny. Consequently, the implementation of good practices can be evaluated by analyzing the AST associated with a source code segment. Several highly specialized tools for static analysis and code reviews exist to identify potential security concerns [7].

2.2 Ontologies

Ontologies have been widely used in computer science to formally define domains of discourse. They can be described as a conceptualization of explicit information [8] that consists of properties, concepts (also often referred as classes), and relations between them. In contrast to databases, ontologies allow to work with incomplete knowledge, due to their support for an open world assumption. Additionally, their formal representation allows for the use of ontological reasoning services and easy extensibility [8]. Ontologies are an important part of knowledge modeling and sharing, by acting as a common language [8]. This becomes imminent in the context of semantic web technologies, of which ontologies are a fundamental building block. Description logics (DL) is used to define an ontological model which does not only provide a more precise and expressive representation than traditional data semantics, but also provides the basis for automated reasoning support. For a more detailed coverage of DL and reasoning services, we refer the reader to [8].

3. Motivation and Challenges

Before describing our approach in detail, we analyze some requirements and challenges specific to the problem domain. In order to be able to address security concerns, software maintainers have to be knowledgeable about existing security concerns and their tested, often even certified and readily available solutions. To support maintainers in locating and analyzing security concerns, existing knowledge from distributed components and resources has first to be integrated and semantically modeled. Such a representation is the basic requirement to enable the integration of software engineers' knowledge, relevant resources, and processes. What follows are three

typical use cases describing potential application areas of our approach. We will revisit these use cases throughout the article to illustrate how our approach can addresses the identified requirements.

Use Case #1: *Proactive Knowledge Dissemination.* Maintenance activities are often performed across organizational boundaries with knowledge and expertise being distributed across resources. In order to avoid the creation of information silos in these heterogeneous development environments, knowledge dissemination among stakeholders becomes an essential part of software development. From a programmer perspective, locating and extracting relevant knowledge and resources become a major challenge. In order to support maintainers in their current work context, new approaches for the dissemination of existing knowledge and resources relevant to their current context have to be made readily available as part of their working environment.

Use Case #2: *Preventative Maintenance.* Software evolution is not limited to postmortem fixes of current security flaws. In order to improve the maintainability and reduce the number of potential security concerns, preventative maintenance activities, similar to refactoring in traditional OO programming, should be applied to improve the structural quality of a system. Given a common and semantic rich representation of both source code and security patterns, automated tool support can be provided to locate potential areas for the restructuring and refactoring of potential security concerns.

Use Case #3: *Establishing Trustworthiness.* The concept of activity is at the core of any software process model. From a software process perspective, it is not only important to determine which activities need to be performed, but also establish procedures on how to accomplish them. Establishing trustworthiness, for example, requires maintainers to apply organization and application domain specific processes and activities to document that an application meets or exceeds the expected quality.

4. Context-Awareness

Schilit et al. [9] introduced the term context-awareness in the domain of ubiquitous computing. In our research, we consider a process to be context-aware if it supports guidance, which is context sensitive to the user's state, while performing a specific development/maintenance task. In our approach, knowledge is built from software engineering artifacts (external sensors) as well as from developers and process domain specific inputs.

Given these available resources, we further refine the notion of context-awareness as the current development status. From a programmer perspective, it can be expressed by the 5 w's: *Who* is doing *what*, *when*, *where*, and *why*? At the core of our context-aware software engineering environment is a software engineering ontology. The ontology is responsible for modeling knowledge in a formal, uniform, and semantic rich representation that allows the use of ontological reasoning services. In order to support context-awareness, we extended individual sub-ontologies through new concepts, properties, and relations to allow their integration (through shared concepts and ontology alignment) and the use of ontological reasoners. Given such an enriched unified ontological representation, consistency checks and advanced knowledge retrieval using the provided ontological reasoners can be performed to generate context-aware advice.

5. SE-Advisor (Use Case #1)

In this section, we describe the system architecture and implementation aspects of our SE-Advisor. The SE-Advisor is a continuation of our previous work [2], which was extended to provide context-aware guidance for software engineers while addressing security concerns. Within the SE-Advisor, knowledge integration and sharing is performed through a unified ontological representation. A more detailed discussion can be found in [2]. The ontological knowledge base (KB) itself is hosted as a central repository on a server, with persistent storage being provided by a RDBMS.

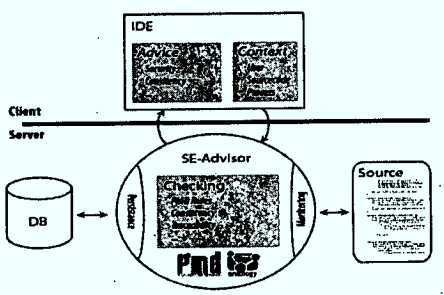

Figure 1. SE-Advisory system overview

Given the highly distributed environment in which our SE-Advisor will have to be deployed, we adopted a client-server architecture in which the clients communicate over a network with the advice server (Figure 1). The client side provides the server with the current context in which the user is working. The server processes the context information by executing parameterized ontological security queries based on the provided user context information. Inference services

are used (if applicable) to establish context-aware advice. This process is repeated for every change in the context state. It should be noted that the granularity of what corresponds to a context state change is administered as part of the server settings.

For the static source code analysis, we integrated PMD [10] within our SE-Advisor. PMD is an open source Java source code analyzer that can find potential problems such as bugs, dead code, design problems, etc. through the use of rules which are executed against an abstract syntax tree (AST).

The source code analysis is performed on every change (corresponding to a commit within a concurrent versioning system). Analysis results are cached within the ontology to optimize performance.

The SE-Advisor prototype was developed within Eclipse, an open source Java IDE that is extensible through plug-ins. As part of our research project, we developed several plug-ins to enable our context-aware security advising system. Contextual information describing the current user state is maintained through three plug-ins: (1) A user plug-in that requires the user to login to the system to identify and specify the user being monitored; (2) A source code plug-in which interacts with the Eclipse source code editor in order to discover the current source code fragment being worked on; and (3) A process plug-in that establishes the process context, by identifying the current process and process-step the user is working on. A fourth plug-in displays resulting contextual advices as part of an Eclipse view. Context changes are reflected by automatically updating advices in the Eclipse view *(use case #1)*.

6. Security and Context-Awareness

Software evolution is concerned with addressing constant changes in functional and non-functional requirements, in order to ensure that the software meets the current and future overall quality objectives of the system. In the literature, software quality and security attributes are often tightly linked, with security being considered a subset of quality. Security attributes can be addressed at various abstraction levels, including structural or source code level. At the structural level, security patterns can be applied in a given context, where a suitable pattern context is identified and the application of the pattern will lead to an improvement in the quality of the system. At the source code level, security rules can be applied to ensure that a set of static guidelines are followed in order to avoid pitfalls, or anti-patterns.

6.1 The Intercepting Validator Security Pattern (Use Case #2)

The *Intercepting Validator* (IV) security pattern is an example of structural analysis that can lead to the refactoring of an existing system to improve its evolvability (preventative maintenance - use case #2).

For most dynamic Web applications, one important security criterion is the prevention of malicious data injection. If not dealt with properly, it may result in exploitable system vulnerabilities such as cross-site scripting and SQL injections. In [11] the IV pattern, which is a specialization of the *Intercepting Filter* pattern [12], is used in an attempt to thwart this type of vulnerability. In short, *filters* are components (inserted between a client and the resources it requires) to screen requests and process them in various ways: redirection, data obfuscation, etc. In order to be processed by the server program, the request data is embedded in software objects. These objects are said to be tainted [13] by untrusted user input. The IV component is responsible for validating *Tainted Objects* to ensure that they do not contain malicious data. Moreover, all tainted objects of a server program must go through its central validating mechanism. This is one of the main characteristics of the IV pattern.

We enriched the IV pattern to become a component, by adding two new responsibilities. First, alerts must be raised when it is determined that objects have been tainted by malicious data. Alerts can take different forms such as log entries and the throwing of exceptions. Second, if the tainted object contains valid information, it must be allowed to be forwarded to the next *Intercepting Filter* of the chain.

Figure 2 shows a high-level ontological model illustrating the concepts described above. After being populated and analyzed by a reasoner, this ontology will be queried in order to answer semantic level questions such as:

- Which components might constitute *IVs*?
- Is every *Tainted Object* validated by an *IV*?
- Do *IV* components raise proper exceptions?
- Do *IV* components properly log attack data?
- Is the *IV* pattern correctly implemented?

Answers to these questions will provide valuable information, helping maintenance developers performing security improvement refactoring tasks. For instance, components that need change in the context of security oriented refactoring can be easily identified. In addition, once completed, the changes can be controlled by regularly executing a predefined set of queries to ensure, for example, the centralization of the *Tainted Object* validation (use case #2).

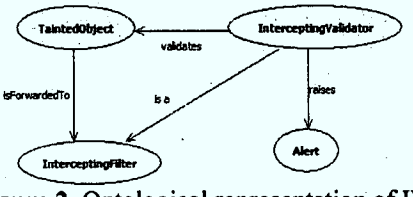

Figure 2. Ontological representation of IV

6.2 Memory Leaks (Use Case #3)

Memory leaks have been shown as one of the most common safety hazards, mainly because they can cause software to crash. Java distinguishes itself from other languages such as C++, by providing built-in memory management, which frees the programmer of explicitly dealing with memory allocation and deallocation. The Java Virtual Machine specification includes automated garbage collection to free unused memory areas. This might provide developers with a false sense of freedom, i.e., that they can code without having to focus on memory management. However, it is still possible for memory leaks to occur in Java. In particular, a Java anti-pattern called Leak Collection has been documented in [14]. It consists of static *Collections* class members on which no or too few remove operations are performed. Being static, these collections are instantiated the first time the corresponding class is loaded. From that point on, they hold references to objects for as long as the application is running, unless they are explicitly removed. In cases such as Web applications, collections might not be freed up for a long time. In this context, static collection can be seen as risks for potential memory leaks and therefore, should be further analyzed.

Considering the cost and difficulty of finding and removing memory leaks, tool support is required to help identify potential sources of memory leakage. In our approach, we integrate PMD [10], a static source code analysis tool that parses all classes within a Java application and analyzes the generated AST. We then created rules implemented either in Java or as XPath queries to analyze the parse tree.

We have applied the *Memory Leak* rule (Figure 3) on the code of a popular open source J2EE application server, JBoss. The resulting violation information is detailed enough to provide developers with quick feedback as to which part of the code to investigate for potential memory leaks. A combination of rules and security patterns should be applied as part of any code review and maintenance processes to validate the quality of the code and structure from a security perspective and maintain trust in the modified application (use case #3).

```
public class MemoryLeakRule extends AbstractRule {
    @Override
    public Object visit
        (ASTFieldDeclaration field, Object data)
    {
        if (field !=null && field.isStatic())
        {
            ASTType type = (ASTType)
            field.getFirstChildOfType
                (ASTType.class);
            if (type != null)
            {
                ASTReferenceType refType =
                    (ASTReferenceType)
                    type.getFirstChildOfType
                    (ASTReferenceType.class);
                if (refType != null)
                {
                    ASTClassOrInterfaceType coiType =
                        (ASTClassOrInterfaceType)
                        refType.getFirstChildOfType
                        (ASTClassOrInterfaceType.class);
                    if (coiType != null)
                    {
                        if (coiType.getImage() != null
                            && CollectionUtil.isCollectionType
                            (coiType.getImage(), true))
                        {
                            addViolation(data, field);
                        }
                    }
                }
            }
        }
        return super.visit(field, data);
    }
}
```

Figure 3. PMD rule for leaking collection anti-pattern

7. Threats to Validity

Knowledge modeling. Formal semantics provide a means of representing and ensuring some consistency in modeling knowledge. However, they still do not guarantee that either sufficient or the right information is captured. In our approach, we do not claim to capture all domain specific knowledge. Instead, we argue that by using an ontological model for the knowledge representation in our context-aware SE-Advisor environment, we support the modeling of incomplete and often inconsistent knowledge found in software artifacts. In spite of these modeling techniques, there will always remain a gap between the actual and modeled knowledge.

Security concerns. Given the existence of many, often highly specialized analysis (both static and dynamic) tools to locate security concerns, we see our approach as complementary to them. Our goal is not to replace these tools. Rather, we focus on: (1) the integration of knowledge resources from existing tools (e.g., PMD) to enrich our ontological KB. (2) Provide a common unified and semantic rich representation to trace security concerns and security patterns across various abstraction levels and artifacts. However, artifacts might often not be available or consistent. An ontological representation can therefore provide us with flexibility to work and apply reasoning based on this open world assumption. (3) Apply ontologies to

model security concerns and investigate the tradeoff between precision and the need for richer semantic representations. Given limitations of current ontological modeling and inference services with respect to expressiveness and scalability, further analysis is needed to evaluate their applicability to model security concerns at different abstraction levels.

Context-awareness. In order to provide maintainers with security concerns and locate potential refactorings (security patterns), it is crucial that knowledge specific to the user context is captured and made available. Our approach provides such flexibility with regards to defining new conditions and constraints applicable during knowledge exploration and therefore, within the advices provided by our system. One remaining main challenge is that concepts might not always be clearly identifiable in a given context. For example, "experienced user" is fuzzy in terms of the interpretation as to what constitutes an experienced user.

8. Related Work

Most of the existing work on context-aware software development systems, e.g. [15], has focused on defining what constitutes a particular context and formalizing the services provided by such environments [15]. The development of environments that hide external resources from end users by allowing them to be immersed without having direct exposure to the sensor data was presented in [15].

Existing work on dynamic analysis tools to detect security concerns in software systems [13] has mainly focused on the reactive approach to post-mortem security concerns, in order to fix security flaws. On the other hand, static analysis has been applied successfully to improve software quality and security without the need for collecting and analyzing behavioural information. Also, instead of focusing only on the validation of user inputs, the use of design patterns is preferable. For instance, such techniques do not guarantee how software reacts when an attack is detected. Given our unified representation, we further combine semantic rich structural (pattern level) and source code level analysis (rules) to provide contextual support in locating and analyzing security concerns.

9. Conclusions and Future Work

In this paper, we presented a novel approach that integrates resources and knowledge related to security concerns in a common ontological representation. The ontological representation does not only support the integration of knowledge resources at various abstraction and semantic levels, it also provides the foundation for developing our context-aware SE-Advisor tool.

As part of our future work, we plan to enrich our existing ontological model with additional artifacts and security concerns, as well as further evaluate its applicability.

Acknowledgement:
This research was partially funded by DRDC Valcartier (contract no. W7701-081745/001/QCV).

References

[1] L. Brown, et al., "The authenticator pattern," *Proc. of the 6th Conf. on Pattern Languages of Programs*, 1999.

[2] J. Rilling, et al., "Beyond Information Silos - An Omnipresent Approach to Software Evolution," Int. J. of Semantic Computing, Special Issue on Ambient Semantic Computing, In Press.

[3] M. Schumacher, et al., *Security Patterns: Integrating Security and Systems Engineering*, Wiley, 2006.

[4] A. Rodriguez, E. Fernandez-Medina, and M. Piattini, "An MDA Approach to Develop Secure Business Processes through a UML 2.0 Extension," J. of Computer Systems, Science and Engineering, vol. 22, no. 5, Sept. 2007, pp. 307-319

[5] C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2005.

[6] C. Lai, "Java Insecurity: Accounting for Subtleties that Can Compromise Code," *IEEE Software*, vol. 25, no. 1, Jan./Feb. 2008, pp. 13-19.

[7] *Software Quality and Security Analysis | Coverity*, Apr. 2009; http://www.coverity.com/.

[8] F. Baader, et al., *The Description Logic Handbook: Theory, Implementation and Applications*, Cambridge Univ. Press, 2003.

[9] *OWL Web Ontology Language Reference*, Apr. 2009; http:// www.w3.org/ TR/owl-ref.

[10] *PMD*, Apr. 2009; http://pmd.sourceforge.net/.

[11] B. Blakley and C. Heath, "Security Design Patterns Technical Guide - Version 1," The Open Group, 2004.

[12] *Design Patterns: Intercepting Filter*, Apr. 2009; http:// java.sun.com/blueprints/patterns/InterceptingFilter.html.

[13] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," *Proc. of the 21st Ann. Computer Security Applications Conf.*, Dec. 2005, pp. 303-311.

[14] W. Crawford and J. Kaplan, *J2EE Design Patterns*, O'Reilly, 2003.

[15] E.H.L Aarts and J.L. Encarnação, *True Visions: The Emergence of Ambient Intelligence*, Springer, 2006.