Supporting secure programming in web applications through interactive static analysis

Jun Zhu *, Jing Xie, Heather Richter Lipford, Bill Chu

Department of Software and Information Systems, University of North Carolina at Charlotte, Charlotte, NC 28223, USA

ARTICLE INFO

Article history:
Received 26 September 2013
Received in revised form 23 November 2013
Accepted 25 November 2013
Available online 5 December 2013

Keywords:
Secure programming
Static analysis
Interactive static analysis
Software vulnerabilities

ABSTRACT

Many security incidents are caused by software developers’ failure to adhere to secure programming practices. Static analysis tools have been used to detect software vulnerabilities. However, their wide usage by developers is limited by the special training required to write rules customized to application-specific logic. Our approach is interactive static analysis, to integrate static analysis into Integrated Development Environment (IDE) and provide in-situ secure programming support to help developers prevent vulnerabilities during code construction. No additional training is required nor are there any assumptions on ways programs are built. Our work is motivated in part by the observation that many vulnerabilities are introduced due to failure to practice secure programming by knowledgeable developers. We implemented a prototype interactive static analysis tool as a plug-in for Java in Eclipse. Our technical evaluation of our prototype detected multiple zero-day vulnerabilities in a large open source project. Our evaluations also suggest that false positives may be limited to a very small class of use cases.

Introduction

Many computer security problems are caused by software vulnerabilities, software flaws that can be exploited by attackers and result in data and financial loss as well as inconvenience to customers. There are many causes for such vulnerabilities, but the most common ones are flaws introduced by developers during program construction. A number of vulnerabilities can be addressed with relatively straightforward coding practices, referred to as secure programming [1].

such as performing validation on user input to prevent various forms of injection attacks. While there are a number of programming practices and libraries to help prevent vulnerabilities, developers must remember to utilize them, and do so correctly.

While secure programming practices have been well documented [2–6], developers continue to make the same mistakes. For example, in 2011 Veracode analyzed over 6750 web application builds across 40 different industry sectors. A third of these applications had SQL injection vulnerabilities which had been documented over a decade ago and could be simply addressed by parameterized SQL statements [7]. The vast majority of vulnerabilities found in the Veracode study were caused by ignoring proper secure programming practices and could be corrected relatively easily.

The most commercially successful method of detecting security vulnerabilities in code is through static analysis (e.g.
Fortify SCA [8], Veracode [9], and Coverity [10]). However, these tools require special training and often are not used by regular developers, thus requiring additional steps in the software development process to find and remove vulnerabilities. Performing static analysis effectively to detect software vulnerabilities often requires application-specific knowledge for several reasons. First, static analysis often generates many false positive warnings. For example, default rules in static analysis tools warn developers to validate untrusted input, regardless of whether proper input validation code has been added. To suppress these warnings in cases when input validation functions have been provided, customized rules must be written by people knowledgeable in using static analysis tools as well as understanding application-specific logic. Second, applications often must satisfy certain security invariants, e.g., access control rules. Such invariants are by nature application-specific. Again customized rules must be written in order for static analysis tools to detect missing security invariants. As a result, using static analysis to detect software vulnerabilities can be expensive in practice because it requires close collaborations between software developers and software security specialists trained to use static analysis tools. The full benefit of using static analysis to detect software vulnerabilities is often not realized because of this high cost.

Programmer errors, including security ones, are unavoidable even for well-trained programmers. One major cause of programming mistakes is software developer’s heavy cognitive load dealing with a multitude of issues, such as functional requirements, runtime performance, deadlines, and security [11–13]. Consider Donald Knuth’s analysis of 867 software errors he made while writing TEX [14]. It is clear from his error log that some of these errors could have made TEX vulnerable to security breaches. The following quotes illustrate 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” [14].

We believe that by providing helpful warnings to developers in-situ and gathering application-specific knowledge from them, during their programming tasks, we can more easily detect and resolve security vulnerabilities. In this paper we describe performing interactive static analysis on patterns of code leading to vulnerabilities, and reminding and assisting developers in resolving them within their development environment. We will demonstrate and validate our approach with a prototype implementation. The most important benefit of our approach is a mantra for software development: finding and fixing problems earlier in the development life cycle is less costly and less time consuming. In addition, by interacting with those most familiar with the application-specific context, the analysis techniques can be more focused and customized to that context.

**Overview of interactive static analysis**

We provide an overview of interactive static analysis using the motivating example illustrated in Fig. 1. It is a code snippet adapted from a Java servlet-based weblog management application. Method `doPost()` is one of the entry points of this application. It creates a weblog entry. In a typical default security rule set for static analysis in Java, API `request.getParameter()` as used on line 3 and line 4 is designated as a taint source, as it reads in untrusted input. Database operation APIs, such as `statement.executeUpdate()`, are designated as a data flow sink, as it modifies the database. Many such database operation APIs are privileged operations and require access control checks before invocation. Because method call `DBUtil.updateWeblogEntry()` used on line 15 calls `statement.executeUpdate()` in its implementation, `DBUtil.updateWeblogEntry()` may be considered as a data flow sink as well as being a privileged operation. Good secure programming practice calls for all input parameters to this method be validated and access control checks must be provided.
Applying static analysis to this program, one may find two issues. First, untrusted inputs are propagated from the data flow taint sources on line 3 and line 4 to the data flow sink on line 15, making the code vulnerable to injection attacks. These cases can be found easily by most static analyzers using default security rules. However, in this case, they are false positives as input validations have been performed on lines 7 and 8. To suppress these false positives, customized rules must be written. Second, there exists an execution path from the entry method `doPost()` on line 1 to a privileged operation `DBUtil.updateWeblogEntry()` on line 15. If an access control check is not performed, there would be a broken access control vulnerability. Performing this type of static analysis requires intimate knowledge of the application logic and is typically not performed by default in static analysis tools.

Our approach can support vulnerability detection by the developer by integrating static analysis into the Integrated Development Environment (IDE) as a plug-in, facilitating the two-way interaction between static analysis and the developers. **Plug-in to Developer:** the plug-in analyzes the source code in the background and reminds developers about vulnerable code through instant security warnings in-situ. Developers could interact with the warnings, get the contextual explanations about the vulnerable code, and resolve them with the assistance from the plug-in. **Developer to plug-in:** the developer is prompted by the plug-in and asked to identify and annotate application-specific logic critical for security, which we named interactive code annotation [15]. The annotations are then leveraged to perform additional static analysis, which helps discover vulnerabilities and reduces false positives without the intervention by someone with special training in static analysis to write customized rules.

Developers are not required to have any knowledge of static analysis, nor are they security experts. However, they do have to be familiar with basic security concepts (e.g. access control, encryption) as well as basic secure programming techniques (e.g. input validation, prepared SQL statements). We assume interactive support for secure programming occurs in the context of an organization which values secure programming and wants to support sharing secure programming knowledge and standards among development teams. An interactive static analysis tool can be configured by an organization’s software security group (SSG), which is responsible for ensuring software security as identified by best industry practice [16]. Thus, the SSG could use interactive static analysis to communicate and promote organizational and/or application-specific programming standards. For example, an SSG may list operations that are considered privileged (e.g. inserting a weblog into the database) and that the application must provide an access control check. Another benefit of an interactive static analysis tool is that it can provide logs of how security considerations were addressed during program construction, which are very valuable information for more effective code review and auditing.

Our prototype implementation of interactive static analysis, ASIDE (Application Security in IDE) is a plug-in for Java in Eclipse. ASIDE is driven by a set of security specifications describing patterns of code that may be of concern from a secure programming perspective. This specification language can be used by a SSG with little special training. For example, one can specify database tables and operations that require access control checks (e.g. inserting into the weblog table), or an API that reads untrusted data. ASIDE continuously monitors changes in the Eclipse workspace to detect these code patterns. Much of the static analysis task can be performed incrementally by a separate background thread and thus will not block the developer from normal code editing or other interactions with the IDE.

When a specified code pattern is detected, developers are given in-situ warnings of potential vulnerabilities alongside the code they are working on, as well as optional explanations and suggestions to resolve those vulnerabilities. Fig. 2 shows a screenshot of the motivating example within Eclipse, in which method invocation `DBUtil.updateWeblogEntry()` on line 37 is highlighted in red and flagged with an ASIDE warning because this method updates the weblog. By highlighting this method, ASIDE is requesting that the developer annotate the code performing the access control check for this privileged operation. The annotation can be completed entirely with intuitive mouse-based point-and-clicks. Hovering over the...
message, as shown in Fig. 3, the developer is given a brief explanation of what is requested with an optional link to read more explanations. A developer can dismiss the warning with the option “Click me to remove this warning” if he considers the warning as a false positive.

By selecting the option of “Click me to annotate a control logic” the developer can highlight code that performs access control for this database operation. The annotated code will be highlighted in green, the flag with the warned sensitive operation will turn light brown, and the highlighting color will turn to yellow, as shown in Fig. 4. In this case the access control check involves two Boolean tests: (a) checking to make sure a user is logged in and (b) the user has the right to write to this particular weblog.

Key design issues for interactive annotation include the following: (a) Where to prompt the developer for annotation, and (b) what are the characteristics of acceptable annotations. Requesting annotation at the place where privileged operations are invoked may not always be the best place. For example, a utility method may be written to update an account balance. This utility method may be utilized by different use cases (e.g. “fee payment” and “fund transfer”). Requesting annotation for access control logic for updating the balance may not provide adequate application context (e.g. “fund transfer” may require two factor authentication for example). A much more reasonable point of annotation is at the top level of a use case. In the context of web applications, a use case is most likely to correspond to a web entry method. Therefore, in
the illustrated example above, the request for annotation is presented at the level of the web entry method `doPost()`. The warning is placed beside a method call, `DBUtil.updateWeblogEntry()` on line 37 in Fig. 2. This method invocation indirectly performs the privileged operation.

Acceptable annotations must satisfy the following requirements. First, they must consist of a set of logical assertions (Boolean expressions in if statements, cases in switch statements, or program assertions), or method invocations. Second, the annotated code must be on the execution path from the entry method to the warned sensitive operation. In our running example, all annotations are Boolean expressions as illustrated on lines 34, 35 in Fig. 4.

Interactive annotation could also be used to request annotation for validation or sanitization routines in addition to access control logic. For example, in the motivating example of Fig. 1, the developer will be prompted on line 15, `DBUtil.updateWeblogEntry(author, validatedTitle, validatedContent, pubTime)`, and asked to annotate validation routines for the parameters used in the method call. In this case, the developer performs input validation on lines 7 and 8, so he is expected to annotate the method calls `Util.developerDefinedValidationMethod(title)` and `Util.developerDefinedValidationMethod(content)`.

To summarize, our interactive static analysis framework generally aims to address two major categories of vulnerabilities: lack of enforcing security invariants and lack of sanitization of untrusted data. Our approach as described above can address seven out of OWASP Top Ten web application vulnerabilities [17]: Injection, Cross-Site Scripting, Unvalidated Redirects and Forwards, which are all caused by lack of sanitization of untrusted data; Broken Authentication, Insecure Direct Object Reference, CSRF, and Missing Functional Level Access Control which are all caused by lack of enforcing security invariants.

Prototype implementation and evaluation

In the previous section, we illustrated the interactive static analysis framework with a code example and our prototype implementation, ASIDE. In this section, we will first describe the underlying static analysis mechanism we implemented in ASIDE, called path coverage static analysis, and then describe a set of evaluations we performed.

Path coverage static analysis

The focus of our static analysis component is on detecting vulnerabilities originating from improper untrusted data handling and failure to maintain security invariants. Improper data handling means that untrusted data is propagated to data flow sinks without sanitation. Failure to maintain security invariants means program execution could reach security-sensitive operation on the control flow path without checking security invariants. A common characteristic for both categories of vulnerabilities is that proper security requirements are not enforced on at least one execution path. Therefore our static analysis mechanism addresses what we refer to as the “path coverage problem”: security requirements, data sanitization or security invariant checks must be enforced on all execution paths in a coherent manner. We will first introduce a few concepts and definitions used in our path coverage analysis, and then describe the path coverage analysis with examples.

Path: An execution path from a program entry point to a particular operation of interest. Algorithms for enumerating execution paths have been well studied.

Security entry point: One of the two points in a program: (1) entry point of a program, e.g. `doPost` method, which is a common entry point for Java Servlet-based web applications; (2) an API call where untrusted data is read into the program, which is also referred to as a taint source.

Security sinks: Security-sensitive operations or events, for example include accessing sensitive information (e.g. updating a customer database table), or performing an operation that requires authorization (e.g. switching a relay on a power grid).

Security path: A security path is an execution path with a security entry point and a security sink.

Security path requirements: There are often security-related requirements related to execution paths, such as sanitization of untrusted data or enforcing security invariants. Security path requirements are typically part of the software requirements. For example, a web application must perform an access control check before updating the customer database table, and must also perform checks to prevent CSRF attack against updating the customer table. Note that security path requirements are always given with respect to security sinks.

Security code instance: In our approach code annotation requests the developer to interactively annotate fragments of code that fulfill security path requirements. We refer to these code fragments as security code instances.

Generic principle of path coverage analysis: The generic principle that guides our interactive static analysis is, for a given security sink, there is at least one security code instance for each security path leading to it.

We use the following running example of a Java-based online banking application to illustrate the generic principle of path coverage analysis. Suppose this application has three database tables: `user`, `account`, and `transaction`. The application consists of a set of use cases such as `login`, `bill pay` and `view transactions`. Security sinks are accesses (read, write, delete, or update) operations to one of these tables. We denote the set of sinks as S. Security path requirements for this application may include (a) all untrusted data involved in execution paths to any of these sinks must be sanitized; (b) access control must be performed on all execution paths leading to any of these operations; (c) CSRF prevention must be provided for insert, delete, or update operations on these tables.

Various expressions such as method invocations and Boolean expressions could be used as security code instances to satisfy security path requirements. At this stage of our work we assume all the security invariants are enforced in the form of Boolean expressions. For each sink s in S, there is a set of Boolean expressions used to enforce security invariants \( I(s) = \{C_1, \ldots, C_n\} \). Each \( C_i \) is a Boolean expression. These expressions are part of the conditions that determine security invariants. For example, Fig. 5 shows a flow chart representation of a Java code fragment implementing access control to the update operation on the account table. In this example, the sink, s, is in the block “update account information,” highlighted in yellow. \( I(s) \) consists of the set of Boolean expressions highlighted in green: “User authenticated”, “User owns this account”, and “User is an authorized bank employee”. The exact logic combinations of these Boolean expressions to
implement security invariants are determined by the execution flow.

For the example account update use case, enforced security invariants to access customer accounts are either (a) the user is logged in and the user is the owner of the account, or (b) the user is logged in and the user is an appropriately authorized bank employee. A key insight of the path coverage analysis approach is to detect conflicts in the way security invariants are being checked along different execution paths in the same application. These conflicts often lead to common security vulnerabilities.

Input to path coverage static analysis is an application represented as a set of Abstract Syntax Trees (ASTs), a set of sinks S and a set of security entry points E. For each sink s in S, the set of Boolean expressions enforcing security invariants \( R(s) \) is obtained through interactive annotation.

The core of the path coverage analysis is to enumerate all execution paths from entry points to sinks, and analyze annotations along the paths. As an illustrative example, consider the flow chart in Fig. 5. In this case there are two execution paths: (a) web-entry \( \rightarrow \) User-authenticated \( \rightarrow \) User owns-account \( \rightarrow \) Update-account, and (b) web entry \( \rightarrow \) User authenticated \( \rightarrow \) User does-not-own-account \( \rightarrow \) User-authorized-bank employee \( \rightarrow \) Update account. In this case no vulnerabilities are detected because both execution paths are properly covered by Boolean expressions of annotated security invariants.

To illustrate the path coverage analysis with our prototype ASIDE, consider the motivating code example as shown in Fig. 4 in section ‘Overview of interactive static analysis’, two annotations (in green) have been made for the sink \( DBUtil.updateWeblogEntry() \) (in yellow), and there exists only one execution path from the entry point to the sink, doPost \( \rightarrow \) isLoggedInUser != null \( \rightarrow \) isLoggedInUser.isAuthorizedToAuthor() \( \rightarrow \) DBUtil.updateWeblogEntry(), on which there exist two annotations, and thus no vulnerabilities are detected.

We will further illustrate the path coverage analysis in section ‘Effectiveness in detecting software vulnerabilities’, in which we will illustrate how path coverage analysis is applied to detect a broken access control vulnerability in a large open source project.

Evaluations

Our evaluation of ASIDE consists of the following aspects. First, we performed a technical evaluation with a large open source project to demonstrate the effectiveness of interactive annotation in detecting security vulnerabilities. Second, we performed false positive analysis of interactive annotation with a real-world web project. Third, we evaluated the complexity of the code annotation task with a real-world web project showing that the annotation task is a relatively small burden for users. Finally we conducted an exploratory user study to understand how developers may actually use ASIDE in the field.

Effectiveness in detecting software vulnerabilities

To evaluate the effectiveness of interactive static analysis in detecting software vulnerabilities, we ran our prototype ASIDE on Apache Roller [18], which is a widely used weblog server open source project, and successfully discovered multiple zero-day Cross-site Request Forgery (CSRF) vulnerabilities [19], which exist in all the released versions of Roller, from 3.0 to 5.0. All the vulnerabilities were confirmed by our penetration testing as well as the Roller team. With our vulnerability report and resolution suggestions, the Roller development team has released Apache Roller 5.0.1 with the security fix [20]. Among the multiple CSRF vulnerabilities found, we choose one of them as an example to describe how interactive static analysis worked.

Fig. 6 shows a code snippets of the method \texttt{ActionForward save()} in \texttt{org.apache.roller.ui.authoring.struts.actions.WeblogEntryFormAction.java} from Apache Roller 3.0 [21]. Apache Roller 3.0 uses the Apache Struts I framework, under which the \texttt{save()} method is a web application entry point because of object inheritance. This method handles requests of new weblog entry creation. As shown in the Fig. 6, line 321 is highlighted with a red flag security warning for \texttt{weblogMgr.saveWeblogEntry(entry)} because it inserts a weblog entry into the database. ASIDE is requesting the developer to annotate program logic provided to check access control as well as prevent Cross-site Request Forgery. In this case, the Boolean expression \( \texttt{rses.isUserAuthorizedToAuthor(site) || (rses.isUserAuthorizedToAuthor(site) & & form.get_Status().equals(WeblogEntryData.PUBLISHED))} \) on lines 261 and 262 provides access control checks, but the developer is not able to locate and annotate the CSRF protection check. Failing to identify CSRF protection logic, the developer would have realized that a CSRF protection check needs to be added.

In addition to the CSRF vulnerabilities, ASIDE also detected an access control vulnerability in Apache Roller 3.1 [21], which has been confirmed in the bug tracking history repository of Roller [22]. Fig. 7 shows the screenshots of three methods that are related to the access control vulnerability. Method \texttt{doGet()} in Fig. 7a is a web entry method that handles all authentication web requests in Roller, in which it calls \texttt{Handler.getHandler(req)} on line 50. Method \texttt{Handler}. 

![Flow chart of Java code updating the account table.](image_url)
getHandler(req) invokes the authenticate() method shown in Fig. 7b. Method handler.processGet() on line 53 in Fig. 7a invokes a privileged database query through a sequence of method calls. Method authenticate() in Fig. 7b calls the method verifyUser(username, password) on line 51, which is shown in Fig. 7c.

Thus there exist execution paths from the entry method doGet() on line 47 of Fig. 7a to privileged database queries. ASIDE would flag a warning on line 53 in Fig. 7a requesting for annotation of control logic checks and also highlight the code in red. Access control checks are provided in verifyUser(username, password) by four Boolean expressions on lines 58, 61, 65, 68 as shown in Fig. 7c. Once the developer finishes annotating these control checks, ASIDE detects that there exists an execution path from the entry method doGet() to the sensitive operation handler.processGet() on line 53 in Fig. 7a without any annotated control checks on it, and thus reports an access control vulnerability. The vulnerability originates from the control flow in the method authenticate() shown in Fig. 7b. The access control checks are performed in method verifyUser(username, password) which is called on line 51 in Fig. 7b. However, once the Boolean expressions st.hasMoreToke() on line 42 or basic.equalsIgnoreCase('Basic') on line 44 is FALSE, the execution will not reach the method call verifyUser(username, password) on line 51, and thus no annotated control checks will be executed, which leads to an access control vulnerability.

**False positive for interactive annotation**

We performed evaluation of false positives of interactive static analysis on Tunestore, a web application developed by a professional developer for our department's web application development curriculum. It is written in Java using the Apache Struts I framework. It provides basic functionality of an online music store. A user can view all available songs in the store with or without logging in. Registered users can comment on songs, purchase songs and send gifts to friends.

Under Struts I every use case is implemented by an org.apache.struts.action Action class. For instance, the login functionality is implemented by class LoginAction.java. Each Action class has an execute() method serving as an entry point for the web application. Database operations are implemented with Java SQL APIs. ASIDE identified 24 cases where an entry point is able to reach a sensitive database operation on a control flow path. For example, Fig. 8 shows a screenshot of class LoginAction.java. On line 55 method execute() calls stmt.executeQuery(sql), a method that retrieves the username and password from the database. Since reading user information is a sensitive operation, ASIDE requests that the developer annotate the code for the access control check.

A false positive is a case where ASIDE requests annotation of access control logic where in reality access control is not necessary. The login case illustrated above is an example of a false positive because there will be no access control check for the login function. We manually inspected all 24 requests for annotation by ASIDE and identified 5 cases as false positives, as shown in Table 1. Use cases for Login and Registration each include two database operations. All false positives are involved with use cases such as login, logout and registration, in which accessing the sensitive users table does not require access control, since these are starting points where users will be granted access rights.

The false positive rate of (5/24) should be viewed in the context of the application. Tunestore is a small web application with only 16 use cases including login, logout, and registration. If the pattern we observed here, that false positives for access control annotation are limited to a small class of use cases (login, logout and registration), is confirmed in large web applications with many more use cases, false positive rates would be much lower.

**Complexity of the annotation task**

For interactive annotation to work, a developer must be able to easily identify code in response to annotations requested.
by ASIDE. We use the Annotation Distance (AD) metric to approximate the annotation complexity of the annotation task. It is based on the distance between the location where annotation is requested and the location of the code in response to the annotation request. The intuition is that the larger this distance, the more effort a developer must spend to...
search and look for the appropriate place to make an annotation.

Specifically AD equals to one when the annotated code can be found in the same method where the annotation is requested (the warning is displayed). AD equals to two in cases where code to be annotated can be found in another method within the same class as where annotation is requested. AD equals to 3 when the code to be annotated can be found in another class within the same package as where annotation is requested. AD equals to 4 when the code to be annotated can be found in another package. In cases where multiple annotations are made to a request (e.g. in the motivating example, there are two Boolean expressions in response to the request for annotation of access control logic), the corresponding AD is the sum of ADs for each annotation given.

We measured the AD score for each of the 19 positive cases identified by ASIDE in Tunestore. Some of them already have the appropriate checks implemented in code while the rest do not. Again as mentioned earlier for those cases where the developer cannot find the code to annotate, the annotation process will help developers to discover and prevent broken access controls by reminding them to implement that code. Table 2 shows the result of applying the distance metric to the 19 true positives.

All 19 cases require a user to login and maintain an active session. Tunestore enforces this type of access control check in the same method as where annotation is requested, thus, AD is one. In addition to checking whether the current session is active, ten cases need CSRF protection check, shown in the last column of Table 2. Tunestore does not provide any defense against CSRF, thus the developer would need to write the corresponding defense code in these ten cases, and then may annotate the checks for corresponding paths.

There is one case, viewing of songs, where AD is 4. The annotation request will be prompted on line 21 of ListAction.java as shown in Fig. 9, DBUtil.getCDsForUser(...), which contains database operations that require an access control check. This method is declared in a different package and thus according to our definition the AD is 4.

Our evaluation of Tunestore suggests that the annotation complexity of interactive annotation may be modest for web-based applications. Most access control logic checks are within the same class as where annotation is requested. These checks, however, are not of a direct and uniform format such as a Boolean conditional tests. Other forms of logic tests include WHERE clause of a SQL statement. ASIDE has implemented the most intuitive checks, which are conditional tests that are often implemented as Boolean test. ASIDE needs to be further enhanced to support annotation of other types of logic checks.

Developer behavior

Next we want to understand how developers actually behave in response to a request of interactive annotation. Because there are a large number of potential factors involved, we conducted an exploratory user study with the specific purpose of understanding critical variables to help us design more focused user studies in the future. Our exploratory study seeks to answer the following questions. (a) Even though most annotations can be found within the same method, can developers easily identify them? (b) If developers were NOT able to annotate correct
control logic, what were the major difficulties/obstacles preventing them from doing so? And (c) How can we improve ASIDE’s user interaction to better serve developers?

Our exploratory study observed student participants using ASIDE on functioning web applications, each with a number of vulnerabilities. We asked all participants to think aloud while they were reviewing source code in Eclipse with ASIDE warnings. During the observation, we also asked participants questions whenever we found an action worth further probing. We recorded all screens as well as conversations with participants for data analysis.

We recruited 8 student volunteers from 3 different sources. Three of them were from a web development course in our college. Two were from a web security course, which aims to teach students ethical hacking and secure programming. The other 3 students were from a highly selective security scholarship program at the college. Participants were at different experience levels with respect to web development and secure programming. The 3 from the web development course were the least experienced in programming with no prior exposure to secure programming. The 2 from the secure programming course were more experienced in programming, and were in the process of acquiring secure programming knowledge. The 3 from the scholarship program were constantly being exposed to a variety of security-related projects. In addition to being immersed in software security, all three of them had taken the secure web application course in current or prior semesters. All participants were first time users of ASIDE.

For the 3 participants recruited from the web development course, they worked on an online stock trading system, a project that is part of their course assignment. The rest of the participants worked on Tunestore described earlier. Prior to the study, the latter group of 5 students already had knowledge of Tunestore since they had been asked to find and fix certain vulnerabilities in it. We chose Tunestore for this latter group of participants because we wanted to simulate the environment where developers work on projects they are very familiar with. Even though this group of developers knew some security vulnerabilities in Tunestore, it was still their first experience with ASIDE.

We gave an introduction of ASIDE to all participants on a sample web application by addressing a warning through the 3 different options provided by ASIDE as shown in Figs. 2–4: read more about this warning; annotate a control logic; remove this warning. Then, we imported the participant’s project into a new workspace of Eclipse, and started screen and voice recordings. We let participants examine annotation requests shown in his/her code. Each session lasted from 30 to 40 min.

Our findings indicate that in order for a developer to provide the correct annotation, he/she must first understand and appreciate the underlying problem. In other words the developer must be able to understand why it is important to provide the annotation. The approach we provided to convey this information to the developer, an explanation web page, may not be adequate for some developers. We extracted three characteristics that heavily influenced participant behavior from our observations.

**Attacker mentality**

Having an established attacker mentality, the mindset of manipulating an application to perform unintended functions, is strongly correlated with performance of the annotation task. The three participants with limited understand of security and penetration testing failed to comprehend annotation requests. Moreover, none of them were able to identify whether a warning is a true positive or a false positive, or give any judgment as to whether the tool makes sense. For instance, participant P1 said:

[P1] *I don’t know what this user.isPrivileged( ) means.*

When asked whether she read the provided web page material explaining broken access control problems, participant P2 responded that she did read through it. Yet when confronted to explain what is the problem described in the page, she murmured and then switched the topic of discussion.

Participant P3 glanced quickly through the explanatory page explaining broken access control, but found no information that was relevant to his code. He dismissed the page as a result. He then failed to identify whether a warning was indicating a real vulnerability or not. He also failed to give reasons why he thought a warning was a false positive.

In contrast, the other 5 participants, who either had been or were taking a course on penetration testing and attacking/breaking web applications by exploiting code level vulnerabilities were able to make a quick and accurate decision on the false positives on both the login servlet and register servlet. Three of them succeeded in identifying lack of access control before changing an existing user’s password, while the other two were able to realize the issue after we mentioned the possibility of broken access control.

**Knowledge of attacks**

We find that knowledge of the attacks is another factor influencing developer understanding of annotation requests. For instance, someone who understands broken access control may not understand Cross-site Request Forgery and therefore not be able to provide annotation of code preventing

| Use case       | Entry point                                      | Sensitive operation                      | Reason                                      |
|----------------|-------------------------------------------------|------------------------------------------|---------------------------------------------|
| Login          | LoginAction.java:34:execute(...)               | LoginAction.java:55:executeQuery(...)    | Application logic does not require access control |
| Logout         | LogoutAction.java:29:execute(...)              | LogoutAction.java:41:executeUpdate(...)   |                                             |
| Registration   | RegisterAction.java:29:execute(...)            | RegisterAction.java:56:executeUpdate(...) |                                             |
|                | RegisterAction.java:29:execute(...)            | RegisterAction.java:65:executeUpdate(...) |                                             |

Table 1 Five false positive cases of ASIDE’s analysis results on Tunestore.
Cross-site Request Forgery. Among all 5 participants who possessed an attacker mentality, none of them immediately spotted the Cross-site Request Forgery (CSRF) vulnerabilities that were indicated by ASIDE warnings (part of the interview records are shown in Table 3).

Once we explained further, through discussions, of CSRF in the context of Tunestore, four out of the 5 participants successfully annotated the corresponding prevention logic following instructions provided by ASIDE.

Programming capability

Three of the 8 participants either had experience writing professional code or were about to become a professional developer. They were able to perform ASIDE annotations successfully and correctly on the first try. When asked about whether the phrase “Boolean control check” makes sense to them, they responded positively. For instance, P8 (a professional developer) expressed with confidence in the following dialog:

[Investigator] Do you consider it being easy and intuitive to find the right check?

[P8] I think so, I think if it’s your code and especially if this system works for the purpose of designing for it, for instance, if it’s identifying an issue here and I have to create a check in order to verify the statement, I don’t think finding the Boolean check is going to be the difficult part at all.

In contrast, the rest of the participants had trouble locating a Boolean control check for a warning even after they had been given detailed description of the potential vulnerability. The other 5 found it difficult to accurately explain the phrase “Boolean control check”.

Throughout the study, participants identified several user interface issues that can be improved to help developers use ASIDE more effectively. For example, (a) Highlight all Boolean logic checks in the current active/visible editor view. The current design requires users to look for and locate such code. Our study shows that users have difficulty locating as well as selecting the right code piece. (b) Attach the information presentation box to the cursor and have the box move along the cursor. The current design has the information box sitting statically at the bottom right corner of the Eclipse window. It can be easily overlooked since it is far away from the code the user is focusing on.

Table 2 Complexity of annotating for Tunestore.

| Use case           | Entry point                                      | Path                                | Sensitive operation                                      | Annotation distance | Required check(s)                   |
|--------------------|--------------------------------------------------|-------------------------------------|----------------------------------------------------------|---------------------|-------------------------------------|
| Change Password    | PasswordAction.java:31:execute(...)              | PasswordAction.java:58:executeUpdate(...) | 2 WHERE clause/CSRF check                                 |
| Add Balance        | AddBalanceAction.java:56:execute(...)            | AddBalanceAction.java:93:executeUpdate(...) | 2 WHERE clause/CSRF check                                 |
| Purchase a Song    | BuyAction.java:32:execute(...)                   | BuyAction.java:73:executeQuery(...)   | 1 WHERE clause                                           |
| Comment a Song     | LeaveCommentAction.java:30:execute(...)          | LeaveCommentAction.java:46:executeUpdate(...) | 2 WHERE clause/CSRF check                                 |
| Add a friend       | AddFriendAction.java:32:execute(...)             | AddFriendAction.java:45:executeQuery(...) | 1 WHERE clause                                           |
| View Friends       | FriendsAction.java:33:execute(...)               | FriendsAction.java:45:executeQuery(...) | 1 WHERE clause                                           |
| Gift Setup         | GiftSetupAction.java:34:execute(...)             | GiftSetupAction.java:60:executeQuery(...) | 1 WHERE clause                                           |
| Gift a Friend      | GiveAction.java:32:execute(...)                  | GiveAction.java:56:executeQuery(...)   | 1 WHERE clause                                           |
| Display Account Balance | LeftAction.java:25:execute(...)                 | LeftAction.java:34:executeQuery(...)   | 1 WHERE clause                                           |
| View Songs for User| ListAction.java:17:execute(...)                  | DBUtil.java:110:executeQuery(...)     | 4 WHERE clause                                           |

Fig. 9 Method execute(...) in ListAction.java in which AD is 4.
Our exploratory study suggests that ASIDE has the potential to be useful for developers who are experienced in application development and are aware of how security vulnerabilities can lead to attacks. However, we need to carefully examine the language ASIDE uses to communicate with developers to make sure it is more broadly understandable. We also need to explore how to provide additional awareness of security vulnerabilities for those who may have little knowledge of them so they can also utilize ASIDE. Table 4 gives a summary of the results from the study.

Related work

**Programmer errors.** Research on human errors [12] and programmer errors [11] suggests that these errors stem from three types of cognitive breakdowns [13,23]: skill-based breakdowns where programmers fail to perform routine actions at critical times; rule-based breakdowns where programmer fail to include required actions in a new context; and knowledge-based breakdowns where a programmer’s knowledge is insufficient. We believe that ASIDE’s real-time, in-situ, and contextualize support in the IDE could help developers avoid many secure programming errors by reminding the programmer about the issues and providing contextual explanations.

**Secure programming education and training.** Software developer education and training has been identified as critical to achieve better software security. Most research work is focused on developing educational materials and secure programming guidelines [2–6]. However, research [13] suggests that developers’ conceptual understanding of security may be disconnected with whether they would apply secure programming in practice. Our work addresses this gap by providing reminders in a developers’ familiar working environment. We are also investigating the use of ASIDE in education and training by introducing ASIDE in university computing curriculum [34].

**Detecting vulnerabilities using static analysis.** Many static analysis techniques have been proposed to detect injection vulnerabilities via data flow analysis [24–27]. Common injection vulnerabilities include SQL injection, Cross-site Scripting (XSS), Command Injection, etc. These vulnerabilities are caused by improper handling of untrusted input. Using data flow analysis to detect injection vulnerabilities is also widely practiced in industry with commercial static analysis tools.

Access control vulnerabilities stem from failure to properly check access credentials before granting access to sensitive resources. Compared with injection vulnerabilities, detection of access control vulnerabilities using static analysis is much less researched. A number of techniques have been proposed to address specific types of access control models such as vulnerabilities associated with role-specific access control [28–31]. Sun et al. [28] aims to identify role-specific access control vulnerabilities in PHP web applications. It requires the specification of a set of pre-specified roles from which it infers implicit access control policy by gathering pages that could be accessed through explicit links. It then looks for access control vulnerabilities by looking for pages that may be accessed via forced browsing. Son et al. [29] proposed a static analysis and code transformation approach to detect access control errors along with providing fixes automatically. It constructs an access control template (ACT) which can be based either on user input or program analysis. ACT is then used to find access violations and propose fixes. RoleCast [30] detects role-specific access control vulnerabilities relying on web application conventions. It infers the collection of user roles by grouping all the program files according to statistical measurements, and then derives the set of critical variables related to access control. It identifies missing access control checks based on inconsistencies in checking critical variables in different program contexts. This approach only covers common database operations (e.g. INSERTS, UPDATE, and DELETE).

These approaches detect access control vulnerabilities via some automatic program analysis techniques. Strong assumptions (e.g. role-based access control or SQL-based database access) are often made for the analysis to be effective, making them difficult to apply to real-world applications which often involve exceptions. In contrast, our approach relies on developers to provide application-specific information to help with static analysis and does not impose any assumption on how applications are built.

Most commercial static analysis tools, such as Fortify [8] and Coverity [10], allow developers to write application-specific security rules for customized analysis. This process requires a lot of security expertise and knowledge of the specific security specification languages provided by the tools.

**Code annotation.** Prior work on code annotation, such as Microsoft SAL annotations [32], relies on textual extensions of programming languages. Similarly, static analysis tools such as Findbugs [33], Fortify [8], Coverity [10], allow the developer to textually annotate their code to improve the accuracy of static analysis. In many cases, requiring developers to learn additional textual languages can potentially discourage developers from using the tool. In contrast, our approach leverages GUI support in the IDE to solicit annotations from the developer, which makes annotation more intuitive.

| Table 3 | Interview responses of participants related to knowledge of attacks. |
|---------|---------------------------------------------------------------------|
| **Participant Interview response** | **Description** |
| P4 | It seems like the example (role-based access control) here, ..., it helped understand what access control is and what you want to check for. But, it might be hard to apply practically [to CSRF]... |
| P7 | I’ve been programming for 20 years. Basically, when you say access control to me, I think of some sort of role-base access control. And then if you say, request forgery, I don’t intuitively put those two together... In order to get me into thinking about that CSRF vulnerability/direction, probably this would have to been expanded a little bit so that I would have known in this context; we are talking about role-based access control and CSRF. |
| P8 | I can imagine it’s difficult to provide contextual explanations, so if your example page had more examples, so if you had like one example of a type of check and an example of this could be Cross-site Request Forgery |

J. Zhu et al.
Conclusions and future work

The main contribution of our work is the framework of interactive static analysis, which integrates static analysis into the Integrated Development Environment as a plug-in and provides in-situ security warnings to help developers detect and resolve vulnerabilities more easily while they are writing their code. This approach could address 7 out of the OWASP Top 10 web application vulnerabilities. We implemented a prototype plug-in of our approach for Eclipse for Java, named ASIDE. We evaluated our approach with ASIDE from multiple aspects, including technical evaluation as well as user evaluation.

Our technical evaluation of ASIDE with a popular large-scale open source project successfully detected multiple zero-day vulnerabilities, which demonstrates that interactive static analysis has the capability to help developers detect vulnerabilities while they are writing code. Our evaluation on the false positives of ASIDE suggests that false positives may be limited to a small class of use cases. Our evaluation of developer’s annotation complexity in performing interactive code annotation suggests that the annotation task is relatively easy to perform with most information easily reachable within the same code block they are working on. And furthermore, our exploratory user evaluation suggests that while developers appreciate ASIDE’s in-situ reminder support and become aware of security vulnerabilities in the program being constructed, it may be most effective for developers with adequate understanding of secure programming.

Our evaluation results suggest that our approach has the potential to help developers write more secure code through in-situ reminders and interactive static analysis. Furthermore, an interactive static analysis tool can positively impact the implementation of a Secure Software Development Life Cycle (SSDLC) by providing additional tool support for SSDLC. For example, it can help enforce secure programming standards by making sure only approved input validation functions are used. Information collected via interactive annotation can be used in code reviews to make sure critical security invariants are implemented correctly.

For the future work, we plan to conduct further evaluations of the effectiveness of interactive static analysis using larger open source projects. We also plan to develop ways to provide more meaningful, contextual explanations to help developers with less experience with secure programming to effectively use interactive static analysis.

Conflict of interest

The authors have declared no conflict of interest.

Acknowledgment

This work is supported in part by Grants from the National Science Foundation 1044745 and 0830624.

References

[1] Bishop M, Orvis BJ. A clinic to teach good programming practices. In: Proceedings from the tenth colloquium on information systems security education; 2006. p. 168–74.
[2] Graff M, Wyk KR. Secure coding: principles and practices. Sebastopol, CA: O’Reilly; 2003.
[3] Howard M, LeBlanc D. Writing secure code. Redmond, WA: Microsoft Press; 2003.
[4] Oracle.com. Secure coding guidelines for the Java programming language, version 4.0; 2012. <http://www.oracle.com/technetwork/java/seccodeguide-139067.html> [retrieved 05.09.12].
[5] Seacord RC. CERT C secure coding standards. Boston, MA: Addison-Wesley Professional; 2008.
[6] OWASP.org. OWASP secure coding practices quick reference guide, version 2.0; 2013. <https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf> [retrieved 15.03.13].
[7] Veracode. State of software security report volume 1, 2, 3 and 4; 2011. <http://www.veracode.com/reports/index.html> [retrieved 20.12.11].
[8] HP Fortify Static Code Analyzer; 2013. <http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812#.UUnHuVthvUY>.
Veracode Static Analyzer. <http://www.veracode.com/products/static>.

Coverity Static Analysis Verification Engine. <http://www.coverity.com/products/coverity-save.html>.

Ko AJ, Myers BA. A framework and methodology for studying the causes of software errors in programming systems. J Vis Lang Comput 2005;16:41–84.

Reason J. Human error. Cambridge, UK: Cambridge University Press; 1990.

Xie J, Lipford HR, Chu B. Why do programmers make security errors? In: Proceedings of 2011 IEEE symposium on visual languages and human centric computing; 2011. p. 161–4.

Knuth DE. The errors of tex. Softw Pract Exper 1989; 19: 607–85.

Xie J, Chu B, Lipford HR. Idea: interactive support for secure software development. In: Proceedings of the third international conference on engineering secure software and systems, ESSoS’11. p. 248–55.

McGraw BCG, Migués S. Building security in maturity model; 2011. <http://www.bsimm2.com/resources/> [retrieved 20.12.11].

OWASP Top Ten Web Vulnerabilities; 2010. <https://owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2010>.

Apache Roller. <http://roller.apache.org>.

CVE-2012-2380. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-2380>.

Apache Roller 5.0.1. <http://rollerweblogger.org/project/entry/roller_501_security_fix>.

Apache Roller 3.0. <http://rollerweblogger.org/project/>.

Roller ROL-1701. <https://issues.apache.org/jira/browse/ROL-1701>.

Xie J, Chu B, Lipford HR, Melton JT. Aside: IDE support for web application security. In: Proceedings of the 27th annual computer security applications conference, ACSAC’11; 2011. p. 267–76.

Martin M, Livshits B, Lam MS. Finding application errors and security flaws using PQL: a program query language. In: Proceedings of the 20th annual ACM SIGPLan conference on object oriented programming systems languages and applications (OOPSLA’05); 2005. p. 365–83.

Livshits VB, Lam MS. Finding security errors in Java programs with static analysis. In: Proceedings of the 14th Usenix security symposium; 2005. p. 271–86.

Jovanovic N, Kruegel C, Kirda E. Pixy: a static analysis tool for detecting web application vulnerabilities. S&P 2006:258–63.

Xie Y, Aiken A. Static detection of security vulnerabilities in scripting languages. In: Proceedings of USENIX security symposium; 2006. p. 179–92.

Sun F, Xu L, Su Z. Static detection of access control vulnerabilities in web applications. In: Proceedings of the 20th USENIX security symposium; 2011.

Son S, McKinley KS, Shmatikov V. Rolecast: finding missing security checks when you do not know what checks are. In: Proceedings of the 26th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications; 2011. p. 1069–84.

Son S, McKinley KS, Shmatikov V. Fix me up: repairing access-control bugs in web. In: Proceedings of 20th annual network and distributed system security symposium (NDSS), San Diego, CA; 2013.

Doupe A, Boe B, Kruegel C, Vigna G. Fear the EAR: discovering and mitigating execution after redirect vulnerabilities. In: Proceeding of the 18th ACM conference on computer and communications security; 2011.

Microsoft. Microsoft SAL annotations; 2011. <http://msdn.microsoft.com/en-us/library/ms235402.aspx> [retrieved 20.12.11].

FindBugs. <http://findbugs.sourceforge.net/>.

Zhu J, Lipford HR, Chu B. Interactive support for secure programming education. In: Proceedings of ACM technical symposium on computer science education (SIGCSE), Denver, Colorado, USA; 2013.