Security vulnerabilities detection and protection using Eclipse

Marco Guarnieri[1], Paul El Khoury[2] and Gabriel Serme[3]

[1] Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
0guarnieri.marco0@gmail.com
[2] SAP AG, Deutschland
paul.el.khoury@sap.com,
[3] SAP Labs and Eurecom, France
gabriel.serme@sap.com

Abstract. After a decade of existence, still, Cross-site scripting, SQL Injection and other of Input validation associated security vulnerabilities can cause severe damage once exploited. To analyze this fact, [14] conducted an empirical study, while OWASP and SANS defined their respective risk-based approaches. Taking these results into consideration, three deficiencies can be highlighted: a lack of up skilling developers, a high ratio of false positive findings in security code scanners and an erroneous planning of security corrections. In this paper, we present how using the Eclipse platform and the JDT compiler, a proper tooling can be provided to overcome these deficiencies. We present a static analyzer that assists developers to report these security vulnerabilities. We show as well how we integrate an Aspect Oriented tool for semi-automated correction of these findings. Both tools are designed within an architecture that is monitored by security experts and particularly adequate for agile development.

1 Introduction

Fixing security vulnerabilities before shipment can no more be considered optional. Most of the reported security vulnerabilities are leftovers forgotten by developers, thought to be benign code. Such kind of mistakes can survive unaudited for years until they end up exploited by hackers.

While computer security is primarily a matter of secure design and architecture, it is also known that even with best designed architectures, security bugs will still show up due to poor implementation. Different techniques have been developed to identify security bugs along the software development lifecycle, from code review to penetration testing. In this paper we focus on static code analysis, an automated approach to perform code review. This technique analyzes the source code and/or binary code without executing it and identifies anti-patterns that leads to security bugs. Modern static analysis tools, similarly to compilers, build an Abstract Syntax Tree - a tree representation of the abstract syntactic structure - from the source code and analyze it. Static analysis can report

security bugs even when scanning small pieces of code. Another family of code scanners is based on dynamic analysis techniques that acquire information at runtime. Unlike static analysis, dynamic analysis requires a running executable code. Static analysis scans all the source code while dynamic analysis can verify certain use cases being executed. The major drawback of static analysis is that it can report both false positives and false negatives. The former detects a security vulnerability that is not truly a security vulnerability, while the latter means that it misses to report certain security vulnerabilities. Having false negatives is highly dangerous as it gives one sensation of protection while vulnerability is present and can be exploited, whereas having false positives primarily slows down the static analysis process.

This paper focuses on security vulnerabilities caused by missing input validation, and presents how static analysis with semi-automatic fixing can help developers to make more secure software. More in detail we focus on three vulnerabilities caused by missing input validation, or mis-validation of the input : SQL Injection, Cross Site Scripting (also called XSS) and Path Traversal. Input validation refers to the process of validating all the input to an application before using it. The paper is structured as follows : Section 2 presents an overview of how static analysis can be integrated into an agile software development lifecycle. Section 3 gives an overview of the architecture of the static analysis tool, while Section 4 gives some explanation of how our static analysis process works. Section 5 presents how semi-automated correction of detected security issues can improve developers work. Section 6 presents related work and we finish with Section 7 that presents our conclusions and ideas for future works.

2 Static analysis into the Secure Development Lifecycle

Agile approaches to software development require that the code is refactored, reviewed and tested at each iteration of the development lifecycle. While unit testing can be used to check if functional requirements are fulfilled during iterations, checking emerging properties of software such as security or safety is more difficult. Several approaches were presented in order to integrate security into agile development lifecycle. As example the Agile version of the Microsoft Security Development Life Cycle [15], i.e. A-SDL, presents static analysis tools as means to fulfill certain security requirements. Those security requirements are expected to be checked during every Sprint of Scrum project. Without automated tools, the verification would not scale to be accomplished in every Sprint. The current static analysis tools, mostly separated from the development environments, require in most cases deep knowledge of security vulnerabilities to distinguish false positives from true positives. Typically security experts are the dedicated team that analyse projects for all of the company. An adoption of such an agile security development approach would not be feasible in such settings as it would be against Scrum principles.

Even when static analysis tools discover security vulnerabilities, determining which of these reported vulnerabilities are false positives and which are true

positives, in addition which from the true positives can actually be exploitable, required deep knowledge in the code being scanned. From a planning perspective, a conflict can be highlighted between the need of certain security expertise to analyse the reported results and the expertise of the product development. This expertise mismatch would cause a considerable waste of resources since the teams expected to verify security vulnerabilities, i.e. the code analysis team, would be looking into leftovers, possibly vulnerabilities, without knowing the expected behaviour and especially external interactions.
Maintaining the separation of roles between the security experts performing the code scanning and the team members developing the application raise a critical complication, typically, from a time perspective, due to the human interaction between security experts and developers. If such an approach would have to scale to what most of the agile approaches describe, the amount of iteration between developers and experts would need to be reduced. That could be reduced by upskilling the developers and reducing the interaction between them and the security experts for the analysis of the security scans of the project.
While a centralized team for the static analysis that analyse full projects at the end of the security development lifecycle is useful, including static analysis into an agile approach is a way to improve the efficiency and to reduce the cost of the static analysis process. A deep analysis has been conducted in [11] representing error introduction, detection and repair costs along the software lifecycle. Whereas issues are introduced early in the lifecycle, from conceptual design with a peak at the programming phase, they are detected from unit testing to operational phase. The cost to repair errors prior to testing is rather low. But it significantly increases over the last phases with a maximum peak during system operation [1].

This work aims to provide to each developer a simple way to do daily static analysis on his code. That would be properly achieved by providing a centralized architecture that allows the security experts to assist the developers in any of the findings. Typically that would include verifying a false positive and adjusting the code scanner test cases, or assisting in reviewing the solutions for the fixes. This approach has some advantages over the approach in which the static analysis stays only at the end:

1. The development group has the expertise of the context with which the project works.
2. Security experts, in charge of the code scanning, can interact with the development team on a case by case basis to fine tune the code scanner test cases and algorithm.
3. Fixing the bug into the development phase requires less time since the findings and the corrections are undertaken by the development team that wrote the code.
4. Solving security issues into the development phase can reduce the number of issues that the security experts should analyse, in this way we could reduce the costs.

3 Architecture

The previous section explains the advantages acquired by using our approach in an agile and decentralized static analysis process early in the software development lifecycle. It raises security awareness for the developers at the time of development and thus reduces cost expected for the after development fixes. In order to take advantages of static analysis support into daily work, we found that our prototype must fulfill the following requirements:

1. It must be user friendly especially for users that are not security experts.
2. It must be integrated into the developers' daily development environment, to maximize adoption and avoid additional steps to run the tool.
3. It must run with a reduced number of lines of code (as expected to be daily scans), but also must scale to large amount of code.
4. It must have access to a security knowledge database, created and maintained by security experts. Developers can help build up the knowledge database by reporting their experience about classes, methods and packages. Generally, developers can tag such element as trusted or not.
5. It must support developers in correcting issues. Also, the prototype can educate developers to understand why this error happen and what are the steps to mitigate the error and avoid it in future development.

Figure 1 represents the architecture of our prototype. First of all, we consider two main stakeholders involved in the configuration and usage of the prototype. Security experts regroup different profile whose role is to provide and configure Knowledge Database in order to avoid false positives and negatives. They have three main tasks. First, they update the Knowledge Base, adding to it classes or methods that can be considered as trusted for one or more vulnerabilities. Second, they are in charge of running the static analysis at the end of the Software Development Lifecycle using standard commercial tools. If the results of the final static analysis highlight some false negatives, vulnerabilities that our tool did not catch before, security experts must modify the Knowledge Base accordingly. In this way we can have a double check on the results of static analysis and on the accuracy of the trustiness definitions. Third, they receive feedback from developers on possible trusted objects for one or more security vulnerabilities; they must analyse them more in detail and, if these objects are really trusted they tag them as trusted into the Knowledge Base. The second role is the developer, interacting directly with the static analysis engine to verify vulnerabilities in application, code and libraries under its responsibility.

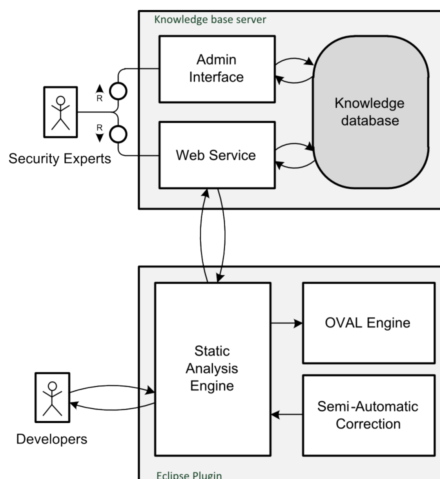

Fig. 1. Architecture

The knowledge base is shared among developers that can enhance it over time, by reporting feedbacks to security experts. It contains all the security knowledge about trustiness : objects that don't introduce security issues into the code. Security experts maintain and keep under control the definitions used by all developers in an easy way using web application or Web-Services. In this way the code scanner testing rules will be harmonized for all the application or even the company. The knowledge base allows developers to run static analysis. In the context of this paper, we are interested in the prototype that allows daily scan for developers. It is a plugin integrated into the Eclipse IDE that developers use to write code. It uses the Abstract Syntax Tree generated by JDT compiler to simplify the static analysis process, and it accesses the Knowledge Database via Web-Services. The Eclipse plugin is made of different components, that leverage decentralized approach for static analysis :

Static analysis engine It is the main component of our prototype and provide developers with a static analyser that detects security issues related to input validation problems. We will explain in detail our static analysis process in Section 4.

Security Requirements evaluator Coding standards and guidelines can help developers writing more secure code, also if they are not considered as silver bullet solution for the security problem. In the state of the art a lot of different standards exist, both from research and from software companies. We focus on the *Common Weaknesses Enumeration*, developed by *MITRE*[1], which is a list of software weaknesses. This component implements an engine that checks if a small set of these weaknesses are present into the code. Instead of creating a new policy language we choose *OVAL* [4], also from *MITRE*, as a way to express the weaknesses as rules to be fed our code scanner prototype. The existing tools for *OVAL* language aren't focused at source code level and thus we defined an extension of the language to support programming concepts such as methods, fields, variables and so on and an engine that evaluate our policies.

Semi-Automatic correction This component guide developers through the correct and semi-automatic correction of vulnerabilities previously detected. It uses information from the static analysis engine to know what vulnerabilities have to be corrected. Then it requires, through the plugin, inputs from the developer to extract knowledge about the context. These steps allow to gather places in the code where to inject security correction, and the concrete mechanisms to be called. The security correction injection uses Aspect-Oriented Programming (AOP) paradigm that we further explain in Section 5.

4 Static analysis process

Our approach is a static information flow analysis that does not consider context sensitivity. Clearly our approach for the code scanner is expected, theoretically,

[1] MITRE Corporation - http://www.mitre.org

to be less performing than some of the related work but it showed comprehensive results in practice. As already explained in Section 2, our static analysis tool does not have the goal to substitute standard static analysis but only improve it and integrate it into daily development lifecycle.

We define an *input* as a data flow from any external class, method or parameter into the code being programmed. So an *untrusted input* can be defined as an input that we don't know if it was validated for one or more security vulnerabilities, while a *trusted input* can be defined as an incoming input from a standardized implementation or from an unknown class or package that has been tagged as validate for one or more security vulnerabilities. We define as *entry point* any point into the source code where an *untrusted input* enters to the program being scanned. In an analogous way we define as *output* any data flow that goes from the code being programmed into external objects or method invocations. We divide *output* in *untrusted output*, that is an output that we don't know if it will be validated for one or more security vulnerabilities from the receiver, and in *trusted output*, that is an outcoming output to a standardize implementation or to an unknown class or package that has been tagged as validate for one or more security vulnerabilities, and finally we define *exit point* any point into the source code where an *untrusted output* goes out of our program. A *trusted object* is a class or a method that can sanitize all the information flow from an *entry point* to an *exit point* for one or more security vulnerabilities.

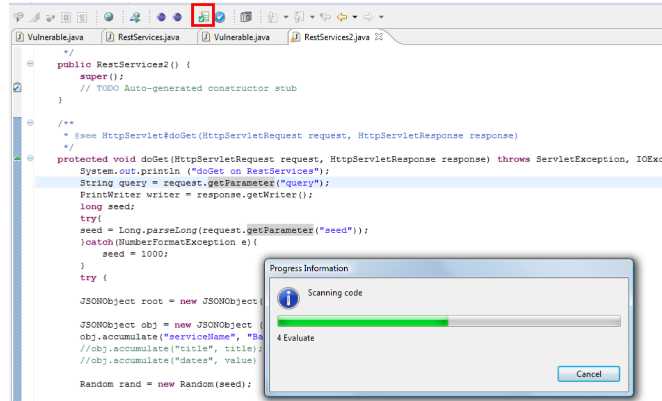

Fig. 2. Code Analysis phase

The problem of identifying security vulnerabilities caused by errors in input validation can be translated to finding an information flow connecting an *entry point* and an *exit point* that does not use a *trusted object* for the considered vulnerabilities. Our approach rely on our *trusted object* definition, that impacts the detection accuracy. We implemented the trustiness definitions into the centralized knowledge base presented in the previous section. The knowledge database represents the definitions using a trusting hierarchy that follows the package hierarchy. Security experts can tag classes, packages or methods as trusted for one or more security vulnerabilities, accordingly to their analysis, feedbacks from

developers or static analysis results. Obviously defining as trusted an element into the hierarchy trust also all the elements below it (i.e. trusting a package trust all the classes and methods into it and trusting a class trust all the fields and methods in it). A trusted object can sanitize one or more security vulnerabilities (e.g. sanitization for SQL Injection [2] is different from sanitization for XSS). Thus users can tag an object as trusted for specific vulnerabilities. Using this approach let users and security experts define strong trustiness policies. It is the major contribution to bring deep knowledge of security for the success of the process.

Markers | Properties | Servers | Data Source Explorer | Snippets | Search | Security Corrections | Security Errors | Security Warnings

14 errors, 0 warnings, 0 others

Entry Point ID	Exit Point ID	Category	Code	Location	Resource	Project
11	10	Malformed_input	title	line 59	RestServices.java	Bigbro_testbench
124	138	Malformed_input	parameter2	line 21	Vulnerable.java	Bigbro_testbench
133	132	Malformed_input	parameter	line 17	Vulnerable.java	Bigbro_testbench
139	143	Malformed_input	query	line 22	Vulnerable.java	Bigbro_testbench
144	145	Malformed_input	array	line 6	CWESnippet.java	Bigbro_testbench
150	403	Cross_Site_Scripting	out.print(parameter)	line 29	Snippet.java	Bigbro_testbench
16	54	Cross_Site_Scripting	writer.print(root.toStrin...	line 91	RestServices.java	Bigbro_testbench
165	164	Malformed_input*	parameter	line 27	Snippet.java	Bigbro_testbench
170	169	Malformed_input	parameter2	line 32	Snippet.java	Bigbro_testbench
179	443	Directory_Traversal	new File(path)	line 41	Snippet.java	Bigbro_testbench
6	211	Cross_Site_Scripting	writer.print(title)	line 92	RestServices.java	Bigbro_testbench
66	70	Malformed_input	query	line 36	RestServices2.java	Bigbro_testbench
71	285	Cross_Site_Scripting	writer.print(query)	line 69	RestServices2.java	Bigbro_testbench

Fig. 3. Code Analysis result

Defining a *trusted object* is a strong assertion as it taints a given flow as valid and free for a given vulnerability. The definition process to trust a class, a package or a method is rigorous. The object must not introduce a specific vulnerabilities into our code. This is the reason why developers report feedback and security experts decide. The experts can also analyse, manage and update the base, if the class, package or method is considered trusted. This phase allows system tuning that is related to a given organization and leads to lesser false positives while ensuring no false negatives.

The integration into Eclipse (cf Figure 2) means the developer can run it anytime to verify code validity with regards to aforementioned vulnerabilities. It also allow the usage of the Abstract Syntax Trees generated by the JDT compiler during the static analysis process. The process can be divided into the following phases:

Entry Points detection It starts with *entry points* identification from the source code. It lists all *inputs* from the program and select untrusted ones.

Construction of the information flow It builds the information flow representation from the source code, and detects all the statements that could propagate input.

Exit Points detection and evaluation It analyses the data through information flow history and detect non-sanitized flow. Upon detection of information flow connecting *entry points* and *exit points* with no *trusted objects*, it detects one or more vulnerabilities.

[2] SQL Injection is best solved using parametrized queries, but in some cases that is infeasible and therefore we try to detect those cases.

The detected vulnerabilities are mainly caused by lack of input validation, namely SQL Injection, Directory Path Traversal and Cross Site Scripting. The engine detects also a more general Malformed Input vulnerability that represent a general input that is not validated using a standard implementation. The engine can be easily extended to support new kinds of vulnerabilities caused by missing input validation, simply adding the definition of the new vulnerability to the centralized knowledge base (and, if exist, adding trusted objects that sanitize it), and creating a new class, that extends the *securitytools.Test* interface, that implements the checks to do on the result of the static analysis to detect the vulnerability.

Figure 3 shows the results of the static analysis on a given project. The project contains several files that have incorrect treatment that leads to errors from Malformed Input to Cross-Site Scripting. Regarding the file *RestServices2.java*, shown in Figure 2, the *Security Errors View* show that at line 36 the variable *query* is initialized with the parameter *query* from an *HttpServletRequest* object that is considered as *untrusted entry point* generating a Malformed Input vulnerability. The tool detects also that at line 69 the same variable is passed as argument to the method *write* of a *PrintWriter* object without proper validation, thus causing a Cross Site Scripting Vulnerability. Providing such a view allow developers to track errors on their code and quickly jump from one to another to correct them. As the correction is error prone, we though as a good idea to provide a semi-automatic correction to assist the developer in mitigate vulnerabilities.

5 Automation on security vulnerability correction

The approach we describe comprises the automatic discovery of vulnerability and weaknesses in the code. In addition, we propose to integrate a protection phase after the static analysis process. This approach distinguish our work from state of the art, and is a natural continuation to fix issues : once vulnerabilities have been discovered, we can immediately apply mechanisms to mitigate risk. In this section, we present first the concepts behind the automatic correction that leverage AOP usage, then we explain how automation is made possible thanks to the agile approach with developer involvement.

To begin with, Aspect-Oriented Programming [9] (AOP) is a paradigm to ease programming concerns that crosscut and pervade applications. The term has been coined around 1995 by a group led by Gregor Kiczales, with the goal to bring proper separation of concerns for cross cutting functionalities. The aspect concept is composed of several advice/pointcut couples. Pointcuts allow to define where (points in the source code of an application) or when (events during the execution of an application) aspects should apply modifications. Pointcuts are expressed in pointcut languages and often contain a large number of aspect-specific constructs that match specific structures of the language in which base applications are expressed, such a pattern language based on language syntax. Advices are used to define modifications an aspect may perform on the base

application. Advices are often expressed in terms of some general-purpose language with a small number of aspect-specific extensions. The process to inject aspects into the base application is called *weaving*. The main advantage using this technology is the ability to intervene in the execution without interfering with the base program code, thus facilitating maintainability.

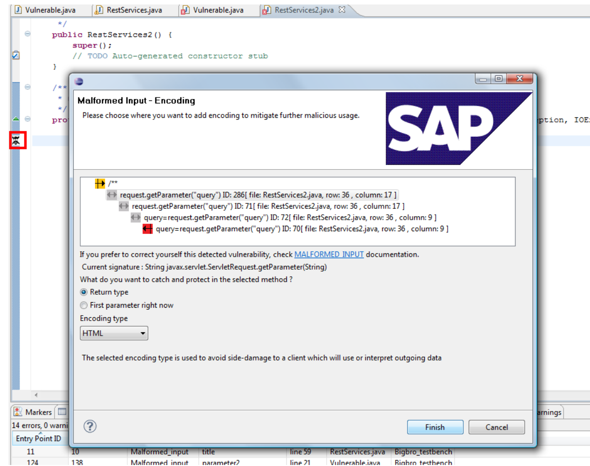

Fig. 4. Gathering context for vulnerability protection

AOP is a good candidate to correct security concerns, and we have decided to adopt it in our approach. As we explain in previous section, requirements specific to this technology have to be respected : definition of the code advice to mitigate a vulnerability and definition of the pointcut to link the aspect to the base application. The former is facilitated by the Static Analysis Engine that detects a definite set of vulnerabilities. From each vulnerability, it is possible to come with a correction library that is specific and known in advance. The latter is linked to the Static Analysis results, that indicates which vulnerability occurs, at specific places across the application.

The developer does not need to be security expert to correct vulnerabilities as the plugin provides interactive steps to understand the context. The developer decides to apply a patch at a specific place for a given vulnerability. It generates automatically a piece of code which is a patch to be applied at runtime. This patch system is managed thanks to AspectJ, and the security plugin takes care of dynamically generate aspects snippets. The red square depicted in Figure 4 highlights plugin's integration with developer's workbench. The marker provides link to more information on the security vulnerability and how to correct it manually. But it also guides developer step by step towards correction. More precisely, the developer is able to dig into a specific vulnerability and observe impacted code. The corresponding Abstract Syntax Tree is shown to let the user decide the most appropriate location where to apply the correction. For instance, the developer needs to mitigate a malformed input risk (Figure 4). To mitigate it, the developer has the choice to provide early correction through

input validation, and late correction with proper encoding to avoid further client side misinterpretation. For the latter, the user decides which encoding function has to be applied.

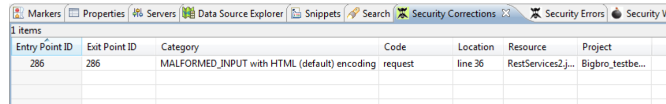

Fig. 5. Correction applied

Once developer specified the context and decided to apply the correction, the plugin generates a correction aspect that is added to an internal list of correction (cf Figure 5). The list allows developers to know how code is impacted. The plugin uses this list at the build time, to embed security mechanisms along binaries. The plugin configures Eclipse build system to automatically weave aspects into the system, without intervention from the developer. It ensures a better modularity with regards to the security, and allows one to correct weaknesses discovered thanks to the previous automation.

Automatic correction integrated in an Eclipse plugin is made possible thanks to inputs from the developer but also thanks to knowledge of vulnerabilities location. The automation brought by the Eclipse plugin allows a broader and consistent application of security across applications. We are yet at an early stage of this work and would like to investigate different impacts of agile and decentralized approach with this system. The first impact is conflict analysis when several aspects impact a path part of the Abstract Syntax Tree. A single application using different aspects woven at different nodes along tree-path can invalidate the initial goal - apply protection. For instance, a web application encoding a parameter twice leads to an improper correction. The conflict analysis is even bigger when one consider several developers participating to a single project. Each collaborator can introduce aspects mechanisms that can bring side-effects. To overcome these issues, the knowledge database can be used to optimize aspect application and requires security expert to decide when a conflict is undecidable.

6 Related Work

The interest into static analysis field has grown over the last decades and brought advancement in the field. Different approaches exist to perform static analysis, from first static analysis tools that uses simple pattern matching techniques, such as RATS or ITS4 [2], to tools that use complex techniques that relies on data flow analysis with context sensitivity, such as the ones presented in [10] or [12]. Different approaches, such as [7] or [16], use string analysis, presented in [3], to detect SQL Injection vulnerabilities. Pixy, presented in [8], is a tool that use data flow analysis to detect XSS vulnerabilities for PHP code. This work differs from our primarily for the language and for the approach that is not integrated into the daily development lifecycle.

Commercial tools, such as Fortify[3] or CodeProfiler[4], can detect also input validation errors, often they can work on more than one programming language, and they are optimized to scan very large code base, such as hundreds of thousands of lines of code. Their defect, for us, is that they require specific knowledge in order to use them. Also, they don't provide decentralized approach to embed static analysis process in software lifecycle. Findbugs[5], developed by University of Maryland, is a static analysis tool to detect bugs into Java code. It provides several front ends, including an Eclipse plugin. The focus is more bug oriented rather than security flaw detection.

Several Eclipse plugins have been developed. Benjamin Livshit released-LAPSE [13] that provides static analysis to detect vulnerabilities in Java web application that implements results presented in [10]. CodePro Analytix, by Google [6], provides several tools to improve software quality and a static analyzer. It search for security bugs, such as errors caused by a lack of input validation, and it allows the user defining what classes and methods could be considered tainted sources or tainted sinks. SSVChecker, presented in [5], executes different static analysis tools (in detail RATS, ExPat and Tuits4) to detect bugs into C/C++ source code. It gives to users the possibility of making operations of union and intersection on results provided by different tools. As it uses other tools, it has all the advantages and drawbacks of them. LAPSE and CodePro Analytix differ from our work as they do not propose decentralized management of security knowledge.

7 Conclusions and Future Works

In this paper, we presented an agile and decentralized approach for static analysis where developers are required to analyse code they produce with expertise support. The code analysis process is embedded in the security lifecycle to be maintainable by security experts while applicable by developers. We make usage of the day-to-day development environment tools - aka Eclipse - to automate the process and integrate a solution closer to the developer. A single environment allows from detection to correction and track of common vulnerabilities during the development phase.

Currently, the tool we created support major flaws in web applications, but we oriented the architecture towards modularity. We allow ones to describe new vulnerabilities in an inter-exchange format based on OVAL language. The tool also guide developers towards vulnerability mitigation using Aspect-Oriented-Programming concepts to separate correction logic from application logic whenever it is possible.

[3] Fortify 360 - https://www.fortify.com/
[4] CodeProfiler - http://www.codeprofilers.com/
[5] Findbugs - http://findbugs.sourceforge.net

Acknowledgment

This work has been partially carried out in the CESSA (Compositional Evolution of Secure Services using Aspects) project, supported by the ANR, the French national research organization (project id.: 09-SEGI-002-01).

References

1. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
2. John Viega Bloch. A static vulnerability scanner for c and c++ code. In *16th Annual Conference Computer Security Applications*, ACSAC'00, 2000.
3. Aske Simon Christensen, Anders Moller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium*, SAS'03, pages 1–18. Springer-Verlag, 2003.
4. MITRE Corporation. Oval : Open vulnerabilities and assesment language web site. http://oval.mitre.org/.
5. Josh Dehlinger, Qian Feng, and Lan Hu. Ssvchecker: unifying static security vulnerability detection tools in an eclipse plug-in. In *Proc. OOPSLA Workshop on eclipse technology eXchange*, Eclipse'06, pages 30–34. ACM, 2006.
6. Google. Codepro analytix. http://code.google.com/javadevtools/codepro/.
7. Carl Gould, Zhendong Su, and Premkumar Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. In *Proc. 26th International Conference on Software Engineering*, ICSE '04, pages 697–698. IEEE Computer Society, 2004.
8. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium On Security And Privacy*, pages 258–263, 2006.
9. Gregor Kiczales, John Lamping, and al. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.
10. Monica S. Lam, John Whaley, V. Benjamin Livshits, and al. Context-sensitive program analysis as database queries. In *Symposium on Principles of database systems*, PODS'05, pages 1–12. ACM, 2005.
11. Peter Liggesmeyer, Martin Rothfelder, and al. Qualitätssicherung softwarebasierter technischer systeme - problembereiche und lösungsansätze. *Informatik-Spektrum*, 21:249–258, 1998. 10.1007/s002870050102.
12. Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows. *Software Maintenance and Reengineering (CSMR)*, 2010.
13. V. Benjamin Livshits. Lapse: Web application security scanner for java. http://suif.stanford.edu/ livshits/work/lapse/index.html.
14. Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proceedings of Financial Cryptography and Data Security 2011*, Lecture Notes in Computer Science, February 2011.
15. Bryan Sullivan. Agile sdl - streamline security practices for agile development. http://msdn.microsoft.com/en-us/magazine/dd153756.aspx.
16. Gary Wassermann and Zhendong Su. An analysis framework for security in web applications. In *Proc. FSE Workshop on Specification and Verification of Component-Based Systems*, SAVCBS'04, pages 70–78, 2004.