



ASIDE: IDE Support for Web Application Security

Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton
 Department of Software and Information Systems
 University of North Carolina at Charlotte
 9201 University City Blvd
 Charlotte, NC, 28223, USA
 jxie2, billchu, heather.lipford@uncc.edu, jtmelton@gmail.com

ABSTRACT

Many of today's application security vulnerabilities are introduced by software developers writing insecure code. This may be due to either a lack of understanding of secure programming practices, and/or developers' lapses of attention on security. Much work on software security has focused on detecting software vulnerabilities through automated analysis techniques. While they are effective, we believe they are not sufficient. We propose to increase developer awareness and promote practice of secure programming by interactively reminding programmers of secure programming practices inside Integrated Development Environments (IDEs). We have implemented a proof-of-concept plugin for Eclipse and Java. Initial evaluation results show that this approach can detect and address common web application vulnerabilities and can serve as an effective aid for programmers. Our approach can also effectively complement existing software security best practices and significantly increase developer productivity.

Categories and Subject Descriptors

D.2 [Software Engineering]: [Software Security]; D.2.4 [Software/Program Verification]: [Correctness Proofs]; D.2.6 [Programming Environments]: [Integrated Environments]

General Terms

Security

Keywords

Secure programming, application security, interactive support, secure software development

1. INTRODUCTION

Software vulnerabilities originating from insecure code are one of the leading causes of security problems people face

today [36]. Unfortunately, many developers have not been adequately trained in writing secure programs that are resistant from attacks violating program confidentiality, integrity, and availability. We refer to this concept of writing secure code as *secure programming* [4]. Programmer errors, including security ones, are unavoidable even for well-trained developers. One major cause of such errors is software developers' heavy cognitive load in dealing with a multitude of issues, such as functional requirements, runtime performance, deadlines, and security. Consider Donald Knuth's analysis of **867** software errors he made while writing T_EX [19]. It is clear from the log that some of these errors could have made T_EX vulnerable to security breaches. The following quote illustrates Knuth's experience of heavy cognitive burden as a major source of software errors:

"Here I did not remember to do everything I had intended when I actually got around to writing a particular part of the code. It was a simple error of omission, rather than commission. . . This seems to be one of my favorite mistakes: I often forget the most obvious things" [19].

Current tool support for secure programming, both from tool vendors as well as within the research community, focuses on catching security errors after the program is written. Static and dynamic analyzers work in a similar way as early compilers: developers must first run the tool, obtain and analyze results, diagnose programs, and finally fix the code if necessary. Thus, these tools tend to be used to find vulnerabilities at the end of the development lifecycle. However, their popularity does not guarantee utilization; other business priorities may take precedence. Moreover, using such tools often requires some security expertise and can be costly. If programmers are removed from this analysis process, these tools will also not help prevent them from continuing to write insecure code.

We believe these vulnerability detection tools could be complemented by interactive support that reminds developers of good secure programming practices in situ, helping them to either close the secure programming knowledge gap or overcome attention/memory lapses. This approach can be justified based on cognitive theories of programmer errors [20, 21, 38]. Our hypothesis is that by providing effective reminder support in an IDE, one can effectively reduce common security vulnerabilities. Our approach is analogous to word processor spelling and grammar support. While people can run spelling and grammar checks after they have written a document, today's word processors also provide visual

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA
 Copyright 2011 ACM 978-1-4503-0672-0/11/12 ...\$10.00.

cues – colored lines drawn underneath potential errors – to help writers notice and fix problems while they are composing text. Similarly, our approach proposes that an IDE could interactively identify parts of the program where security considerations, such as input validation/encoding or Cross-site Request Forgery (CRSF) protection, are needed while programmers are writing code.

In this paper, we describe our interactive reminder approach through the **Application Security IDE (ASIDE)**, an Eclipse plugin for Java. ASIDE uses two key techniques to help programmers prevent errors: *interactive code refactoring* and *interactive code annotation*. While we have previously introduced this *idea* of an interactive approach to secure programming [37], in this paper, we present our prototype *implementation*, along with a detailed *evaluation* of this plugin on two large-scale open source code bases, demonstrating its potential feasibility and effectiveness at identifying and preventing important security vulnerabilities. We also describe a user evaluation showing that users embrace such unobtrusive and helpful reminder support, and actively engage in the interaction with ASIDE. Thus, the major contribution of this paper is the multi-aspect evaluation of our proposed approach. Through our evaluation process, we also identify directions for future improvements to ASIDE and research.

2. ASIDE: APPLICATION SECURITY IN IDE

The ASIDE plugin is a prototype implementation of our interactive approach to reminding and helping programmers to perform good secure programming practices. Our approach was based on a number of design considerations. First, it is easiest and most cost effective for developers to write secure code and to document security implementation during program construction. This means creating a tool that integrates into the programmers' development environment is promising. As a starting point, we chose the popular Eclipse platform, and implemented ASIDE to work with Java. Our current implementation of ASIDE is primarily designed to assist Web-based applications and to mitigate commonly committed vulnerabilities due to improper input validation and/or filtering, broken access control, and Cross-site Request Forgery (CSRF). ASIDE, once activated, continuously monitors Eclipse workspace changes in order to respond to newly created projects, as well as modifications to existing projects. ASIDE performs static analysis incrementally on a small chunk of code under construction, thus ensuring prompt response to developers' immediate code editing. Such analysis is carried out "under the hood" without requiring developers to understand its security implications.

Our second consideration is the interface design principle that recognition is favored over recall [33]. Developers are provided with appropriate visual alerts on secure programming issues and offered assistance to practice secure programming. ASIDE provides these alerts as a colored icon on the left hand side margin of the code editing window, accompanied with highlighted text, see Figure 1. A developer can then either click on that icon or hover over the highlighted text to interact with ASIDE to address the issue. Our two key mechanisms for this interactive support are *interactive code refactoring* and *interactive code annotation*, which are discussed in detail in the following sections. There are no additional steps a developer must remember, and they can interact with ASIDE when they choose. Thus, the tool acts

as a helpful assistant and does not hinder developers' creativity and productivity by dictating a rigid approach to secure programming.

Third, an in-situ reminder tool can be an effective training aid that either helps novices to learn secure programming practices or reinforces developers' secure programming training, making security a first class concern throughout the development process. This will help developers learn to reduce their programmer errors over time, reducing costly analysis and testing after implementation.

Finally, we want to support sharing secure programming knowledge and standards amongst development teams. In an industrial setting, ASIDE could be configured by an organization's software security group (SSG), which is responsible for ensuring software security as identified by best industry practice [11]. Thus, a SSG could use ASIDE to communicate and promote organizational and/or application-specific programming standards. In addition, the plugin can generate logs of how security considerations were addressed during construction, providing necessary information for more effective code review and auditing.

ASIDE first must detect potential vulnerabilities in program code, and alert programmers to those vulnerabilities. Thus, to be effective, ASIDE must exhibit a certain level of precision in identifying vulnerable code in order to provide proper assistance to address it. Based on our prototype implementation, a vulnerability can be easily eliminated by either of the two interactive techniques if the corresponding vulnerable code is correctly identified. Therefore, a key measure of the effectiveness of ASIDE is the measure of its ability to find vulnerable code. For each of the interactive techniques presented below, we present the details of how effective ASIDE is at detecting the corresponding vulnerabilities. The effectiveness of the interactive techniques to address those vulnerabilities heavily depends on how the programmer responds to the warning. This we evaluate based on a preliminary user study of programmer behaviors, presented in Section 7.

3. INTERACTIVE CODE REFACTORING

Interactive code refactoring [37] works in a manner similar to a word processor that corrects spelling and grammatical errors. In this case, ASIDE automatically inserts necessary code, with the help of the programmer. We believe code refactoring can be appropriately applied to input validation and/or filtering types of vulnerabilities. In this section, we discuss an example concerning input validation to illustrate the key concepts.

A developer is alerted by a marker and highlighted text in the edit window when input validation is needed. ASIDE has a rule-based specification language, which is also XML-based, to specify sources of untrusted inputs which we formally named as trust boundaries. Currently two types of rules are supported: Method (API) invocations, for example, `method textttgetParameter(String parameter)` in class `HttpServletRequest` introduces user inputs from clients into the system; and Parameter input, for instance, arguments of the Java program entrance method `main(String[] args)`.

With a mouse click, the developer has access to a list of possible validation options, such as a file path, URL, date, or safe text. Upon the selection of an option, appropriate input validation code will be inserted and the red marker will be dismissed. Figure 1 shows a screenshot of ASIDE facilitating

a developer to select an appropriate input validation type for an identified untrusted input. The library of input validation options can be easily reconfigured by an individual developer or an organization.

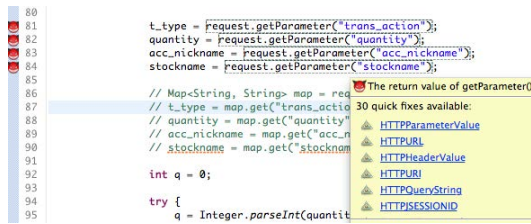


Figure 1: The user interactively chooses the type of input to be validated using a white-list approach.

Figure 2 illustrates how ASIDE refactors code to perform input validation using OWASP Enterprise Security API (ESAPI) Validator [28].

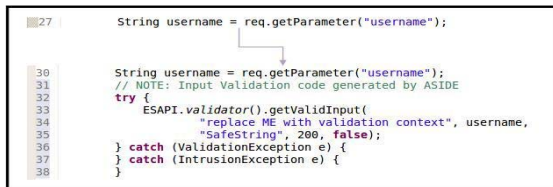


Figure 2: ASIDE validates an input using OWASP ESAPI validator API.

Previous works employing refactoring techniques for secure programming use program transformation rules, which operate on completed programs, and thus work best on legacy code. One recognized limitation of the program transformation approach is the lack of knowledge of specific business logic and context [12]. In contrast, our approach is designed to provide interactive support for secure programming and takes full advantage of developers' contextual knowledge of the application under development.

There are two possible strategies for when to perform input validation. One is to validate a variable containing untrusted input when it is used in a critical operation, such as a database update, insertion, or deletion. The other is to validate an untrusted input as soon as it is read into an application-declared variable. A major disadvantage of the first strategy is that it is not always possible to predict what operations are critical, and thus fails to validate input when the context of the application evolves. ASIDE promotes what we believe is the best practice for secure programming, validating untrusted inputs at the earliest possible time [8].

Another issue is that untrusted inputs could be of composite data type, such as a List, where input types may be different for each element of the list. In the current ASIDE implementation, the developer is warned of taint sources of a composite type with a visually softer yellow marker as shown in Figure 3. ASIDE uses data flow analysis to track an untrusted composite object. As soon as the developer retrieves an element that is of primitive data type (e.g. `java.lang.String`), ASIDE alerts the need to perform input validation and/or encoding in the same manner as described above. Given that the element retrieval from a

composite data type is unbound, ASIDE leaves the marker shown in Figure 3 throughout the development to serve as a continual reminder.

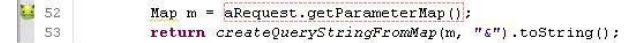


Figure 3: Visually softer marker that marks a tainted input with composite data type: Map.

ASIDE supports two types of input validation rules: syntactic rules and semantic rules. A syntactic rule defines the syntax structure of an acceptable input and is often represented as a regular expression. Examples include valid names, addresses, URLs, filenames, etc. Semantic rules depend on the application context. For example, restricting the domain of URLs, files under certain directories, date range, or extensions for uploaded files. They can also be used to validate inputs of special data types, such as certain properties of a sparse matrix. While validation rules can be declaratively specified by a SSG, a developer has the option to address the identified input validation issue by writing custom routines. This is then documented by ASIDE for later security audit. A developer can also easily dismiss ASIDE warnings if they are determined to be irrelevant. In any case, once the alert has been addressed, the corresponding red marker and text highlights will disappear in order to reduce programmer distraction and annoyance.

Another benefit of ASIDE is that it can help enforce secure software development standards across the organization. For example, a company may deploy a validation and/or encoding library and clearly define a set of trust boundaries for the purpose of performing input validation and/or encoding. Once appropriately configured with the defined trust boundaries and libraries, ASIDE can collect information as to where in the source code an untrusted input was brought into the system and what actions a developer took to address validation and/or encoding. ASIDE can also effectively supplement the security audit process by generating rules for traditional static analyzers. For example, once an untrusted input has been validated, customized Fortify rules can be generated to remove taints, thus avoiding unnecessary issues being generated during the auditing process. This can significantly reduce the time of a software security audit.

4. EVALUATION OF CODE REFACTORING

We now evaluate the effectiveness of ASIDE's vulnerable code detection for interactive code refactoring using an open source project of realistic scale. Thus, in this section, we focus on input validation and/or encoding vulnerabilities, as the ones currently supported by code refactoring. Our goals are to determine: (a) How effective is ASIDE at discovering exploitable software vulnerabilities and preventing them? and (b) What constitutes false positives for ASIDE? The significance of this evaluation is the use of real world cases that help us understand the effectiveness of ASIDE and provide guidance for further research.

4.1 Establishing a baseline using an open source project

We selected Apache Roller (a full-featured blog server) [30] release version 3.0.0 because it is one of the few mature Java EE based open source web applications. A Google search of

“powered by Apache Roller” yielded over 1.8M entries including sites such as blogs.sun.com. One of the co-authors, who is an experienced member of a SSG at a large financial service organization, performed a software security audit using his company’s practices to identify security vulnerabilities that are exploitable in Roller.

The audit process consisted of two parts: (1) automatic static analysis using Fortify SCA [34], and (2) manual examination of Fortify findings. Default Fortify rules were used, followed by manual auditing to eliminate Fortify findings that are not immediately exploitable. For each issue reported by Fortify, its source code was reviewed to:

- determine whether appropriate input validation/encoding has been performed;
- determine whether Fortify’s environmental assumption is valid. For example, in the case of log forging, whether the logging mechanism has not been wrapped in any way that prevents log forging;
- determine whether Fortify’s trust boundary assumption is valid. For instance, whether property files are considered to be trusted, and in this case, data from property files is untrustworthy;
- scrutinize input validation and encoding routines to make sure they are proper. For example, check if blacklist-based filtering is used. File, LDAP, DB, and Web all require different encoders or filters because different data schemes are used; and
- pay close attention to DOS related warnings (e.g. file handles and database connections) as resources may be released in a non-standard way. Often times, warning are generated even when resources are released in a **finally** block.

Roller 3.0.0 has over 65K lines of source code. Fortify reported **3,416** issues in **80** vulnerability categories, out of which, **1,655** issues were determined to be exploitable vulnerabilities. Table 1 summarizes the results of this audit process. Based on the evaluator’s experience, Roller’s security quality is at the average level of what he has evaluated. According to current work load estimate metrics of his enterprise, the analysis work reported here would amount to **2.5** person days.

Table 1: Results rendered by the industry security auditing process on Apache Roller version 3.0.0.

	Critical	High	Medium	Low
Fortify Issue Categories	8	18	2	52
Raw Issues	164	653	13	2,597
Exploitable Issues	37	397	0	1,221

ASIDE’s code refactoring is primarily aimed at preventing vulnerabilities resulting from lack of input validation and/or encoding. Out of the **1,655** Fortify issues that can be exploited in Roller, **922 (58%)** of them are caused by lack of input validation and/or encoding including most of the vulnerabilities from the critical bucket. The rest, mostly in the low security risk category, are related to failure to release resources (e.g. database connection) and other bad coding practices. Table 2 lists the details of the audit findings for

the **922** issues of input validation and/or encoding we will compare to ASIDE.

It is common that multiple Fortify issues share the same root cause of an untrusted input, referred to as a *taint source*. A single taint source may reach multiple *taint sinks*, exploitable API calls, and thus generates several different vulnerabilities. For example, a user-entered value might lead to a Log Forging vulnerability if it is inserted into a log, and a SQL Injection if it is used in an operation that executes a dynamic SQL statement.

The **922** Fortify issues are caused by **143** unique taint sources including both primitive data types (e.g. `java.lang.String`) and composite data types (e.g. `java.util.Map`). Variables requiring output encoding are always the result of a taint source. Thus, we exclude them in our analysis to avoid duplications.

Table 2: Detail results from security auditing against Roller using Fortify SCA.

Severity	Category Name	
Critical	Cross-Site Scripting: Persistent	2
	Cross-Site Scripting: Reflected	2
	Path Manipulation	19
	SQL Injection	11
Medium	Cross-Site Scripting: Persistent	31
	Denial of Service	4
	Header Manipulation	52
	Log Forging	252
	Path Manipulation	6
Low	Cross-Site Scripting: Poor Validation	6
	Log Forging (debug)	531
	SQL Injection	3
	Trust Boundary Violation	3
Total		922

4.2 Vulnerability Coverage of ASIDE

We then imported Roller into an Eclipse platform that has ASIDE installed and configured. ASIDE identified **131** of the **143 (92%)** exploitable taint sources. The remaining **12** cases involve JSP files and Struts form beans. The current ASIDE implementation does not cover these cases, but they could be easily handled in future implementations.

41 of the **143** are taint sources of composite data returned from APIs such as `org.hibernate.Criteria.list()` and `javax.servlet.ServletRequest.getParameterMap()`. ASI-DE performs dataflow tracking within the method where untrusted input is read. When a primitive value (e.g. `java.lang.String`) is retrieved from the composite data structure instance, ASIDE will raise a regular warning and provide assistance to validate and/or encode that input, as described in Section 3.

While we successfully identified tainted inputs of composite data types in Roller, in many cases, developers did not use the elements in that data object within the immediate method. Since ASIDE only currently performs taint tracking within the immediate method declaration, future implementations of ASIDE will be expanded to support taint tracking for composite objects beyond the scope of the immediate method declaration, which would then alert the programmer to all these primitive data type uses.

Our analysis also raised the issue of delayed binding. An

example of delayed binding is the access methods in a POJO (Plain Old Java Object). For example, `setBookTitle()` method of a Java class `Book.java` with a `bookTitle` attribute of `String` type. Binding of access methods to input streams can be delayed until after the program is completed. Thus, at the time of writing the program, there is no strong reason to believe the input is untrusted. After completion of the application, an integrator may bind an untrusted input stream directly to a POJO, making the application vulnerable.

Delayed binding is a difficult problem for existing static analysis tools as well. If the binding specification (typically in XML format) is composed in the same IDE environment, which is usually the case for Java EE development, one could extend ASIDE to help developers discover input validation issues by resolving the binding specifications. Further research is needed on the best approach to address delayed bindings in ASIDE.

4.2.1 False Positive for ASIDE

As we just demonstrated, ASIDE had good coverage of the input validation/encoding issues in Roller. In this section, we discuss the additional warnings that ASIDE generated. In analyzing false positives, we only look at taint sources of primitive data types. Taint sources of composite types are accounted for when elements of the composite object are retrieved and treated as a taint source of a primitive data type.

ASIDE reported 118 taint sources of primitive data types that were not identified as exploitable Roller vulnerabilities by the Fortify software security audit. 94 of them are cases that are not exploitable at the moment. For example, a taint source does not reach any taint sink. Failure to validate/encode the untrusted input may not be exploitable in the context of the current application. However, often times, such untreated inputs will eventually be used, and thus cause exploitable security vulnerability as the software evolves. Therefore, we believe it is still a good secure programming practice to validate/encode all untrusted inputs, regardless of whether they will reach a taint sink or not.

Figure 4 shows another example from Roller, where a tainted request URL is directly passed into an `InvalidRequestException` constructor, and eventually inserted into the error log. Fortify default rules do not acknowledge this code to be vulnerable. However, if the logs are viewed in a web-based log viewer such as a web browser, which is common in some organizations, this would allow an attacker to launch a Cross-site Scripting attack on the system administrator reviewing the log.

```

100     } else {
101         throw new InvalidRequestException("bad path info, "+
102             request.getRequestURL());
103     }

```

Figure 4: Untrusted input is logged through an Exception construction.

Thus, from a broad secure programming best practice perspective, we believe these 94 cases should be regarded as true positives, and ASIDE’s warnings should still be followed. However, from a circumscribed perspective of a specific application, they may be regarded as false positives.

The remaining 24 reported taint sources we regard as false positive, where inputs are used in ways that do not lead to

any recognized security vulnerabilities. These often involve inputs that are highly specific to the application context. For example, as illustrated in Figure 5, an input is tested to see if it equals to a constant value, determining the application flow.

```

179 // are we doing a preview? or a post?
180 String method = request.getParameter("method");
181 boolean preview = (method != null && method.equals("preview")) ? true : false;

```

Figure 5: Untrusted input is used for logic test.

Another such case is shown in Figure 6, where the input is cast into a `Boolean` value with only two possible outcomes: true and false, which will not result in any harm to the intended application logic.

```

125 if (request.getParameter("excerpta") != null) {
126     this.excerpta = Boolean.valueOf(request.getParameter("excerpta")).booleanValue();
127 }
128

```

Figure 6: Untrusted input is parsed into harmless Boolean value.

Because false positive rate often is positively correlated to accuracy, it is difficult to design a highly accurate tool with very low false positive rates. Both traditional analysis tools, such as Fortify SCA, and ASIDE will require manual inspection of warnings to eliminate false positives. However, our analysis of Roller suggests that for vulnerabilities due to improper input validation and/or encoding, ASIDE generates far fewer issues than Fortify, reducing the workload for both developers as well as software security auditors.

Additionally, we believe that it may take less effort to recognize and deal with ASIDE false positives compared to those generated by traditional static analysis. ASIDE’s warnings are generated while the developer is actively engaged in the programming process, making it easier to examine and understand the context of the warning. Moreover, with a click of a button on ASIDE’s resolution menu, the developer can dismiss a warning as false positive. In contrast, false positives generated by traditional analysis tools such as Fortify SCA are often dealt with by either software security auditors who typically do not have full application knowledge or by developers after the program was completed. In both cases, we believe it will take them longer to fully understand the impact of a particular warning generated by static analysis and to recognize it as a false positive. As excessive false positives could have a negative impact on the usability of any tool, further research is needed to understand how false positives in ASIDE impact developer behavior.

5. INTERACTIVE CODE ANNOTATION

Interactive code annotation [37] is a mechanism that helps developers to avoid more subtle security vulnerabilities where code refactoring is not feasible, such as broken access control and CSRF. Instead, programmers are asked to indicate where practices were performed, which is added as an annotation to the code. This serves as both a reminder to perform good practices, and enables further analysis and auditing. We first discuss interactive code annotation using an example of access control.

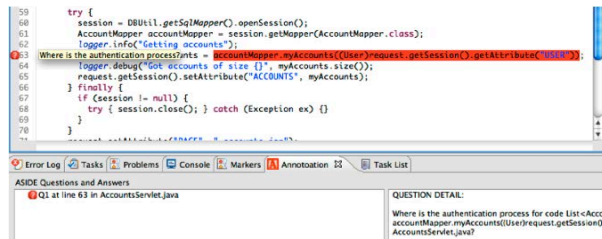
Consider an online banking application with three database tables: `user`, `account`, `account.user`, and `transaction`, where the tables `account` and `transaction` are specified as

requiring authentication in such a way that the subject must be authenticated by the key of the **user** table, referred to as an access control table.

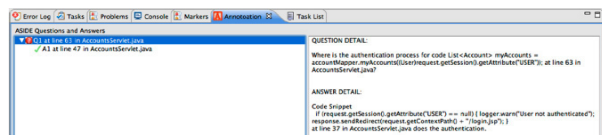
Figure 7(b) shows a highlighted line of code in the `doGet()` method of the accounts servlet, which contains a query to table **account**. ASIDE requests the developer to identify the authentication logic in the program, again by a red marker and highlighted text in the editing window. In this case, the developer highlights a test condition `request.getSession().getAttribute("USER") == null` as illustrated in Figure 7(b), which ASIDE saves as an annotation to the query code. The annotations can be reviewed and modified in an additional view as shown in Figure 7(b) and Figure 7(c), on which different information corresponding to different actions the developer has taken is displayed. Thus, the annotation process is seamlessly integrated into the IDE without requiring the developer to learn any new annotation language syntax.



(a) Developer identifies authentication logic (highlighted text) upon request from the ASIDE (see marker and highlighted text of Figure 7(b)) and annotates it.



(b) ASIDE issues a question for proper access control check that grants/denies the access to the highlighted data access operation. The detail of such request is displayed on the view called *Annotation* below the code editing window.



(c) The *Annotation* view adds the annotated information as a response to the developer's annotating of the access control logic in above Figure 7(a).

Figure 7: An example showing how ASIDE interactive code annotation works.

Several benefits can be derived from this annotation mechanism. First, the developer is reminded of the need to perform authentication, and the annotation process may help the developer to verify that authentication logic has been included. The developer has an opportunity to add intended access control logic should that be missing. Second, the logged annotations provide valuable information for code review. Third, heuristics-based static analysis can be performed to provide more in-depth analysis of the authentication logic.

We are still in the process of implementing interactive code annotation in ASIDE. Thus, the screenshots in this section are of our design, not a working prototype. We first discuss the issues in implementing code annotation, before moving on to our analysis of its potential effectiveness.

Our implementation of code annotation will take a knowledge-based approach relying on the specific structure of the target technology, initially Java servlet-based applications. We are first focusing on requesting programmers' annotations on access control logic. The key implementation issues are (a) how to identify the application context to prompt the developer for annotation and (b) what are the forms of acceptable annotations.

If we know the names of database tables whose access requires authentication, we can easily identify program statements that access these tables. Thus, we will provide a method for designers or a SSG to specify such tables as part of configuring ASIDE. However, such database access statements may not convey important application context, as they may be part of a utility library used by different web requests. For example, the **account** table may be accessed by the same access routine in multiple web requests (e.g. "customer account balance", and "electronic fund transfer"). Also, different web requests may require different access control logic (e.g. two-factor authentication for fund transfer). Therefore, we need to identify locations where application logic takes place by invoking database access operations.

For Java servlet-based web applications, access control annotations can be requested in the context of a web request: as in "where is the authentication logic for accessing the account table in the customer account balance request?", and "where is the authorization logic for accessing the account table in the electronic fund transfer request?" In this example each web request is implemented as a servlet. Therefore, we will start by tracking `doGet()` and `doPost()` methods within each servlet class (and this can be easily extended to include JSP pages)¹. They are referred to hereafter as *entry methods*. Through static analysis techniques, we will detect cases where an entry method leads to a statement accessing a database table which requires access control. The developer will then be requested to provide annotation of authentication in the context of the entry method, as illustrated in Figure 7(b).

Our annotation design satisfies the following requirements: (a) it consists of a set of logical tests (e.g. conditional tests in **if** statement, cases in **switch** statements), as shown in Figure 7(a), and (b) each test must be on at least one execution path, as determined by the control flow, from the start of the entry method to the identified table access statement.

Annotations may enable more in-depth static analysis. We are specifically looking into one type of execution analysis. For example, a broken access control may be detected if there is an execution path in the entry method leading to the database access without any identified access control checks along the path. We believe such an analysis can also be used to help prevent CSRF vulnerabilities. Of course, the accuracy of this analysis depends on the accuracy of the annotation. We now move to our evaluation of code annotation against real world cases.

¹Most popular Java frameworks also have structured entry methods, such as controller in Spring MVC

6. EVALUATE INTERACTIVE CODE ANNOTATION

We have conceptually tested this approach on real world open source projects. The security audit did not identify any broken access control or CSRF issues in Roller. Thus, we turned to bug tracking records to uncover previously discovered issues. Since Apache Roller only has a small number of fully documented security patches, we also included security patch information from Moodle [25], a PHP-based open source project for course management. A Google search of “powered by Moodle” yielded over **4.3M** sites including many large universities. A total of **20** fully documented security patches were found for the two projects. Four of them are due to improper input validation and/or encoding and can be addressed by ASIDE’s code refactoring support. Out of the remaining **16** vulnerabilities, **3** (1 broken access control and 2 CSRF) can be addressed by code annotation and the path analysis heuristics outlined above.

The broken access control issue is from Roller [31]. The authenticator, as illustrated in Figure 8, gets a web request from the client side and checks to see whether the headers of the request are valid. If they are valid, it extracts the credentials and verifies the validity of them. If the credentials are valid, the program goes on to access protected data in the database. If the credentials are not valid, an exception will be thrown, thus preventing unauthenticated access. However, there is another path from the web entry point to the data access point when the headers do not conform to the expected format, as shown in the control flow diagram in Figure 8.

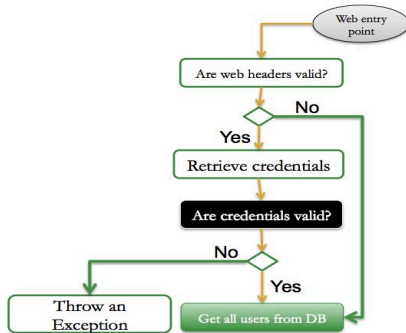


Figure 8: Control flow diagram of how an authentication request is processed.

Applying ASIDE’s code annotation approach, when the application code accesses the protected database resource to get all users’ information, ASIDE would prompt a request for proper access control logic on the path from the web request to the data access method call. Considering that the question should be raised on a transaction level, line 52 in the Servlet processing the request would be highlighted, as shown in Figure 10. In this case, the developer could easily identify the access logic as the logic tests which lie in the method invocation `verifyUser(username, password)` in `BasicAuthenticator.java`, highlighted in Figure 9. In this case, there are three tests to be annotated.

Based on the developer’s annotation, ASIDE would be able to construct a control flow graph, as illustrated in Figure 10, that has one path from a web entry point to a data

```

465 protected void verifyUser() throws HandlerException {
466     try {
467         UserData user = getRoller().getUserManager().getUserByUsername(getUserName());
468         if (user != null && user.hasRole("admin") && user.isEnabled().booleanValue()) {
469             // success! no exception
470         } else {
471             throw new UnauthorizedException("ERROR: User must have the admin role to use
472     }
473 }
  
```

Figure 9: Annotate access control logics.

access point with an annotated access control check on it, while another path from the same entry point to the same data access point has no access control check on it. Therefore, ASIDE would be able to provide a warning to the developer of a potential broken access control vulnerability.

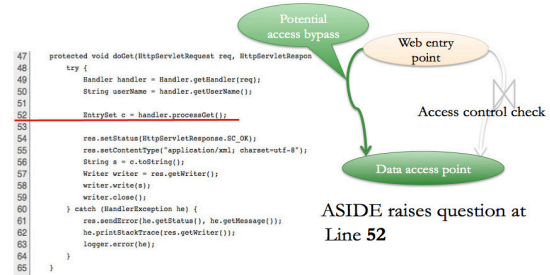


Figure 10: Java Servlet code for processing authentication request (left) and Access control check graph that involves processing the request (right).

Two CSRF vulnerabilities were recorded in Moodle’s bug tracking system. An effective way to prevent CSRF is to assign a random token to each instance of a web page that changes the state of the application. The token will be verified by the server before the application state is changed. The Moodle project is clearly well aware of the CSRF problem and implemented this strategy as a set of standard utility functions, simplifying developers’ tasks. However, developers still missed using this CSRF protection, introducing serious vulnerabilities into the software.

Through code annotation, ASIDE can be designed to remind developer of places where CSRF protection is needed, such as web transactions that change application states. Whenever a form submission/web request contains an operation to update (add, delete, modify) database entries, the form submission needs to be checked for CSRF. We describe one of the CSRF cases in Moodle, MSA-08-0013 [26], in detail; the other example is similar. This CSRF vulnerability is based on editing a user’s profile page, *edit.php*. Since ASIDE is currently being designed for the Java EE environment, we recast this example in equivalent Java terms.

The edit function would have a web entry point such as in a Servlet. Function `update_record()` is called, as highlighted in Figure 11, to update selected database tables through database operations. ASIDE would prompt the developer to annotate a logic test that implements CSRF protection. The request for annotation would be made at the line where `update_record()` is called. In this case, CSRF protection was omitted, so the programmer would be reminded to add such protection. Once the CSRF protection is added with appropriate annotation, ASIDE will apply the path analysis heuristics to further check for possible logic errors that may bypass the CSRF protection.

We have not yet conducted an analysis on false positives

```

$userold = get_record('user', 'id', $usernew->id);
if (update_record('user', $usernew)) {
    if (function_exists('auth_user_update')) {
        // pass a true $userold here
        if (!auth_user_update($userold, $usernew)) {
            // auth update failed, rollback for moodle
            update_record('user', $userold);
            error('Failed to update user data on external auth: '.$user->auth.
                ', See the server logs for more details.');
```

Figure 11: A snippet of source code of the web transaction that changes user profile.

for code annotation since we have not completed a prototype of ASIDE's code annotation functions, however, we would like to point out a case where false positives may appear. It is not uncommon for a web application to access different tables of a database to respond to one single request. Without context, ASIDE would raise the same questions for each and every function call that changes a table, and thus may produce false positives. Further research is needed to make ASIDE more contextual and intelligent about asking questions in these cases.

7. DEVELOPER BEHAVIOR STUDY

The previous evaluations focused on the ability of ASIDE to detect or fix vulnerable code. However, ASIDE must be designed in a way that fits naturally into a developer's work environment in order to be successful. To gain an understanding of programmers' reactions towards real-time secure coding support, we conducted a pilot user study involving 9 graduate students from our college using our current implementation of ASIDE in a controlled experimental setting. All 9 participants were recruited from a Java based web application development course in the Spring 2011 semester. As part of the course, students were briefly introduced to basic secure web programming techniques such as input validation and encoding. However, project grades were assigned only based on functional requirements, not on secure coding practices.

Part of the students' course work was to build an online stock trading system incrementally over 4 projects throughout the semester. Our study focused on the last increment of this project where students were asked to implement functionality including add a banking account, display stock details, make a buy/sell transaction, and display transaction history. Students added these functions on top of their existing work artifacts, which included static web pages, login, logout, and register functionalities.

We asked students to come to our lab and work on their assignment for 3 hours, using Eclipse with ASIDE. ASIDE was implemented as described in Section 3, generating input validation and encoding warnings. Participants were given a brief tutorial of ASIDE, then told to work on whatever aspects of their code they wanted, using ASIDE as they wished. We recorded all their interactions with ASIDE through screen recording software, and logged all interactions with the ASIDE interface. Immediately after the 3-hour study, participants were interviewed about their experiences of using ASIDE. Participation was voluntary and not related to class grades; students were compensated with a \$30 gift card for his/her time.

Our goals with this study were to evaluate the usability of ASIDE and determine: (a) do users pay attention to warn-

ings? (b) do they use code refactoring functions, and (c) do warnings and code generation help developers produce more secure code?

Figure 12 depicts the results of all 9 participants. Over all 9 participants, **101** distinctive ASIDE warnings were generated, resulting in **11.2** warnings per participant on average. **83** warnings were clicked by participants, **9.2** for each participant on average.

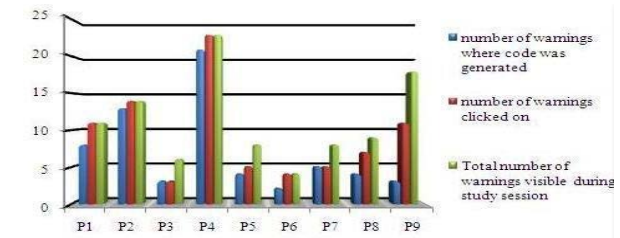


Figure 12: Metrics and results from 9 participants.

Out of the **83** warnings clicked, **63** were addressed (7 per participant on average) by clicking on one of the validation rules provided by ASIDE, leading to validation/encoding code being generated. The remaining warnings were dismissed by participants. All participants used ASIDE to generate validation routines in their development. None of them wrote any customized validation or encoding routines. Thus, ASIDE's code refactoring was effective in helping students write more secure code, even though they were not required to do so.

Multiple factors explain why certain warnings were not clicked or acted upon. Some of the warnings were generated when the participant wrote debugging code, which was soon deleted. Perhaps students ignored security warnings on code that they knew was transient. Other cases have to do with a bug in the version of ASIDE used, which falsely warned participants of the need for output encoding. We noticed that participants learned this warning was a false positive after one or more encounters, and then ignored those warnings thereafter. However, at least in this study, the presence of a false positive did not seem to cause the participants to not pay attention to other ASIDE warnings. Thus, in cases where participants encountered false positives, they were able to recognize them quickly and use the options provided by ASIDE to dismiss the false positives.

We transcribed all interviews into text via Inqscribe [15], and coded the transcripts using Atlas.ti [2] to identify general themes and interesting cases. All but one of the participants indicated that if they were not given ASIDE, they would not have been aware that some of the inputs needed to be validated before being used. Furthermore, they unanimously agreed that they would not write their own validation routines should ASIDE not provide them in the context of their assignment. As one participant stated:

[P1] That (The warning design of ASIDE) was good because hadn't it prompted me, I wouldn't have to realize I have to inspect those input values.

No one expressed that the ASIDE warnings were annoying, and all seemed to have faith that using ASIDE would make their code more secure.

[P4] No (it does not bother me). It gives me warnings so that I can write secure code.

So overall, 8 out of 9 participants expressed a positive impression of ASIDE, with one being neutral. This study demonstrated that ASIDE was usable by our participants. They were able to utilize ASIDE quickly, in the context of their own development, potentially improving the security of their code. We plan to expand upon this study to examine the use of ASIDE by professional developers to more deeply investigate the impact on programmer behavior in a variety of contexts.

8. RELATED WORK

Research into tool support for software security focuses heavily on machine-related issues, such as technique advancements for vulnerability detection effectiveness, accuracy, and vulnerability coverage, with very little concern with human factors issues. The two prominent techniques are static and dynamic program analyses. Static analysis typically is based on taint tracking [27, 17, 21, 7] and dynamic analyses are often based on model checking [14, 9, 22] and symbolic execution [6, 10, 32]. As both approaches have their advantages and disadvantages, a variety of work has explored the combination of these two techniques in an attempt to achieve better performance [3, 23, 13].

Software developer education and training has also been directed at achieving better software security. Most efforts have been spent on developing educational material and guidelines for the best secure programming practices [5, 16, 1, 18, 35]. However, the mere existence of such abundant information does not guarantee its use by programmers [38]. Our work takes human factors into consideration and fills this missing gap, complementing program analysis tools to help developers write more secure code.

Our work is in part motivated by studies on human errors [29], and programmer errors in particular [20], which demonstrate that programmer errors contribute a great deal to software flaws. Many such errors are caused by three types of cognitive breakdowns: *skill-based breakdowns* where programmers fail to perform routine actions at critical times; *rule-based breakdowns* where programmers fail to do an action in a new context; and *knowledge-based breakdowns* where a programmer's knowledge is insufficient. We believe that ASIDE's contextualized reminders can help address many of these issues by filling in knowledge gaps and reminding programmers of secure programming issues within their current context.

Prior work on code annotation (e.g. [24]) used textual extensions of programming languages, such as C and Java, which demands developers to learn and use yet another type of language. In contrast, our approach leverages GUI support to make the process more intuitive, direct and easy to use. Furthermore, soliciting security information from the developer through annotation could open up new ways to detect software vulnerabilities.

9. CONCLUSIONS

The main contribution of our work is to augment IDE tools to intelligently recognize software security issues and remind and assist developers with possible mitigation actions. We evaluated this approach against mature open source projects. Our results suggest that this approach is

effective in detecting and helping prevent common types of web application vulnerabilities, such as lack of proper input validation and/or encoding, CSRF, and broken access control.

No single tool is capable of detecting all software vulnerabilities, and ASIDE is no exception. However, we believe that ASIDE can be most effective in supplementing static analysis tools and increasing the productivity of current best software security practices. For input validation and encoding issues, which often constitute the largest percentage of issues in a typical web application, ASIDE can significantly reduce the number of issues generated by static analysis by helping programmers to prevent them in the first place. Thus, ASIDE can help improve software security and reduce the number of security fixes needed after static analysis, as well as saving time for the software security audit by 50%, based on our Roller case study.

Our user study results suggest that ASIDE is effective in helping novice developers/students to write more secure code when using code refactoring. They appear to pay attention to ASIDE warnings and follow ASIDE advice to perform input validation and/or encoding. Further studies, including studies involving experienced developers, are needed to show whether ASIDE can improve developers' understanding and practice of secure programming in more contexts. Preliminary evidence appears to suggest that users can quickly recognize false positives in ASIDE and take easy action to dismiss them. More research on false positives and how they impact on developers using ASIDE is needed.

Finally, our approach also provides a tool platform to provide additional support for the secure software development lifecycle in the areas of enforcing standards, information collection for secure coding metrics, and capturing developer rationale for code review or in depth program analysis. In an enterprise environment, ASIDE can serve as a medium that communicates a Software Security Group's secure coding knowledge to developers. As we continue the development of ASIDE, we plan to further investigate the use of these features and how they can contribute to secure applications.

ACKNOWLEDGMENTS This work is supported in part by a grant from the National Science Foundation 0830624, and a research/educational license from HP Fortify Inc. We would like to thank Will Stranathan for his contribution in the formation of this idea and critiques and suggestions on the initial prototyping of ASIDE. We also want to thank all our participants for their time.

10. REFERENCES

- [1] S. Ardi, D. Byers, P. H. Meland, I. A. Tondel, and N. Shahmehri. How can the developer benefit from security modeling? In *The Second International Conference on Availability, Reliability and Security, 2007*, pages 1017–1025, april 2007.
- [2] Atlas.ti. Atlas.ti, 2011. www.atlasti.com.
- [3] D. Balzarotti, M. Cova, V. Felmetzger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 387–401. IEEE Computer Society, 2008.
- [4] M. Bishop and B. J. Orvis. A clinic to teach good programming practices. In *Proceedings from the Tenth*

- Colloquium on Information Systems Security Education*, pages 168–174, June 2006.
- [5] CERT. CERT Secure Coding, 2011. www.cert.org/secure-coding.
 - [6] A. Chaudhuri and J. S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM conference on Computer and communications security, CCS '10*, pages 585–594. ACM, 2010.
 - [7] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2:76–79, November 2004.
 - [8] B. Chess and J. West. *Secure programming with static analysis*. Addison-Wesley Professional, first edition, 2007.
 - [9] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX conference on Security, USENIX Security'10*, pages 10–10. USENIX Association, 2010.
 - [10] X. Fu and K. Qian. Safeli: Sql injection scanner using symbolic execution. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications, TAV-WEB '08*, pages 34–39. ACM, 2008.
 - [11] B. C. G. McGraw and S. Migués. Building security in maturity model, 2011. www.bsimm2.com.
 - [12] 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, ESSoS '09*, pages 75–90. Springer-Verlag, 2009.
 - [13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web, WWW '04*, pages 40–52. ACM, 2004.
 - [14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Verifying web applications using bounded model checking. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 199–, Washington, DC, USA, 2004. IEEE Computer Society.
 - [15] Inqscribe. Inqscribe, 2011. www.inqscribe.com.
 - [16] S. Institute. SANS Institute, 2011. www.sans.org.
 - [17] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6 pp. –263, may 2006.
 - [18] K. Karppinen, L. Yonkwa, and M. Lindvall. Why developers insert security vulnerabilities into their code. In *Proceedings of the 2009 Second International Conferences on Advances in Computer-Human Interactions, ACHI '09*, pages 289–294. IEEE Computer Society, 2009.
 - [19] D. E. Knuth. The errors of tex. *Softw. Pract. Exper.*, 19:607–685, July 1989.
 - [20] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *J. Vis. Lang. Comput.*, 16:41–84, February 2005.
 - [21] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.
 - [22] M. Martin and M. S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium*, pages 31–43. USENIX Association, 2008.
 - [23] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, 2005.
 - [24] Microsoft. Microsoft SAL Annotations, 2011. <http://msdn.microsoft.com/en-us/library/ms235402.aspx>.
 - [25] Moodle. Moodle, 2011. <http://moodle.org>.
 - [26] Moodle. MSA-08-0013, 2011. <http://moodle.org/mod/forum/discuss.php?d=101405>.
 - [27] G. Naumovich and P. Centonze. Static analysis of role-based access control in j2ee applications. *SIGSOFT Softw. Eng. Notes*, 29:1–10, September 2004.
 - [28] OWASP. ESAPI Validator API, 2011. http://owasp-esapi-java.googlecode.com/svn/trunk/_doc/latest/org/owasp/esapi/Validator.html.
 - [29] J. Reason. *Human Error*. Cambridge University Press, Cambridge, UK, 1990.
 - [30] A. Roller. Apache Roller, 2011. <http://roller.apache.org>.
 - [31] A. Roller. ROL-1766, 2011. <https://issues.apache.org/jira/browse/ROL-1766>.
 - [32] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, Mar 2010.
 - [33] H. Sharp, Y. Rogers, and J. Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2 edition, 2007.
 - [34] F. Software. Fortify SCA, 2011. <https://www.fortify.com/products/fortify360/source-code-analyzer.html>.
 - [35] B. Taylor and S. Azadegan. Moving beyond security tracks: integrating security in cs0 and cs1. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education, SIGCSE '08*, pages 320–324. ACM, 2008.
 - [36] VERACODE. State of Software Security Report Volume 1, 2, and 3, 2011. <http://www.veracode.com/reports/index.html>.
 - [37] J. Xie, B. Chu, and H. R. Lipford. Idea: interactive support for secure software development. In *Proceedings of the Third international conference on Engineering secure software and systems, ESSoS'11*, pages 248–255. Springer-Verlag, 2011.
 - [38] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *Proceedings of 2011 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 161–164, 2011.