Towards Assisted Remediation of Security Vulnerabilities

Gabriel Serme[§], Anderson Santana De Oliveira[*], Marco Guarnieri[†] and Paul El Khoury[‡]
[*]*SAP Research*
Sophia Antipolis, France
{name.lastname}@sap.com
[§]*Eurecom*
Sophia Antipolis, France
serme@eurecom.fr
[†]*Dept. of Information Technology*
and Mathematical Methods
University of Bergamo, Italy
0guarnieri.marco0@gmail.com
[‡]*SAP AG*
Walldorf, Germany
paul.el.khoury@sap.com

***Abstract*—Security vulnerabilities are still prevalent in systems despite the existence of their countermeasures for several decades. In order to detect the security vulnerabilities missed by developers, complex solutions are undertaken like static analysis, often after the development phase and with a loss of context. Although vulnerabilities are found, there is also an absence of systematic protection against them. In this paper, we introduce an integrated Eclipse plug-in to assist developers in the detection and mitigation of security vulnerabilities using Aspect-Oriented Programming early in the development life-cycle. The work is a combination of static analysis and protection code generation during the development phase. We leverage the developer interaction with the integrated tool to obtain more knowledge about the system, and to report back a better overview of the different security aspects already applied, then we discuss challenges for such code correction approach. The results are an in-depth solution to assist developers to provide software with higher security standards.**

***Keywords*-Security, AOP, Software Engineering, Static Analysis, Vulnerability Remediation**

I. Introduction

After a decade of existence, cross-site scripting (XSS), SQL Injection and other of types of security vulnerabilities associated to input validation can cause severe damage once exploited. To analyze this fact, Scholte et al. [1] conducted an empirical study that shows that the number of reported vulnerabilities is not decreasing.

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. Thus, 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 some benign code. Such kind of mistakes can survive unaudited for years until they end up exploited by hackers.

The software development lifecycle introduces several steps to audit and test the code produced by developers in order to detect the security bugs, such as code review tools for early detection of security bugs to penetration testing. The tools are used to automate some tasks normally handled manually or requiring complex processing and data manipulation. They are able to detect several of errors and software defects, but developers have to face heterogeneous tools, each one with a different process to make it run correctly, and they have to analyze the results of all the tools, merge them and fix the source code accordingly. For instance, code scanner tools are usually designed to be independent from the developers' environment. Therefore, they gain in flexibility but loose comprehensiveness and the possibility to interact with people having the experience on application code. Thus, tools produce results that are not directly linked to application defects. It is the case for example for code scanner tools triggering several false positives, which are not actual vulnerabilities.

The contributions of this paper are twofold. First, we focus on static code analysis, an automated approach to perform code review integrated in developer's environment. This technique analyzes the source code and/or binary code without executing it and identifies anti-patterns that lead to security bugs. We focus on security vulnerabilities caused by missing input validation, the process of validating all the inputs to an application before using it. Although our tool handles other kinds of vulnerabilities, here we discuss on three main vulnerabilities caused by missing input validation, or mis-validation of the input: cross-site scripting (also called XSS), Directory Path Traversal and SQL Injection. Second, we provide an innovative assisted remediation process that employs Aspect-Oriented Programming for semi-automatic vulnerability correction. The combination of these mechanisms improves the quality of the software with respect to security requirements.

The paper is structured as follows: Section II presents the overall agile approach to conduct code scanning and correct vulnerability during the development phase. Then, Section III presents the architecture we adopt to combine the static analysis with the code correction component. The Section IV describes the static analysis process with its integration in the developers' environment. Then, we explain techniques for assisted remediation along with pros and cons in Section V. Finally, we discuss the advantages of our approach compared to related work in Section VI, and we conclude in Section VII.

II. An agile approach

Agile approaches to software development require the code to be refactored, reviewed and tested at each iteration of the development lifecycle. While unit testing can be used to check functional requirements fulfillment during iterations, checking emerging properties of software such as security or safety is more difficult. We aim to provide each developer with a simple way to do daily security static analysis on his code. That would be properly achieved by providing a security code scanner integrated in the development environment, i.e., Eclipse IDE in this case, and a decentralized architecture that allows the security experts to assist the developers in any of the findings. Typically, that would include verifying false positives and correspondingly adjusting the code scanner test cases, or assisting in reviewing the solutions for the fixes. It brings several advantages over the approach in which the static analysis phase stays only at the end. The expertise of the context in which the code was developed lies in development groups. Therefore, the interaction between development team and security experts is faster with less effort in finding and applying corrections on the security functionalities. The experts provide support on a case basis for a better tuning of false positive detection across teams and reducing final costs of maintenance: solving security issues into the development phase can reduce the number of issues that the security experts should analyze at the end.

Maintaining the separation of roles between the security experts performing the code scanning and the team members developing the application raises 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 up-skilling the developers and reducing the interaction between them and the security experts for the analysis of the security scans of the project, which is simplified by the introduction of our tool.

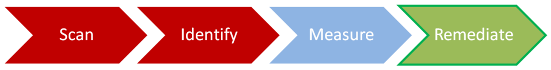


Figure 1. Vulnerability remediation process. The red corresponds to the static analysis component. The green one corresponds to the remediation component. The blue one corresponds to assisted processing

Our incentive is to harvest 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 development time and reduces maintenance costs. A tool covering the previous needs should fulfill several requirements:

- *easy-to use* for users non-experts in security
- *domain specific* with integration into developers' daily environment, to maximize adoption and avoid additional steps to run the tool
- *adjustable* to maximize project knowledge and reduce false positives and negatives
- *reflexive* to adjust accuracy of the scan over time, with collaborative feedbacks for example
- *supportive* to assist developers in correcting and understand issues.
- *educative* to help developers understanding errors, steps to correct existing error, and techniques to prevent future vulnerability

We have developed an Eclipse Plugin, presented in [2], made of components leveraging decentralized approach for static analysis. It gives direct access to detected flaws and global overview on system vulnerabilities. The developer analyzes its code and review vulnerabilities when necessary.

Figure 1 presents the interaction between the two phases: the static analysis phase allows scanning the code in order to identify and classify the different vulnerabilities found. It is described in details in Section IV. The measurement is performed directly by developers who decide what to remediate by undertaken actions, with support from our second component. The full remediation process is given in Section V .

III. Architecture

Figure 2 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 and developers regroup different profiles whose goal is to provide and configure the knowledge database in order to avoid false positives and negatives, and to provide better accuracy during the analysis phase. They have two main tasks. First, they update the knowledge base, adding to its classes or methods that can be considered as trusted for one or more vulnerabilities. Second, the knowledge database receives feedback from analysis on possible trusted objects for one or more security vulnerabilities; they must analyze them more in detail and, if these objects are really trusted they tag them as trusted into the knowledge base. We better explain the different concepts and tasks in Section IV.

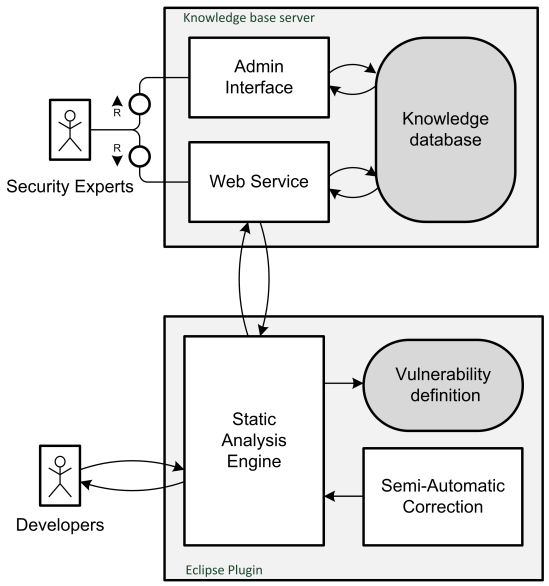


Figure 2. Architecture

The second role is the developer, interacting directly with the static analysis engine to verify vulnerabilities in application, code and libraries under its responsibility. The developer at this stage can be naive, therefore with no focus on complexity of security flow. The knowledge base is shared among developers. It contains all the security knowledge about trust: objects that do not introduce security issues into the code. Security experts and developers with understanding of security patterns maintain and keep under control the definitions used by all developers in an easy way using one admin web application or some web-services. In this way the code scanner testing rules are harmonized for the whole application or even on a project-basis. The knowledge base allows developers to run static analysis that is perfectly adapted to the context of their project.

In industrial scale projects, daily scans are recommended. In order to facilitate this task, we provide a plugin for Eclipse that uses an abstract syntax tree (AST) generated by the JDT compiler - the compiler that Eclipse provides as part of the Java Development Tools platform, to simplify the static analysis process. The plugin accesses the knowledge database via web-services making it possible to each developer to run independently the code scanner. We detail its components in the next section.

IV. Static analysis

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. Modern static analysis tools, similarly to compilers, build an abstract syntax tree that represents the abstract syntactic structure of the code from the source code and analyze it.

A. Static Analysis Process

In a nutshell, our process allows developers to run a check on their code to uncover potential vulnerabilities by checking for inputs that have not been validated. It finds information flows connecting an entry point and exit point that does not use a trusted object for the considered vulnerabilities. The algorithm uses an abstract syntax tree of the software in conjunction with the knowledge base to identify the vulnerable points. The Figure 3 presents the different analysis steps performed from the moment developer presses the analysis button to the display of results.

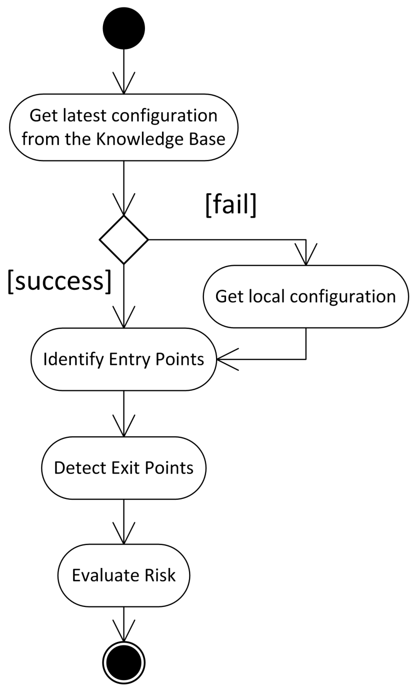


Figure 3. Static Analysis Activity Diagram

The static analysis works on Document Object Model generated by the Eclipse JDT component able of handling all constructs described in the Java Language Specification [3]. The static analysis process is described as follows:

- The engine contacts the knowledge database in order to retrieve the up-to-date and most accurate configuration from the shared platform. If the developer cannot retrieve the configuration, it can still work independently with the latest local configuration.
- The process identifies all *entry points* of interest in the accessible source code and libraries. The analysis is based on the previously mentioned AST. We are gathering the different variables and fields used as well as the different methods. We apply a first filter with pattern-matching on the potential *entry points*: a method call or a new object instantiation might be tagged as returning trusted inputs.
- For each *entry point* the control flow is followed to create the connections between methods, variables and fields to discover all the *exit points*. For instance, the engine visits assignments, method invocations and construction of new objects with the variables and fields detected during the entry point gathering.
- Once the different *exit points* have been collected, we evaluate the risk of having security vulnerabilities in the code. We check for an absence of validation in the flow for the different kinds of vulnerabilities. For instance, if the flow from an entry point to an exit point passes through a method or a class, which is known to validate SQL input, the flow is tagged as trusted for this specific vulnerability. Of course, the tag runs from the moment where the method validates for the vulnerability to the moment of a novel composition with potential vulnerable code, or until an *exit point*.

B. Multiple vulnerability analysis

In the previous section, we have presented the global analysis process. In this section, we discuss more in-depth the notion of trusted object and vulnerability propagation for the different vulnerabilities we address. The Listing 1 presents some source code vulnerable to cross site scripting. The vulnerability propagates from the request parameter to the object *query*, which is then written in the response. 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.

```java
/** This servlet proposes XSS example. */
public class EchoServlet extends HttpServlet {
  protected void doGet(HttpServletRequest req,
        HttpServletResponse resp) {
    PrintWriter writer = resp.getWriter();
    String query =
          req.getParameter("query") ;

    resp.setContentType("text/html");
    writer.print("<html><h1>Results for ");
    writer.print (query);
    writer.print("</h1></html>");
    writer.flush();
    writer.close();
    }
}
```

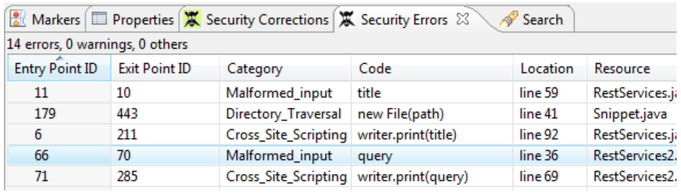
Listing 1. Vulnerability propagation of a cross site scripting

We define an *input* as a data flow from any external class, method or parameter into the code being programmed. We also define as *entry point* any point into the source code where an *untrusted input* enters to the program being scanned, like the *query* input from Listing 1. In an analogous way we define as *output* any data flow that goes from the code being programmed into external objects or method invocations. Our approach relies on our *trusted object* definition, which impacts the detection accuracy. A *trusted object* is a class or a method that can sanitizes all the information flow from an *entry point* to an *exit point* for one or more security vulnerabilities. We implemented the trust 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 a trusted element in the trust hierarchy also adds all the elements below it: trusting a package trusts all the classes and methods into it and trusting a class trusts all the fields and methods in it. A trusted object can sanitize one or more security vulnerabilities (e.g., sanitization method can be valid for both SQL Injection and cross site scripting). This approach enables developers and security experts to define strong trust policies with regards to the system they are securing.

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 must be rigorous: it influences the risk evaluation accuracy. The object must not introduce a specific vulnerability into the code. This is the reason why developers report feedback and security experts take the decision. The experts can also analyze, 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 fewer false positives while ensuring no false negatives.

The detected vulnerabilities (Figure 4 gives an example of analysis result in the tool) 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 represents any input that is not validated using a standard implementation.



14 errors, 0 warnings, 0 others

Entry Point ID	Exit Point ID	Category	Code	Location	Resource
11	10	Malformed_input	title	line 59	RestServices.j
179	443	Directory_Traversal	new File(path)	line 41	Snippet.java
6	211	Cross_Site_Scripting	writer.print(title)	line 92	RestServices.j
66	70	Malformed_input	query	line 36	RestServices2.
71	285	Cross_Site_Scripting	writer.print(query)	line 69	RestServices2.

Figure 4. Code Analysis result

Vulnerability	**Origin**	**Potential Remediation**
Cross-Site Scripting	Server does not validate input coming from external source	Validate input and filter or encode properly the output depending on the usage: the encoding differs from HTML content to Javascript content for example
SQL Injection	Server does not validate input and use it directly in a construct of a SQL Query	Use a parameterized query or a safe API. Escape special characters. Validate the input used in the construction of query
Directory Path Traversal	Application server is misconfigured, or the file-system policy contains weaknesses	Enclose the application with strict policies, that restrict access to the filesystem by default. Filter and validate the input prior to direct file access
Other malformed input	Misvalidation	Validate input, determine the origin and possible manipulation from externals

Table I
LIST OF DETECTED VULNERABILITIES WITH POTENTIAL ORIGIN AND POTENTIAL REMEDIATION.

The engine can be easily extended to support new kinds of vulnerabilities caused by missing input validation. One needs to add the definition of the new vulnerability to the centralized knowledge base (and, if exist, adding trusted objects that mitigate it), and creating a new class extending an interface, that implements the checks to be done on the result of the static analysis to detect the vulnerability.

V. Assisted Remediation

Performing static analysis is yet integrated in quality processes in several companies. But, the actual identification of vulnerabilities does not mean they are correctly mitigated. Given this problem, we can have several approaches: (i) refactoring the code, (ii) applying a proxy in inbound and outbound connections, and finally the solution we adopted, (iii) to generate protection code linked to the application being analyzed.

Software refactoring involves the developer into understanding the design of its application and the potential threats, to manually rewrite part of the code. The refactoring improves the design, performance and manageability of the code, but is difficult to address. It costs time and is error prone. Up to six distinct activities have been observed in [4] from identification to verification of refactoring. The impacted code is generally scattered over the application, and some part can be left unchecked easily. This can lead to an inconsistent state where the application does not reflect the intended goal. In terms of vulnerability remediation, the software refactoring is one of the most powerful due to the flexibility in terms of code rewriting and architecture evolution.

The proxy solution is equivalent to a gray-box approach, with no in-depth visibility of internal processes. It can be heavy to put in place, especially when the environment is under control of a different entity than the development team. For instance, on cloud platforms, one can deploy its application but has limited management on other capabilities, leading to the impossibility to apply filter on the application. The lack of flexibility and the absence of small adjustments make it complicated to adopt at the development phase.

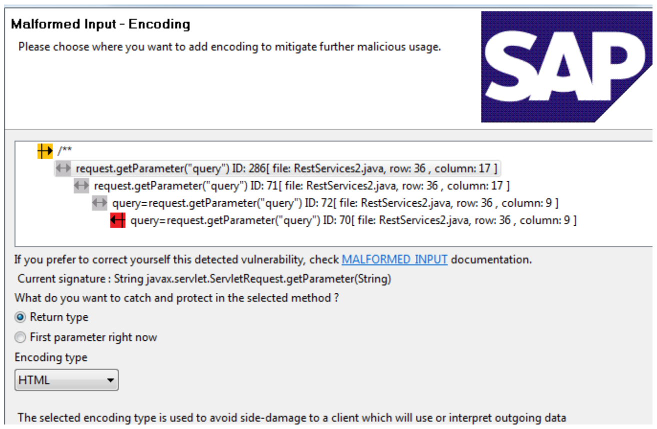
In this work we provide inline protection with the application. This solution has several advantages, but also brings new limitations due to the technology we use: Aspect-Oriented Programming paradigm (AOP) [5], which is a paradigm to ease programming concerns that crosscut and pervade applications. In the next section, we describe our methodology and provide a comprehensive list of advantages and drawbacks.

A. Methodology

The approach comprises the automatic discovery of vulnerability and weaknesses in the code. In addition, we integrate a protection phase tied to the analysis process which guides 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 inputs from the developer to extract knowledge about the context, like in Figure 5. These steps allow gathering places in the code where to inject security correction. The security correction uses AOP. The goal is to bring proper separation of concerns for cross cutting functionalities, such as security. Thus, code related to a concern is maintained separately from the base application. The main advantage using this technology is the ability to intervene in the control flow of a program without interfering with the base program code.

The list of vulnerability we cover principally are in Table I. The Table highlights the potential origin vulnerabilities and some of known remediation techniques. These vulnerabilities are known and subject to high attention. For instance, we can retrieve them in the OWASP Top Ten [6] for several years now, but also in the MITRE Top 25 Most Dangerous Software Errors [7]. Albeit several approaches exist to remediate the vulnerabilities, we are considering mainly escaping and validation to consistently remediate the problems with the aspect-oriented technique.

By adopting this approach, we reduce the time to correct vulnerabilities by applying semi-automatic and pre-defined


Malformed Input - Encoding

Please choose where you want to add encoding to mitigate further malicious usage.

```
/**
  request.getParameter("query") ID: 286[ file: RestServices2.java, row: 36 , column: 17 ]
    request.getParameter("query") ID: 71[ file: RestServices2.java, row: 36 , column: 17 ]
      query=request.getParameter("query") ID: 72[ file: RestServices2.java, row: 36 , column: 9 ]
        query=request.getParameter("query") ID: 70[ file: RestServices2.java, row: 36 , column: 9 ]
```

If you prefer to correct yourself this detected vulnerability, check MALFORMED INPUT documentation.

Current signature : String javax.servlet.ServletRequest.getParameter(String)

What do you want to catch and protect in the selected method ?

[✓ Return type]

First parameter right now

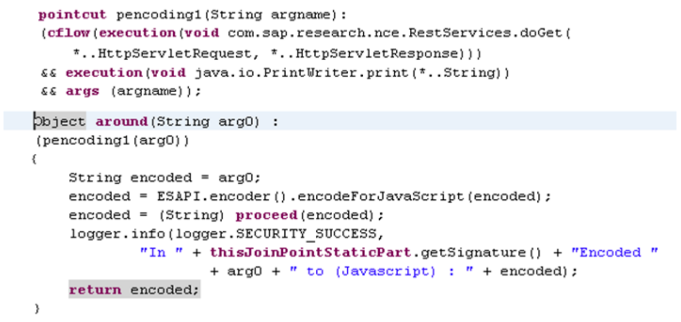
Encoding type: HTML

The selected encoding type is used to avoid side-damage to a client which will use or interpret outgoing data

Figure 5. Gathering context for vulnerability protection

mechanisms to mitigate them. We use the component to apply protection code which is mostly tangled and scattered over an application.

Correcting a security vulnerability is not trivial. Different refactoring are possible depending on the issue. For instance, the guides for secure programming advises SQL prepared statement to prevent SQL Injection. But, developers might be constrained by their frameworks to forge SQL queries themselves. Therefore, developers would try another approach such as input validation and escaping of special characters.


```
pointcut pencoding1(String argname):
 (cflow(execution(void com.sap.research.nce.RestServices.doGet(
     *..HttpServletRequest, *..HttpServletResponse)))
 && execution(void java.io.PrintWriter.print(*..String))
 && args (argname));

Object around(String arg0) :
 (pencoding1(arg0))
{
     String encoded = arg0;
     encoded = ESAPI.encoder().encodeForJavaScript(encoded);
     encoded = (String) proceed(encoded);
     logger.info(logger.SECURITY_SUCCESS,
             "In " + thisJoinPointStaticPart.getSignature() + "Encoded "
                      + arg0 + " to (Javascript) : " + encoded);
     return encoded;
 }
```

Figure 6. Example of correction snippet generated for a malformed input

We assist developers by proposing them automated solutions. For the previously mentioned correction, our integrated solution would propose to mitigate the vulnerability with an automatic detection of incoming, unsafe and unchecked variables. The developer does not need to be security expert to correct vulnerabilities as our approach provides interactive steps to generate AOP protection code, like in Figure 6. Although semi-automation simplifies the process to introduce protection code, the technique can introduce several side-effects if the developers are not following closely what is generated. The plug-in gives an overview for the developer of all corrected vulnerabilities, allowing him to visually manage and re-arrange them in case of need. Currently, the prototype does not analyze interaction between the different protection code generated. By adopting this approach, we allow better understanding from a user point of view of the different vulnerabilities affecting the system, and we guide the developer towards more compliance in its application. The protection code can be deployed by security expert teams and change without refactoring.

B. Constraints from Aspect-Oriented Programming

The usage of AOP in the remediation of vulnerability bring us more flexibility. One can evolve the techniques used to protect the application, by switching the process to resolve a problem, making the security solution independent from the application. But this approach also brings us some limitations we discuss in this section.

Firstly, the language is designed to modify the application control flow. One of the limitations we have is related to the deep modification we need to perform in order to replace partially a behavior. For example, suppose a SQL query written manually in the application we would like to validate. We are able to weave validation and escaping code, but we can hardly modify the application to construct a parameterized query.

Secondly, the aspects cover the application in the whole. When more than one aspect is involved, the cross-cutting concerns can intersect. Therefore, we need to analyze aspect interaction and prevent an annihilation of the behavior we intended to address.

Thirdly, the evolution of the program leads to a different repartition of vulnerabilities. The vulnerabilities are detected after the static analysis phase. We are not addressing yet this problem of evolution to maintain the relation between the aspects and the application. This differs from the *fragile pointcut* problem inherent of aspect using pointcut languages referring to the syntax of the base language: the evolution affects the application as a whole, by introducing new entry points and exit points that need to be considered, or introducing methods that validate a flow for a given vulnerability.

The fourth constraint is that aspects have no specific certification. The actual protection library is defined globally, but applied locally, with a late binding to the application. The protection code is the same everywhere, but we put strong trust in the protection library by assuming that aspects are behaving properly with the actual modification of the flow to mitigate the vulnerability.

Finally, the fifth constraint is user acceptance. Since the developers rely on cross cutting solution, the code itself does not reflect the exact state of the application. The point where the aspect interferes with the base application is not presented in the code. We address this limitation with the strong interaction with the developer's environment. The Eclipse plugin provides a mean to display remediation code in place at a given time.

VI. Related work

The interest into static analysis field has led to several approaches. They go from simple techniques like pattern matching and string analysis like in [8]–[11] to more complex techniques like data flow analysis in [12]–[14]. Commercial tools, such as Fortify [15] or CodeProfiler [16] propose better integration in developers' environment but lack of decentralized approach and assistance in security management. Several tools are based on the Eclipse's platform and detect vulnerabilities in web applications [17] , flaws [18], bugs [19], and propose testing and audit to verify respect of organizational guidelines [20]. Compared to the aforementioned techniques, we advocate a better integration into the daily development lifecycle with our tool, and propose an integrated correction with good accuracy as we leverage developer's knowledge on development context.

Hermosillo et al. [21] uses AOP to protect against web vulnerabilities: XSS and SQL Injection. They use AspectJ - the mainstream AOP language, to intercept method calls in an application server then perform validation on parameters. Viega et al. [22] presents simple use case on the usage of AOP for software security. Masuhara et al. [23] introduces an aspect primitive for dataflow, allowing to detect vulnerabilities like XSS. Our approach reduces the overhead brought by the detection of vulnerability patterns at runtime and allows wider range of vulnerability detection. Also, the aforementioned approaches do not rely on external tools to gather security context, but rather a manual processing to understand the architecture and decide where to apply aspects. Our approach also brings more awareness to the developer as he obtains a visual indicator of what is applied at which place in its application.

A combination of detection and protection is found in Deeprasertkul et al. [24] approach for detecting faults identified by pre-compiled patterns. Faults are corrected using a correction module. The difference with our approach lies in the detection of faults rather than security vulnerabilities. Also, the correction module fixes the faults statically and prevents further modifications of the introduced code. A recent work conducted by Yang et al. [25] uses static analysis to detect security points to deploy protection code with aspects, on distributed tuple space systems. These two approaches suffer from same limitation as the ones presented in the previous paragraph, which is a lack of visual support from the tool, and a loose of context. It is worth mentioning the work from Hafiz et al. [26], where authors propose several techniques to correct data injection through program transformations. They have list several cases along with steps to describe transformations to realize security policies. Their work can benefit our overall methodology to propose multiple corrections once vulnerability has been identified.

VII. Conclusion and future works

We presented how to overcome several security vulnerabilities using a combination between a static analyzer that assists developers to report security vulnerabilities and a semi-automated correction of these findings with AOP. The usage of an integrated tool to provide support for security bugs detection and mitigation has several advantages. It benefits to several stakeholders at the same time. First, security teams are able to distribute the maintenance of the code to the people writing their code and let them mitigate security bugs whenever they are detected. They can interact closely to decide of the best solutions for a given situation, and apply security across development teams. Developers benefit from this approach, having an operational tool already configured for their development. They can focus on writing their functional code and, time to time, verify the accuracy of their implementation. Security concerns are often cross cutting the application, which tends to have security checks spread around application. Using one central tool to have an overview is more efficient and productive, and gives the possibility to track all applied protection code. The automation allows a broader and consistent application of security across applications. The usage of AOP eases the deployment and change of security protection code, in a single environment and during the development phase. The overall vision we would like to achieve in the future is the specification and maintenance of security concerns in one central place, and usage by developers of these concerns by defining some places in application where they should be active.

We have designed this plug-in for an improved awareness of security concerns from a developer point of view. It is important to notice that correcting vulnerabilities doesn't make the whole system secure. It only means the code tends to be free of security bugs. Other parts of the application, such as authentication flow, authorization checks, etc. are not covered by our analysis. Besides, we encourage developers to look further in vulnerabilities' descriptions, as the automated correction proposed might not be the best choice in all situations. We do not want developers to believe our solution is bullet-proof. It leads to a false sensation of security, which is the opposite of our goal.

Albeit we have listed several benefits for an integrated tool, we know that it suffers from limitations. For instance, when we are developing a tool such as an Eclipse plug-in, we are targeting a platform and a language, thus voluntarily restricting the scope of application. From the tool itself, we have designed a working prototype that we have validated on projects internally at SAP and compared to commercial softwares. In several cases, the agile approach leads to a reduction of false positives and an absence of false negatives. Also, the approach of providing support for correcting the vulnerability is novel and we focus now in improving accuracy of the protection code. Especially, we need to investigate in the cost in term of complexity and maintainability for the different stakeholders interacting with the system.

Acknowledgment

This work has been partially carried out in the CESSA project (project id.: 09-SEGI-002-01).

References

[1] T. Scholte, D. Balzarotti, and E. Kirda, "Quo vadis? a study of the evolution of input validation vulnerabilities in web applications," in *Proceedings of Financial Cryptography and Data Security 2011*, ser. Lecture Notes in Computer Science, February 2011.

[2] M. Guarnieri, P. El Khoury, and G. Serme, "Security vulnerabilities detection and protection using eclipse," in *ECLIPSE-IT 2011*, Milano, ITALY, September 2011.

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha, "Java(TM) Language Specification," http://docs.oracle.com/javase/specs/, January 2005.

[4] T. Mens and T. Tourwe, "A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126 – 139, February 2004.

[5] G. Kiczales, J. Lamping, and al., "Aspect-oriented programming," in *ECOOP*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds. Springer Berlin / Heidelberg, 1997, vol. 1241, pp. 220–242.

[6] OWASP, "OWASP Top Ten Project," http://www.owasp.org/index.php/OWASP_Top_Ten_Project, 2010.

[7] MITRE, "CWE/SANS Top 25 Most Dangerous Software Errors," http://cwe.mitre.org/top25, September 2011.

[8] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *ACSAC*. IEEE Computer Society, 2000, pp. 257–.

[9] C. Gould, Z. Su, and P. T. Devanbu, "Jdbc checker: A static analysis tool for sql/jdbc applications," in *ICSE*. IEEE Computer Society, 2004, pp. 697–698.

[10] G. Wassermann and Z. Su, "An analysis framework for security in web applications," in *Proc. FSE Workshop on Specification and Verification of Component-Based Systems*, ser. SAVCBS'04, 2004, pp. 70–78.

[11] A. S. Christensen, A. Moller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium*, ser. SAS'03. Springer-Verlag, 2003, pp. 1–18.

[12] M. S. Lam, J. Whaley, V. B. Livshits, and al., "Context-sensitive program analysis as database queries," in *Symposium on Principles of database systems*, ser. PODS'05. ACM, 2005, pp. 1–12.

[13] Y. Liu and A. Milanova, "Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows," in *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, ser. CSMR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 146–155. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2010.26

[14] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2008, pp. 387–401.

[15] HP, "Fortify 360," https://www.fortify.com/, June 2012.

[16] Virtual Forge, "Codeprofilers," http://www.codeprofilers.com/, June 2012.

[17] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," in *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005, pp. 271–286.

[18] J. Dehlinger, Q. Feng, and L. Hu, "Ssvchecker: unifying static security vulnerability detection tools in an eclipse plug-in," in *Proc. OOPSLA Workshop on eclipse technology eXchange*, ser. Eclipse'06. ACM, 2006, pp. 30–34.

[19] University of Maryland, "Findbugs," http://findbugs.sourceforge.net, July 2012.

[20] Google, "Codepro analytix," http://code.google.com/javadevtools/codepro/, June 2012.

[21] G. Hermosillo, R. Gomez, L. Seinturier, and L. Duchien, "Aprosec: an aspect for programming secure web applications," in *ARES*. IEEE Computer Society, 2007, pp. 1026–1033.

[22] J. Viega, J. T. Bloch, and P. Ch, "Applying aspect-oriented programming to security," *Cutter IT Journal*, vol. 14, pp. 31–39, 2001.

[23] H. Masuhara and K. Kawauchi, "Dataflow pointcut in aspect-oriented programming," in *APLAS*, ser. Lecture Notes in Computer Science, A. Ohori, Ed., vol. 2895. Springer, 2003, pp. 105–121.

[24] P. Deeprasertkul, P. Bhattarakosol, and F. O'Brien, "Automatic detection and correction of programming faults for software applications," *Journal of Systems and Software*, vol. 78, no. 2, pp. 101–110, 2005.

[25] F. Yang, T. Aotani, H. Masuhara, F. Nielson, and H. R. Nielson, "Combining static analysis and runtime checking in security aspects for distributed tuple spaces," in *COORDINATION*, ser. Lecture Notes in Computer Science, W. D. Meuter and G.-C. Roman, Eds., vol. 6721. Springer, 2011, pp. 202–218.

[26] M. Hafiz, P. Adamczyk, and R. Johnson, "Systematically eradicating data injection attacks using security-oriented program transformations," in *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, ser. ESSoS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 75–90.