

INTERACTIVE PROGRAMMING SUPPORT FOR SECURE SOFTWARE
DEVELOPMENT

by

Jing Xie

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2012

Approved by:

Dr. Bill (Bei-Tseng) Chu

Dr. Heather Richter Lipford

Dr. Andrew J. Ko

Dr. Xintao Wu

Dr. Mary Maureen Brown

©2012
Jing Xie
ALL RIGHTS RESERVED

ABSTRACT

JING XIE. Interactive programming support for secure software development.
(Under the direction of DR. BILL (BEI-TSENG) CHU)

Software vulnerabilities originating from insecure code are one of the leading causes of security problems people face today. Unfortunately, many software developers have not been adequately trained in writing secure programs that are resistant from attacks violating program confidentiality, integrity, and availability, a style of programming which I refer to as *secure programming*. Worse, even well-trained developers can still make programming errors, including security ones. This may be either because of their lack of understanding of secure programming practices, and/or their lapses of attention on security.

Much work on software security has focused on detecting software vulnerabilities through automated analysis techniques. While they are effective, they are neither sufficient nor optimal. For instance, 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. What is worse, these approaches exclude programmers from the

security loop, and therefore, do not discourage them from continuing to write insecure code.

In this dissertation, I investigate an approach to increase developer awareness and promoting good practices of secure programming by interactively reminding programmers of secure programming practices in situ, helping them to either close the secure programming knowledge gap or overcome attention/memory lapses. More specifically, I designed two techniques to help programmers prevent common secure coding errors: *interactive code refactoring* and *interactive code annotation*. My thesis is that by providing reminder support in a programming environment, e.g. modern IDE, one can effectively reduce common security vulnerabilities in software systems.

I have implemented interactive code refactoring as a proof-of-concept plugin for Eclipse (32) and Java (57). Extensive evaluation results show that this approach can detect and address common web application vulnerabilities and can serve as an effective aid for programmers in writing secure code. My approach can also effectively complement existing software security best practices and significantly increase developer productivity. I have also implemented interactive code annotation, and conducted user studies to investigate its effectiveness and impact on developers' programming behaviors and awareness towards writing secure code.

ACKNOWLEDGMENTS

I owe my deepest gratitude to my advisor, Dr. Bill Chu, a respectful and admirable person who offered me the opportunity of being his student when I was in desperate need of an advisor. It was him who led me on to the pleasant and fulfilling journey of helping developing secure software. He is not only an advisor in research but also a true mentor in life. I cannot remember how many times his wise advice cleared my confusions towards life and helped me be a strong person. I thank him for having faith in me from the beginning, for being patient and encouraging throughout my PhD study. I am proud to have been his student.

I also cannot express how fortunate and grateful I am having Dr. Heather Lipford as my advisor. The moment she showed up, my research life was illuminated and enriched. From her live demonstration of being a passionate, dedicated, and inspiring researcher, I learned to be passionate, dedicated and hardworking towards research, work and life. She, with her unique perspective, can always offer constructive feedback that polishes one's idea. If it were not because of her, I sincerely doubt I would have come this far with my research. I will always cherish this relationship.

My committee members, Dr. Andrew Ko, Dr. Xintao Wu, Dr. Raphael Tsu and Dr. Mary Maureen Brown, thank you so much for investing your valuable time on me. Without your constructive feedback, I would not have had this dissertation.

I am deeply grateful to my parents for their years of encouragement and unconditional love and support of my every adventure. They have always been there telling me that they are proud of me for both my successes and failures. They, despite the

pain inflicted by me coming alone to a foreign land, gave me freedom to chase my dream.

This dissertation would not be made possible without the help from all participants of the studies involved. I will never forget the help from my labmates and friends here at UNC Charlotte who made my everyday life interesting and enjoyable, who shared my joy and pain of being a PhD student. Thank you Leting Wu, Xianlin Hu, Xiaowei Ying, Erin Carroll, Michael Whitney, Andrew Besmer, Berto Gonzalez, Okan Pala, Vikash Singh, and more.

I am thankful to Tony Chen and Tony Kombol, the instructors who gave me permission and helped me in recruiting student participants from their classes. I really appreciate their willingness in making changes to course syllabus to accommodate my study. My thanks are extended to John Melton and Will Stranathan, who have collaborated on and offered valuable insights for my dissertation project.

I would also like to thank HP Fortify for its generous education license which allowed me to use its product Fortify SCA for my study. Finally, I would also like to thank the National Science Foundation for its financial support. Without that, I would not have been pursuing a PhD in the first place.

TABLE OF CONTENTS

LIST OF FIGURES	ix
LIST OF TABLES	xii
CHAPTER 1: INTRODUCTION	1
1.1 Scope of Research	4
1.2 Dissertation Overview	5
CHAPTER 2: RELATED WORK	7
2.1 Attack Detection and Prevention	7
2.2 Vulnerability Detection and Discovery	11
2.3 Secure Programming	14
2.4 Programming Errors	15
CHAPTER 3: WHY DO PROGRAMMERS MAKE SECURITY ERRORS	17
3.1 Study Methodology	18
3.2 Study Results	19
3.3 Study Discussion	26
CHAPTER 4: INTERACTIVE SECURE PROGRAMMING SUPPORT	29
4.1 Interactive Support for Software Programming	29
4.2 Interactive Secure Programming Support	30
CHAPTER 5: INTERACTIVE CODE REFACTORING	35
5.1 Target Vulnerabilities Profile	35
5.2 Interactive Code Refactoring	44
5.3 Open Source Projects Evaluation	49

5.4	Developer Study	56
CHAPTER 6: INTERACTIVE CODE ANNOTATION		80
6.1	Target Vulnerabilities Profile	81
6.2	Interactive Code Annotation	82
6.3	Walkthrough Evaluation	84
6.4	CodeAnnotate	89
6.5	Performance Measurement	97
6.6	User Study	113
CHAPTER 7: CONCLUSION AND FUTURE WORK		130
7.1	Restatement of Contributions	130
7.2	Future Work	132
7.3	Closing Remarks	135
REFERENCES		136
APPENDIX A: RULES		144
APPENDIX B: STUDY MATERIALS		149

LIST OF FIGURES

FIGURE 1: Software security best practices	5
FIGURE 2: Code Refactoring warnings	45
FIGURE 3: Options for input validation	46
FIGURE 4: ASIDE validates an input using OWASP ESAPI validator API.	46
FIGURE 5: CodeRefactoring warnings for Map	47
FIGURE 6: Untrusted input is logged through an Exception construction.	55
FIGURE 7: Untrusted input is used for logic test.	55
FIGURE 8: Untrusted input is parsed into harmless Boolean value.	55
FIGURE 9: CodeGen options	58
FIGURE 10: CodeGen generates code	58
FIGURE 11: Explanation warning details	59
FIGURE 12: Metrics from students with CodeGen.	63
FIGURE 13: Metrics from students with Explanation.	65
FIGURE 14: Metrics for Professional Developers with CodeGen.	71
FIGURE 15: Metrics for Professional Developers with Explanation.	73
FIGURE 16: Broken access control problem	82
FIGURE 17: Broken access control solution	83
FIGURE 18: ASIDE interactive code annotation example	85
FIGURE 19: Control flow diagram	87
FIGURE 20: Annotate access control logics.	88
FIGURE 21: Authentication in Java servlet	88

FIGURE 22: Code for changing profile	90
FIGURE 23: Launch CodeAnnotate	91
FIGURE 24: CodeAnnotate UI	92
FIGURE 25: CodeAnnotate offers 3 options from which a developer can select.	93
FIGURE 26: CodeAnnotate explains warning detail	94
FIGURE 27: Click me to annotate a control logic	95
FIGURE 28: CodeAnnotate UI Elements	96
FIGURE 29: Undo an annotation	97
FIGURE 30: <i>Tunestore</i> Login Action Servlet	100
FIGURE 31: 10 false positive cases of <i>Tunestore</i>	102
FIGURE 32: Complexity of annotating for <i>Tunestore</i>	104
FIGURE 33: iBatis implementation for <code>loginUser()</code> method	108
FIGURE 34: 3 false positives from <i>Goldrush</i>	110
FIGURE 35: Complexity of annotating for <i>Goldrush</i>	112
FIGURE 36: CodeAnnotate warning	118
FIGURE 37: CodeRefactoring warning for SQL Injection	118
FIGURE 38: CodeAnnotate warning for single statement	119
FIGURE 39: Summary of Think-aloud study	127
FIGURE 40: Trust boundary rule	144
FIGURE 41: Input validation rule	146
FIGURE 42: Sensitive accessor rule	147
FIGURE 43: Consent Form	149

FIGURE 44: Consent Form	150
FIGURE 45: Interview Questions	151
FIGURE 46: Interview Questions	152
FIGURE 47: Consent Form	153
FIGURE 48: Consent Form.	154
FIGURE 49: Description of Programming Task	155
FIGURE 50: Description of Programming Task	156
FIGURE 51: Interview Questions	157
FIGURE 52: Consent Form	158
FIGURE 53: Consent Form.	159
FIGURE 54: Study Instructions	160
FIGURE 55: Study Instructions	161
FIGURE 56: Study Instructions	162
FIGURE 57: Development Environment Setup	163
FIGURE 58: Description of Programming Task	164
FIGURE 59: Description of Programming Task	165
FIGURE 60: IDescription of Programming Task	166
FIGURE 61: Interview Questions	167

LIST OF TABLES

TABLE 1: Fortify SCA results	51
TABLE 2: SCA result details	52
TABLE 3: Access control tables	83
TABLE 4: Security issues in open source projects	86
TABLE 5: The 8 paths in <i>Goldrush</i> found by <i>CodeAnnotate</i>	109
TABLE 6: Trust boundary rule mapping.	145
TABLE 7: Trust boundary rule mapping.	146
TABLE 8: Sensitive accessor rule mapping.	148

CHAPTER 1: INTRODUCTION

Software is essential for computing and has become ubiquitous and pervasive throughout society. Insecure software that is vulnerable to malicious attacks, therefore, poses tremendous risks to the security of people's daily lives. A recent attack (54) on a large financial company's website exploited an application vulnerability that not only caused the company a loss of \$2.7 million and its reputation but also brought its customers huge inconveniences and potential troubles despite the free liability of their financial loss. For example, customers needed to change payment information that was associated with the compromised account.

The security quality of software is therefore no longer a privileged property of critical systems such as aircraft control systems, but instead a concern of developers of all types of software, including even a simple personal website (2). Software security is about building software that is secure in a manner that it is resistant to malicious attacks. Software may be insecure for various reasons. Studies show that, however, software flaws are one of the root causes of the majority of exploitations and breaches (78). Software flaws are essentially program bugs. When such flaws can be exploited in a way that violates the confidentiality, integrity and availability of the information on which the software relies, they are referred to as software vulnerabilities.

There are multiple points where software vulnerabilities can be introduced through-

out the software's development life cycle. For instance, the design of a password retrieval system can employ a weak challenge question that leads to a logic vulnerability; an implementation of credential verification can process user inputs without proper validation which leads to a SQL Injection; a configuration of the deploy environment may lead to dysfunction of the system. Vulnerabilities that are due to insecure code written by programmers, however, are most commonly exploited by attackers (78).

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.

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, etc. Thus, education and training software developers about secure programming is not a satisfactory solution to preventing security errors.

In this dissertation, I discuss a different approach that provides software developers interactive support for secure programming practices in order to produce more

secure software. The approach reminds the developers of better 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 (65; 39). My hypothesis is that by providing effective reminder support in an IDE, one can effectively reduce common security vulnerabilities. This approach is analogous to word processors' 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 cues - colored lines drawn underneath potential errors - to help writers notice and fix problems while they are composing text. Similarly, this 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.

This dissertation work has several technical, theoretical and Human Computer Interaction contributions:

- Provides an in-depth understanding of why software programmers make security errors during programming with support of empirical evidence.
- Devises a new approach which reminds software programmers of potential insecure code and provides them secure programming support during program construction, to help them write secure code, in order to eventually develop more secure software.
- Develops two techniques, *interactive code refactoring* and *interactive code an-*

notation, to assist programmers in producing code with fewer common code vulnerabilities.

- Implements prototype software in the form of plugins for the Eclipse platform and conducts an extensive study of open source projects to evaluate the effectiveness of the proposed techniques in addressing common vulnerable code.
- Conducts user studies evaluating the current design of the implemented prototype to gain insights on how developers perceive and react to this interactive approach.

1.1 Scope of Research

Security in Information Technology (IT) is a loaded concept. It means different things in different contexts, to different people. It is plain and easy for the general public as well as computing professionals to relate to data encryption, network firewall, or antivirus, when talking about information security. However, my work in this dissertation, although falls into the big umbrella of information security, it should be categorized more precisely into software security. The term software security was first used in 2001 by Viega and McGraw (79) as the idea of engineering software so that it continues to function correctly under malicious attacks (47).

Security is an emergent property of a software system. A security problem is more likely to arise associated with a normal functional part of the system (say, the interface to the database module) than in some given security feature. This is because software security has to do with the way software is implemented in general, including the way security features are implemented. This is an important reason why software security

must be part of a full lifecycle approach. Figure 1 specifies one set of best practices that need to be applied to the various software artifacts produced during software development.

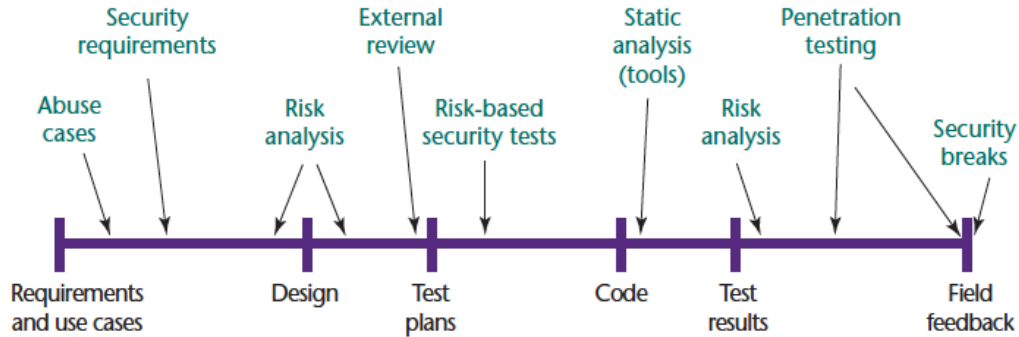


Figure 1: Software security best practices applied to various software artifacts (47).

While it is apparent from Figure 1 that a piece of software can be made vulnerable at various stages of its lifecycle, my research only focuses on the coding phase. It is at this stage where a number of *code vulnerabilities* are introduced by software programmers writing insecure code.

1.2 Dissertation Overview

In this dissertation, I first present an overview of all the related work in Chapter 2. I then report my research study of investigating the underlying causes of programmers making security errors in Chapter 3. In Chapter 4, I present an overview of my proposed approach, Application Security in IDE (ASIDE), to improving the security quality of software by providing interactive secure programming support for developers. More specifically, I introduce two techniques, *interactive code refactoring* and *interactive code annotation*, to address two different types of common secure programming issues. In Chapter 5, I detail my research efforts on implementing

and evaluating *interactive code refactoring*. I dedicate Chapter 6 to design, implementation and evaluation of *interactive code annotation*. I conclude my dissertation contributions and outline future work in Chapter 7.

CHAPTER 2: RELATED WORK

Confined within the declared scope of research, the work that is relevant primarily falls into 4 strands: *attack* detection and prevention, *vulnerability* detection and discovery, *secure programming*, and programming *errors*. Attack detection and prevention aims at preventing attacks launched by malicious users. Vulnerability detection and discovery is about finding security holes in software. Secure programming and programming errors shift the attention to software developers.

2.1 Attack Detection and Prevention

Software, for historical reasons, is developed without following a sound security methodology. Most software is vulnerable to attacks when deployed. This is because security vulnerabilities are either overlooked or not properly addressed during development. Being heavily influenced by the way people address security issues in networking for the past two decades, researchers in software security identify specific *attacks* as their prime target for prevention. They direct their effort to protect systems from being attacked by detecting attacks at the time they are taking place, and preventing them when being detected to thwart potential damage to the system.

2.1.1 Learning-based Prevention

To detect attacks and further prevent them from being carried out successfully, one needs to have a clear definition of what an attack is, or what makes an attack an

illegitimate interaction with the system. Thus, most techniques that are developed function in two phases. The first phase is to build a model of legitimate interactions. The second phase is to capture interactions and then check them for compliance with the established model. If the interaction fits into the model, it is considered as legitimate. Otherwise, it is viewed as an attack, and certain actions would be taken to mitigate the attack.

To put this approach in perspective, Halfond and Orso (24) implemented a tool, AMNESIA, that detects and neutralizes SQL Injection attacks that are commonly seen as one of the most serious security threats to web applications. Cova et al. (18) investigated an approach to first analyzing the internal state of a web application, and learning the relationships between the application's critical execution points and the application's internal state, then identifying attacks that attempt to bring an application into an inconsistent, anomalous state that violates the intended workflow of the application. Earlier efforts include developing an anomaly-based system that learns the profiles of the normal database access performed by web-based applications using a number of different models (77).

2.1.2 Proxy-based Prevention

Researchers also have resorted to proxies which intercept and examine incoming requests from client side to differentiate malicious requests from legitimate ones before they are consumed by application backends. Boyd and Keromytis (9) applied the concept of instruction-set randomization to SQL, creating instances of the language that are unpredictable to an attacker, in order to catch potential SQL injection

attacks. They implemented the idea by using an intermediary proxy to translate randomized SQL to MySQL's standard language. Liu et al. (42) proposed an SQL Proxy-based Blocker, which harnesses the effectiveness and adaptivity of genetic algorithms to dynamically detect and extract users' inputs for undesirable SQL control sequences. Bisht et al. (8) designed and implemented CANDID, which dynamically mines programmers' intended query structure on any user input. It requires extra instrumentation to transform the web application code, which is usually tied to a specific programming language.

2.1.3 Dynamic Prevention

Another type of technique, which targets injection-based attacks, is to track precise taint information about user input. It taints the input strings and tracks those taints along the information flow of a program. Huang et al. (31) developed WebSSARI, a tool which uses a static analysis technique based on type-based information flow to identify possible vulnerabilities in PHP web applications. This type-based approach, however, operates at a coarse-grain level: any data derived from tainted input is considered fully tainted. WebSSARI can insert calls to sanitization routines that filter potentially dangerous content from tainted values before they are passed to security-critical functions. Nguyen-Tuong et al. (53) proposed techniques for tracking taintedness at a much finer granularity. Their system can be more automated than WebSSARI: all they require is that the server uses their modified a PHP interpreter to protect all web applications running on the server. Su and Wassermann (74) presented a formal definition of SQL Injection attacks and proposed a sound and complete

(under certain assumptions) algorithm, which can identify all SQL Injection attacks, by using an augmented grammar and by distinguishing untrusted inputs from the rest of the strings by means of a marking mechanism. Halfond et al. (25) researched along the same line of tracking taint information at the character level and using a syntax-aware evaluation to examine tainted input. They differ their approach by employing a novel concept of positive tainting, which the researchers argue to be a safer way of identifying trusted data. Positive tainting differs from traditional tainting in that it is based on the identification, marking, and tracking of trusted, rather than untrusted, data.

2.1.4 Vulnerability-specific Attack Prevention

Some attacks exploit vulnerabilities that have less generic characteristics, such as the ones described above. Quite often, the defense strategies have to do with more than just the application itself, but also involves the infrastructure, such as the web browser, on which the application relies. For instance, Barth et al. (4) introduced loginCSRF, a new variant on Cross-site Request Forgery (CSRF) attack, which renders existing CSRF defense techniques ineffective. Therefore, they proposed that browsers implement the `Origin` header to provide the security benefits of the `Referer` header while responding to privacy concerns. Maes (44) presented a client-side policy enforcement framework to transparently protect the end-user against CSRF, and implemented it in the form of a Firefox extension. The framework monitors all outgoing web requests within the browser and enforces a configurable cross-domain policy. The default policy is carefully selected to transparently operate in a

web 2.0 context.

2.2 Vulnerability Detection and Discovery

There is a rich volume of work of finding vulnerabilities in software that is fully developed and deployed in some cases. The major approaches taken to discovering vulnerabilities are in three forms. The first one is to perform static analysis on source code. The second one is to carry out dynamic analysis on running applications. The third approach simulates attack scenarios by injecting attacks into applications.

2.2.1 Static Source Code Analysis

Chess and West (16) provided a comprehensive overview of static analysis approaches for security problems. With the shifting of the computing paradigm from desktop to web, vulnerabilities in web applications naturally become the target of research. Livshits and Lam (43) proposed a static analysis technique which is based on a scalable and precise points-to analysis to find vulnerabilities that can be abstracted as general *tainted object propagation* problem, which include SQL Injection, Cross-site Scripting (XSS), HTTP Splitting, etc. Wassermann and Su (81) advanced this approach by characterizing the values a string variable may assume with a context free grammar, tracking the nonterminals that represent user-modifiable data, and modeling string operations precisely as language transducers to eventually check the conformance to the policy that an attack happens when user input changes the intended syntactic structure of a generated query. Other efforts that have impact include Pixy (34; 35) which is a static taint analysis for PHP. It propagates limited string information and implements a finely tuned alias analysis. Xie and Aiken (86)

designed a more precise and scalable analysis for finding SQL Injection in PHP by using block- and function-summaries. Tripp et al. (76) utilized well-studied static taint analysis techniques and built a static analysis tool that can scale to large industrial web applications, model essential web application code artifacts, and generate consumable reports for a wide range of attack vectors. More information can be found in a survey (64) of static analysis methods for identifying security vulnerabilities in software systems.

2.2.2 Dynamic Runtime Analysis

Another active thread of research that finds vulnerabilities in software focuses on performing dynamic analysis on running applications. Most commonly, such dynamic analysis is based on two well studied concepts: symbolic execution and model checking. For example, Chaudhuri and Foster (14) developed a symbolic executor, Rubyx, to look for vulnerabilities in web applications that are built on Rails. Depending on the construction of the Rubyx specification, which is built from general assertions, assumptions, and object invariants, Rubyx can be adapted to detect a wide range of vulnerabilities including Cross-site Scripting, Cross-Site Request Forgery, an insufficient authentication, insufficient access control. Felmetsger et al. (20) demonstrated the possibility of using a multi-step approach, which involves extensive dynamic analysis, to detect logic vulnerabilities in web applications. They first used dynamic analysis and observed the normal operation of a web application to infer a simple set of behavioral specifications. Then, leveraging the knowledge of the typical execution paradigm of web applications, they filtered the learned specifications to reduce false

positives, and then used model checking over symbolic input to identify program paths that are likely to violate these specifications under specific conditions, indicating the presence of a certain type of web application logic flaw. Huang et al. (30) formalized web application vulnerabilities as a secure information flow problem with fixed diameters and then used bounded model checking to achieve counterexample generation and complete verification. Fu and Qian (21) developed a tool set called "SAFEI" to detect SQL Injection vulnerabilities in web applications by instrumenting the bytecode of Java web applications in conjunction with symbolic execution. At each location that executes a SQL query, an equation is constructed to find out the initial values of web controls that lead to the breach of database security. The equation is solved by a hybrid string solver where the solution obtained is used to construct test cases. Schwartz et al. (70) conducted a thorough review of the utilization of dynamic analysis in detecting software security vulnerabilities.

2.2.3 Attack Injection

There is also research that takes an approach that is inline with how attackers find vulnerabilities in software systems, which is to attack deployed systems and see whether the systems can break. Kieyzun et al. (38) presented a technique that generates sample inputs, symbolically tracks taints through execution, and mutates the inputs to produce concrete exploits. These automatically generated inputs were demonstrated to be effective in exposing SQL Injection and Cross-site Scripting attacks that are common in modern web applications. Martin and Lam (45) investigated using goal-directed model checking to automatically generate attacks exploiting taint-

based vulnerabilities in large web applications. The approach was implemented as a system called QED, which accepts any Java web application that is written to the standard servlet specification. It requires an analyst to specify the vulnerability of interest in a specification that looks like a Java code fragment, along with a range of values for form parameters. QED then generates a goal-directed analysis from the specification to perform session-aware tests, optimizes to eliminate inputs that are not of interest, and feeds the remainder to a model checker. The checker will systematically explore the remaining state space and report example attacks if the vulnerability specification is matched.

2.2.4 Summary

While these approaches are making it more effective in finding or preventing exploitations of vulnerabilities in software, they overlook an essential element that plays a pivotal role in vulnerability introduction, which is software developers. Vulnerabilities are normally introduced into software by software developers writing insecure code unintentionally. The approach to finding vulnerabilities after the program is written excludes developers from the security loop, and thus exerts no influence over preventing them from continuing to produce insecure software. Moreover, finding vulnerabilities is not the end, given that efforts must be taken to fix what has been found as vulnerable.

2.3 Secure Programming

The term “secure programming” is widely used but loosely defined in the security community. In most cases, it implies a programming style that bears security implica-

tions of code and implements defensive code that resists malicious exploits. Another term “secure coding” also carries a similar meaning, and is used interchangeably with secure programming. The vast majority of effort to increase software programmers’ awareness of programming in a secure fashion has been placed on training and guideline generation. For instance, there is the Top 10 Secure Coding Practices (13) from CERT (12). OWASP (60) has an open source project (61) that is dedicated to providing a quick reference guide for secure programming best practices. More formal presentations include Viega and Messier’s secure programming cookbook (80).

An exception that deviates from the conventional training is Bishop’s Secure Programming Clinic approach (7). Continuous reinforcement by using clinics is a common and effective technique used to improve students’ writing ability in law schools and English departments. Secure programming clinics thus reinforce good programming style in a similar fashion as how writing clinic reinforce good writing style.

2.4 Programming Errors

Research into software errors has a long history starting in the early 1980s. Ko and Myers (39) provided a comprehensive summary of related work. Moreover, they explored the underlying cognitive mechanisms of general programming errors due to the interaction between a programmer and a programming system based on James Reason’s *Human Error* (65), and identified three types of cognitive breakdowns that lead to programming errors. *Skill-based breakdowns* are where programmers fail to perform routine actions at critical times; *rule-based breakdowns* are where programmers fail to do an action in a new context; and *knowledge-based breakdowns* are where

a programmer's knowledge is insufficient. In addition, they formed the chains of cognitive breakdowns over the course of programming activity to explain the introduction of programming errors.

CHAPTER 3: WHY DO PROGRAMMERS MAKE SECURITY ERRORS

A great deal of effort motivated by Microsoft's Secure Development Lifecycle (SDL) initiative (27), has been placed on reducing the vulnerabilities in software over the past decade. The overall security quality of software, however, is still far away from where we need it to be. Most effort was on procedural, technical improvements which overlooked a fundamental question, which is why do programmers make security errors? In most cases, programmers do not write insecure code intentionally but do so anyway.

The earliest attempt to explain this phenomenon appeared in Wheeler's Secure Programming for Linux and Unix HOWTO (83), where he presented a list of purported reasons that were collected and summarized by Aleph One on Bugtraq. One of the major factors is that programmers were not educated about real-world vulnerabilities, let alone how to write code that does not introduce vulnerabilities. In addition, security is considered as an inhibitor of easy programming because it costs extra development time and additional testing. More formal effort (22) postulates three reasons for "Why Good People Write Bad Code": Technical factors referring to the underlying complexity of the task itself; Psychological factors including poor mental models or difficulty with risk assessment and real-world factors comprising lack of financial incentives and production pressures. These opinions are informative

but lack empirical evidence.

More recently, Woon and Kankanhalli (84) conducted a survey investigating the intention of information systems professionals to practice secure development and revealed that the most important factors for the lack of intention towards secure development are attitudes regarding career usefulness as well as the influence of social norms. The target population, however, is too generic, thus does not reflect the characteristics that are unique to programmers. Therefore, I expanded upon these results by looking more in depth at developers who currently write code, and their particular attitudes and practices.

3.1 Study Methodology

I conducted a study that involved semi-structured face-to-face and phone interviews with 15 professional software developers with various backgrounds over a 3-month period. Twelve of the participants are former students at UNC Charlotte, from throughout the past 13 years. The remaining 3 were other personal contacts of Dr. Bill Chu's or were referred to me by personal contacts. Programming is a major professional activity of all interviewees.

All interviewees worked for organizations including apparel manufacturing, banking and financial corporations, independent software service providers and corporation technology providers. They also developed a variety of software ranging from front-end web applications, internal middleware to back-end database development. All of them used mainstream programming languages such as Java, C++, C, Python, PHP, and JavaScript. They averaged 12.6 years of experience, with a median of 11 years,

as professional software developers.

The length of each interview ranged from 30 minutes to one hour. During the conversations, I first asked about their professional software development backgrounds. Then I focused on the interviewees' opinions of the relationship between software security and the software development life cycle. I also discussed with them the most important security concerns for their software, and next explored the procedures, mechanisms and tools employed by their companies for secure software development. Throughout, I asked participants about their own personal perceptions and practices regarding software security, and what prompts them to think about or act upon security issues. All the interviews were recorded and transcribed. I used a qualitative data management and analysis tool, Atlas.ti, to iteratively perform open coding and identify general themes and patterns.

3.2 Study Results

For this study, I did not seek participants with specific security knowledge. I found, however, that most of my participants did have a reasonable awareness and knowledge of software security issues. Seven of the participants expressed high software security awareness. They were able to name their major security concerns and identify code patterns which had security implications. Moreover, they were capable of elucidating the causes of some security issues and the correct mitigations for them. Five of the interviewees had moderate awareness. They maintained good knowledge of security features such as using a password to authenticate, but overlooked the cases that involve malicious users, e.g. Cross-site Scripting (XSS) attacks. One participant had

a low level of awareness: he was only able to discuss software security on an abstract level. Only two participants had very little idea of the concept of software security.

While I obviously have a small sample of developers, these results suggest that the message on the importance of software security and knowledge of security vulnerabilities is getting to developers. The majority agreed that software security is important in all phases of the software development lifecycle. Despite this general knowledge, however, interviewees were not able to concretely describe their own personal practices with regard to software security, even those with high security awareness. I identified several themes regarding their perceptions, which explain their lack of individual responsibility and practices towards software security.

3.2.1 Misplaced Trust in Process

Professional software development is a collaborative process involving multiple parties and individuals. Each party takes responsibility for a certain component of the whole. For instance, designers conceive of and develop the architecture of a project, while software developers actually write the code to implement a design. Testers ensure that what developers did is consistent with what designers specified. Ideally, software security should be considered by all parties at every development stage, but I am particularly interested in software security implementation performed by software developers.

Most of the participants had reasonable knowledge about software security. They could identify some of the common security issues and were aware of its importance. However, when asked about their personal practices of addressing software security

issues, they constantly referred me to other processes or parties that handled these security concerns.

For example, one of the interviewees alleged that in his company, software security was fully incorporated in the design phase and software designers were the ones who should be responsible for taking all possibilities into consideration and formulating the corresponding design specifications. Implementation is nothing more than just writing code to carry out the established design. Another participant, when asked a question about how he specifically dealt with XSS vulnerabilities, stated:

[P8] When I do my part of work, I don't have this consideration at implementation phase. To me, that's the case we should consider at design phase.

In three cases, the participants brought up their code review process where the code that is written by them and their co-developers is gathered and then reviewed.

[P5] We take care of that [software security] both in code reviews which happen while the project is in progress, being coded as well as at the end before it gets merged into our master trunk.

Depending on how the company conducts that process, the reviewers can be either the developers themselves for a peer review, or external auditors. As the interviewees reported, security tools were generally involved in that process. Most of them were commercial and open source static code analysis tools such as Coverity (19), Fortify SCA (72), Findbugs (6), and PMD (73).

In another two cases, the participants declared that software testing is capable of discovering all software bugs, and security bugs are no exception.

[P7] But I think most times we catch the security (issues) in testing.

Several participants raised the role of a specialized security group whose main focus is software security. One participant recounted that such a group intervened in the design phase. Another interviewee described that security experts in this group acted as security supervisors of the whole development process.

[P8] In all groups, we have a dedicated team working on the security part.

Developers exhibited a relaxed attitude towards software security when knowing that there were experts to back them up. In all cases, participants expressed satisfaction and trust with the other people and processes that were supposed to be handling the security issues.

3.2.2 Security in Context

In addition to describing how their organization handles software security, many participants also explained that the security issues they are aware of do not apply to their particular development context. For example, three of my interviewees who do not work on web applications acknowledged the importance of common web-based vulnerabilities and exhibited awareness and knowledge of those issues. However, they stated that those vulnerabilities do not concern them since they do not work on web applications. One participant, who has been a middleware developer in a large enterprise technology provider for 7 years, expressed that he does not worry about security since he does not build front-end applications:

[P13] They [the group that works on portal application] might have bigger concerns than us because we are basically sitting in the backend.

Although some software, such as middleware, provides fewer attack vectors to attackers as compared to web applications and web related software (e.g. web browsers), there are still potential vulnerabilities due to similar programming errors, such as not performing proper input validation.

Another reason noted by two of the interviewees was that the software they were developing would only be used by a small and known user population. In one case, the participant was working on internal software that would only be used by corporation employees. The other interviewee asserted that the potential users were non-technical managers. He believed that these users were either technically incapable of doing anything malicious to the software or would not take the risk of losing their job by attacking the software. The problem with this perception is that attacks can also occur by an attacker hijacking an innocent user's account, and then using that account to behave maliciously.

One interviewee also argued that most of their products were based on third party commercial software. Therefore, he believed that their software security depends on commercial software security. As long as the commercial software was up-to-date, he did not think his company's software would be vulnerable to attacks.

[P15] [Q: So that means you trust other people's software?] I have to because, right now it's pretty good for, like, if you have a security hole, you can just upgrade it. You know, to get patches.

Similarly, participants seemed to trust reused code, even those with high security awareness. All my interviewees acknowledged that reusing code is a common programming practice. However, code security was never a criterion they used to help them decide on which reused code to choose. In a study on the use of online resources during programming, Brandt et. al (11) found a similar trust in the correctness of reused code, which made it challenging to find bugs in that adapted code.

The fundamental characteristic of software security is that each and every security hole contributes to the compromise of a system and breach of confidential information. A piece of software that has only one vulnerability is still not secure. Additionally, a piece of vulnerable software may not be the direct target of an attacker, but may still be used in an attack to get closer to a target. Therefore, software security should be considered within all domains and contexts. This is even more important considering the trust placed in the security of other software and reused code.

3.2.3 External Constraints

Professional software development is influenced by a variety of external factors. Business deadlines, planned budget, customer demands, and developer knowledge all impact the priorities for the limited resources of a project. Security is just one consideration among many. The interviewees identified a range of factors that motivate and constrain the attention paid to various security concerns throughout the process, influencing the attention paid by individual developers during their own activities.

A key motivator my participants mentioned for security is the concerns of the customer or client. As one interviewee pointed out:

[P8] If (the) customer cares about security then the company has to care about security.

Government regulations and organization policy also provide rules that developers must adhere to regarding security. However, if there are no regulations or policies that are required, then developers will not be encouraged to perform any extra security work. Developers may even be discouraged from doing additional security development because they do not want to do go beyond the rules and do something wrong.

[P9] I will probably just follow the rules, follow the tradition, and do what the other people did.

Not surprisingly, the business logic of an application is the primary concern. Developers perceive that security may interfere with that logic or make the software more complicated. Participants also reported that security is seen as an expense and potentially time consuming. So as the time or budget is limited, software security is one of the concerns that get overlooked, either explicitly or implicitly.

[P1] Now the only way you can fit a 3-month project into 3 weeks is to cut a whole lot of corners. Security was one of those corners that got cut.

Software security is also seen as a complex technical topic, requiring specific knowledge of developers.

[P13] These days the software is so complex. You would need to have very specific knowledge and expertise to work on that [security].

These concerns are certainly valid. Ensuring secure software does require additional time and resources throughout the development process. However, I also believe that basic secure programming practices can be a part of any development without much added burden, preventing many common and serious vulnerabilities as a result.

3.3 Study Discussion

Throughout the study, it is clear that my interviewees showed a strong reliance on other people, processes, and technology to take care of software security. Given the many concerns of developers who write software code, it is not surprising that they would lighten their load by passing responsibility of software security onto others when possible. Fellow developers, teams, and organizational policies may even encourage such perceptions. The danger, however, is that vulnerabilities are introduced and overlooked due to this lack of concern and misplaced trust in others, who may also not be particularly concerned with their software security. Yet, many security vulnerabilities, including some of the most common and serious, can be prevented with relatively simple code practices. While design, code review, and testing certainly play an important and integral role in overall secure software development, the software developers who make security errors are the best people to prevent security errors in the first place. My interviews reveal that participants did not seem to share this perception, which may be leading to easily preventable security errors and greater costs in detecting and fixing vulnerabilities.

Despite numerous methods and processes (67) that have been researched by the information security and software engineering communities to encourage secure soft-

ware development, the severity of the problem has not been significantly alleviated. For instance, most efforts that aim to educate and train developers to develop secure software have been spent on developing educational material and guidelines for the best secure programming practices (12; 33; 1; 75; 36). However, the mere existence of such abundant information does not guarantee its use by programmers (85). On the other side of the spectrum, 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 (52; 34; 43; 15) and dynamic analyses are often based on model checking (30; 20; 45) and symbolic execution (14; 21; 69). 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; 46; 31).

Existing tool support, however, suffers from common drawbacks which significantly reduce tools' effectiveness and impede their possibility of long-term usefulness. More specifically, it comes to play late at the end of the software development cycle, and thus may not be guaranteed to be used when other software development priorities take precedence. Worse, these tools, regardless of the implementation techniques, are normally used by security experts and thus excludes programmers from the security loop. I, therefore, believe interactive secure programming support for developers can play a role to fill this void.

The results of this study suggest a number of design goals to help bridge this

gap between general security knowledge and concrete secure programming practices. First, developers need greater awareness of specific errors in the context of their own development. Tools that detect and flag such code during program construction, not after code completion, may help alert them to places requiring additional attention. Tools should also be customizable, as different types of software or domains, such as web applications versus middleware, may have different security concerns. Developers are also overburdened with many concerns competing for their time and attention. Tools should be lightweight, even automatically producing or suggesting secure code when possible. Ideally, tools would be integrated into existing development environments, such as the IDE, to reduce training needs and increase accessibility. Tools also need to accommodate the existing development processes which already have responsibility for secure development, such as letting designers customize programming rules for their environments or providing information to aid testing and code review.

CHAPTER 4: INTERACTIVE SECURE PROGRAMMING SUPPORT

4.1 Interactive Support for Software Programming

Software programming is a highly complex human activity, which involves cognitive processes such as reading, writing, learning, reasoning, and problem solving (82). Interactive tool support for various programming tasks has helped to successfully improve programming productivity and to ease the cognitive burden on programmers (10; 37). One example is the syntax-directed editor, which colors program tokens according to their syntactic meaning. This has become an indispensable component of any modern Integrated Development Environment (IDE).

Interactive tool support for software programmers in performing various programming activities has been extensively studied. One line of research is to improve programmer productivity by providing interactive code editing support. A prominent success is the incremental compilation of code in Eclipse that offers immediate feedback on errors as well as provides quick fixes for common problems while programmers are editing source files (40). My approach is modeled after this mechanism.

Facilitating debugging through interactive support has gained attention recently. Hao et al. (26) proposed VIDA which continuously recommends break-points for programmers based on the analysis of execution information and the gathered feedback from a programmer during his/her conventional debugging process, increasing

programmer efficiency. Murphy-Hill and Black (51) investigated an interactive code analyzer that offers refactoring advice when requested. Similarly, I hope to offer secure programming advice to programmers without interrupting their primary tasks and increase their efficiency in performing secure programming.

Researchers in Human Computer Interaction and software engineering have sought to design and evaluate more usable interactive tools to help professional developers reduce general program errors. For example, Ko and Myers have developed a model for program errors (39), which later led to the design and implementation of a new approach for interactive debugging (41). Since security errors are a subset of general programming errors, I believe they can be greatly dealt with via similar interactive support in programming environments.

4.2 Interactive Secure Programming Support

My approach is based on the following 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 that creating a tool that integrates into the programmers' development environment is promising.

My second consideration is the interface design principle that recognition is favored over recall (71). Developers are provided with appropriate visual alerts on secure programming issues and offered assistance to practice 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.

Fourth, I want to support sharing secure programming knowledge and standards amongst development teams. In an industrial setting, the tool should be configured by an organization's software security group (SSG), which is responsible for ensuring software security as identified by best industry practice (48). Thus, a SSG could use the tool to communicate and promote organizational and/or application-specific programming standards. In addition, the tool can generate logs of how security considerations were addressed during construction, providing necessary information for more effective code review and auditing.

Finally, integrating such support into the IDE promotes the accessibility of secure programming knowledge.

I carried out the above design idea of integrating secure programming support into the programming environment through the implementation of ASIDE (*Application Security in IDE*) (87), which is an Eclipse plugin for Java. But before I delve into the details of the working mechanics of ASIDE, I describe a preliminary evaluation to understand why a tool like ASIDE might be effective at preventing software security bugs by investigating how such errors may be committed. Ko and Myers have done a comprehensive survey of previous research on causes of programmer errors (39). According to them, programmer errors can be traced to three types of cognitive breakdowns: *skill-based breakdown*, *rule-based breakdown*, and *knowledge-based breakdown*, which take place during skill-based activities, rule-based activities, and knowledge-based activities, correspondingly.

Skill-based activities are routine, mechanical activities carried out by developers, such as editing program text (39). To put common patterns of skill-based breakdown in a security context, consider when a programmer copies a block of code to reuse in his program. He realizes that input validation needs to be modified to suit the new application context. This could be a routine task by changing input validation instances to a different API call. However, this task (changing API calls in the edit window) was interrupted, for instance, by an instant message from a colleague. When his attention is brought back to the task, an instance of the old input validation was missed, thus causing the software to be vulnerable. Attention shifts are the principle cause of skill-based breakdowns. ASIDE could be effective to mitigate such errors by alerting programmers to important security issues, refocusing the developer's attention on security concerns.

Ko and Myers use the term "rule" to refer to a programmer's learned program plan/pattern. I believe ASIDE can also be effective against common patterns of rule-based breakdowns in security contexts. The first example relates to the inexperience of a programmer. Suppose a programmer, who has been trained on secure programming practices, invokes an unfamiliar API which brings untrusted input into the system. She may not be aware (i.e. she misses a "rule") that input validation is needed, leading to a software vulnerability. The second example relates to information overload, which occurs when too many rules are triggered and one of them may be missed. One can envision a situation where the rule requiring access control is not applied due to programmers' information overload. In both cases, a tool like ASIDE could be effective in preventing such errors by reminding programmers to apply security

related rules.

The third example relates to favoring a previously successful rule without realizing the change of context (39). Consider a case where a block of code which may make certain checks on file integrity is copied and reused. However, more types of checks are needed given the new application context. A developer may not be sufficiently aware of this context switch and mistakenly believe that the existing integrity checks in the code are sufficient. Again, a tool like ASIDE may be helpful to mitigate such an instance by alerting programmers of important security considerations and giving them an opportunity to discover that a different rule might be needed. This might be accomplished by requesting an annotation on file integrity check logic.

Knowledge-based activities typically are at a conceptual level, such as requirements specification and algorithm design (39). For example, it is often impossible to perform an exhaustive search of the problem space. Using black-list input validation instead of white-list input validation is an example of a knowledge-based breakdown leading to security vulnerabilities. A black-list is much easier to construct, based on human cognitive heuristics of selectivity, biased reviewing and availability. Writing proper white-list validation requires significant effort, even for common input types such as people's last name, address, URL, and file path, especially when taking internationalization issues into consideration. ASIDE helps to address this issue by making it easier to use white-list input validation, choosing from a predefined list of options.

The design of ASIDE, in general, meets three criterion. It detects vulnerable code while developers are writing code; it informs developers through warnings; and it provides suggestions to mitigate the identified problems. In the following chapters,

I discuss about two implementations of two techniques that follow this design, *interactive code refactoring* and *interactive code annotation* as well as my effort for evaluating them.

CHAPTER 5: INTERACTIVE CODE REFACTORING

The modern IDE provides rich features that significantly alleviate programmers' cognitive burden in writing code and ensures the correctness of programs. For instance, if a variable is used before being declared under the context of a static programming language, the IDE would annotate the corresponding line which identifies the statement in which the variable is with an error marker that carries information about the error. The error marker can be easily visualized by programmers. A programmer can choose to further investigate the proposed issue by hovering the cursor over the error marker or clicking on it. Upon being hovered or clicked, the error marker displays a message to explain the underlying cause of the error, along with potential solutions that can be employed to solve the problem. By clicking on the item which represents a solution the programmer considers valid and capable of solving the issue, corresponding actions such as code refactoring that have been coded in the IDE will be taken on behalf of the programmer. In the mean time, the error marker will be eliminated from the IDE. Interactive code refactoring works in a similar fashion.

5.1 Target Vulnerabilities Profile

Interactive code refactoring is designed to address security vulnerabilities that stem from improper/insufficient input validation and/or output filtering, of which many are well-known. These range from low-level command injection in systems to high-level

SQL Injection and Cross-site Scripting in modern web applications. The primary and common characteristic of such vulnerabilities is that the program takes external inputs from users, network, file systems, etc. as they are for critical or sensitive operations that can change states of the program. In some case, the program makes efforts to validate the inputs, however, does it in an insufficient manner.

In this section, I present some common vulnerabilities in modern web applications that share this characteristic.

5.1.1 Cross-site Scripting

A Cross-site Scripting (XSS) vulnerability occurs when data enters a web application through an untrusted source. In the case of Persistent (also known as Stored) XSS, the untrusted source is typically a database or other back-end datastore, while in the case of Reflected XSS it is typically a web request. It also occurs when the data is included in dynamic content that is sent to a web user without being validated for malicious content.

The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%...  
  
Statement stmt = conn.createStatement();  
  
ResultSet rs = stmt.executeQuery("select_*_from_emp_where_id="+eid);  
  
if (rs != null) {  
  
    rs.next();  
  
    String name = rs.getString("name");
```


%>

Employee Name: <%= name %>

This code functions correctly when the values of `name` are well-behaved, but it does nothing to prevent exploits if they are not. This code can appear less dangerous because the value of `name` is read from a database, whose contents are apparently managed by the application. However, if the value of `name` originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation/encoding on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

5.1.2 SQL Injection

A SQL Injection vulnerability occurs when data enters a program from an untrusted source and then is used to dynamically construct a SQL query.

The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where the owner matches the user name of the currently-authenticated user.

```

...
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT_*_FROM_items_WHERE_owner_=' "
               + userName + "'_AND_itemname_=' "
               + itemName + "'";
ResultSet rs = stmt.execute(query);
...

```

Because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if `itemName` does not contain a single-quote character. If an attacker with the user name `wiley` enters the string `"name' OR 'a'='a"` for `itemName`, then the query becomes the following:

```

SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';

```

The addition of the `OR 'a'='a'` condition causes the where clause to always evaluate to `true`, so the query becomes logically equivalent to the much simpler query:

```

SELECT * FROM items;

```

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the `items` table, regardless of their specified owner.

5.1.3 Code Injection

Many modern programming languages allow dynamic interpretation of source instructions. This capability allows programmers to perform dynamic instructions based on input received from the user. Code injection vulnerabilities occur when the programmer incorrectly assumes that instructions supplied directly from the user will perform only innocent operations, such as performing simple calculations on active user objects or otherwise modifying the user's state. However, without proper validation, a user might specify operations the programmer does not intend.

In this classic code injection example, the application implements a basic calculator that allows the user to specify commands for execution.

```
...  
    ScriptEngineManager scriptEngineManager = new ScriptEngineManager();  
    ScriptEngine scriptEngine = scriptEngineManager  
        .getEngineByExtension("js");  
    userOps = request.getParameter("operation");  
    Object result = scriptEngine.eval(userOps);  
...
```

The program behaves correctly when the `operation` parameter is a benign value, such as `"8 + 7 * 2"`, in which case the `result` variable is assigned a value of 22. However, if an attacker specifies languages operations that are both valid and malicious, those operations would be executed with the full privilege of the parent process. Such attacks are even more dangerous when the underlying language

provides access to system resources or allows execution of system commands. For example, Javascript allows invocation of Java objects; if an attacker were to specify “`java.System.Runtime.exec("shutdown -h now")`” as the value of operation, a shutdown command would be executed on the host system.

5.1.4 Log Forging

A Log Forging vulnerability occurs when data enters an application from an untrusted source and then is written to an application or system log file. The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```
String val = request.getParameter("val");

try {
    int value = Integer.parseInt(val);
}

catch (NumberFormatException) {
    log.info("Failed to parse val=" + val);
}
```

If a user submits the string “twenty-one” for `val`, the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

However, if an attacker submits the string “twenty-one%0a%0aINFO:+User+logged+out%3dbadguy”, the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

```
INFO: User logged out=badguy
```

Clearly, attackers can use this same mechanism to insert arbitrary log entries.

5.1.5 Header Manipulation

Header Manipulation vulnerabilities occur when data enters a web application through an untrusted source, most frequently an HTTP request and then is included in an HTTP response header sent to a web user without being validated

The following code segment reads the name of the author of a weblog entry, `author`, from an HTTP request and sets it in a cookie header of an HTTP response.

```
String author = request.getParameter(AUTHOR_PARAM);
...
Cookie cookie = new Cookie("author", author);
    cookie.setMaxAge(cookieExpiration);
    response.addCookie(cookie);
```

Assuming a string consisting of standard alpha-numeric characters, such as “Jing Xie”, is submitted in the request the HTTP response including this cookie might take the following form:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jing Xie
...
```

However, because the value of the cookie is formed of unvalidated user input the response will only maintain this form if the value submitted for `AUTHOR_PARAM` does not contain any CR and LF characters. If an attacker submits a malicious string, such as “Ryan Hacker \r\nHTTP/1.1 200 OK\r\n...”, then the HTTP response would be split into two responses of the following form:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Ryan Hacker

HTTP/1.1 200 OK
...
```

Clearly, the second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability of attacker to construct arbitrary HTTP responses permits a variety of resulting attacks, including: cross-user defacement, web and browser cache poisoning, cross-site scripting and page hijacking.

5.1.6 Path Manipulation

Path manipulation errors occur when an attacker can specify a path used in an operation on the filesystem and gains a capability that would not otherwise be permitted by specifying the resource.

For example, the following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could

provide a file name such as “../../tomcat/conf/server.xml”, which causes the application to delete one of its own configuration files.

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" + rName);
...
rFile.delete();
```

5.1.7 Dangerous File Inclusion

Many modern web scripting languages enable code re-use and modularization through the ability to include additional source files within one encapsulating file. This ability is often used to apply a standard look and feel to an application (templating), share functions without the need for compiled code, or break the code into smaller more manageable files. Included files are interpreted as part of the parent file and executed in the same manner. File inclusion vulnerabilities occur when the path of the included file is controlled by unvalidated user input.

The following code takes a user specified template name and includes it in the JSP page to be rendered.

```
...
<jsp:include page="<%=_(String)request.getParameter(\"template\")%>">
...

```

In the above example, an attacker can take complete control of the dynamic include statement by supplying a malicious value for `template` that causes the program to

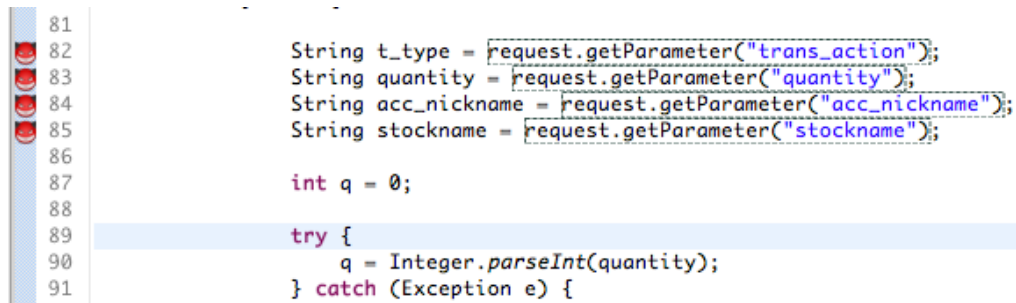
include a file from an external site.

If the attacker specifies a valid file to a dynamic include statement, the contents of that file will be passed to the JSP interpreter. In the case of a plain text file, such as `/etc/shadow`, the file might be rendered as part of the HTML output. Worse, if the attacker can specify a path to a remote site controlled by the attacker, then the dynamic include statement will execute arbitrary malicious code supplied by the attacker.

5.2 Interactive Code Refactoring

I have implemented interactive code refactoring in the form of an Eclipse plugin. My prototype implementation is analogous to how the incremental compiler works in modern IDEs. ASIDE works as a long running process in the background that scans a selected project for program patterns that match pre-defined heuristic rules of security vulnerabilities in the current Eclipse workspace. During the development session, it monitors developers' edits to respond to the changes in the code. Whenever a match is found, ASIDE marks the corresponding source code line using a warning icon on the left margin of the code editor and also highlights the identified vulnerable code using a dashed rectangle, shown in Figure 2.

As stated previously, there are many classes of vulnerabilities that originate from a variety of code patterns. However, for my research, I only consider vulnerabilities that are most known for their prevalence and commonness. More specifically, I care about OWASP Top 10 (55) and SANS Top 25 programming errors (68). In this section, I discuss an example concerning input validation to illustrate the key concepts.



```

81
82 String t_type = request.getParameter("trans_action");
83 String quantity = request.getParameter("quantity");
84 String acc_nickname = request.getParameter("acc_nickname");
85 String stockname = request.getParameter("stockname");
86
87 int q = 0;
88
89 try {
90     q = Integer.parseInt(quantity);
91 } catch (Exception e) {

```

Figure 2: ASIDE identifies vulnerable code and reminds developers through warnings.

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 XML-based, to specify sources of untrusted inputs which I formally named as *trust boundaries*. Currently two types of rules are supported: *Method (API) invocations*, for example, method `getParameter(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 3 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 4 illustrates how ASIDE refactors code to perform input validation using OWASP Enterprise Security API (ESAPI) Validator (62).

```

82 String t_type = request.getParameter("trans_action");
83 String quantity = request.getParameter("quantity");
84 String acc_nickname = request.getParameter("acc_nickname");
85 String stockname = request.getParameter("stockname");
86
87 int q = 0;
88
89 try {
90     q = Integer.parseInt(q);
91 } catch (Exception e) {
92     makeTransactionForUser
93     stockname, "In
94     return;
95 }
96
97 if (q == 0) {

```

The return value of getParameter() at line 83

30 quick fixes available:

- HTTPParameter
- HTTPURL
- HTTPHeader
- HTTPURI
- HTTPQueryString
- HTTPSESSIONID

Figure 3: The user interactively chooses the type of input to be validated using a white-list approach.

```

27 String username = req.getParameter("username");
30
31 String username = req.getParameter("username");
32 // NOTE: Input Validation code generated by ASIDE
33 try {
34     ESAPI.validator().getValidInput(
35         "replace ME with validation context", username,
36         "SafeString", 200, false);
37 } catch (ValidationException e) {
38 } catch (IntrusionException e) {
39 }

```

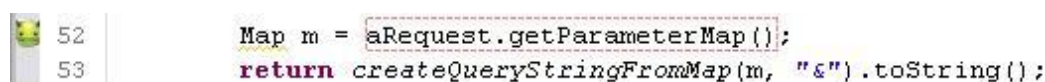
Figure 4: ASIDE validates an input using OWASP ESAPI validator API.

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 (23). In contrast, my 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 I believe is the best practice for secure programming, validating untrusted inputs at the earliest possible time (16). However, it is trivial to extend ASIDE to apply the other strategy.

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 5. 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 5 throughout the development to serve as a continual reminder.



```

52 |         Map m = aRequest.getParameterMap();
53 |         return createQueryStringFromMap(m, "&").toString();

```

Figure 5: 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 (72) 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.

My evaluation of interactive secure programming support for software developers is conducted from three perspectives: a model-theoretic analysis which has been covered in Section 4.2 , which applies theoretic design principles to features of tool under examination; an open source projects analysis presented in Section 5.3; and user studies on human subjects detailed in Section 5.4. All evaluations were conducted on the different versions of implementation of ASIDE, the prototype that embodies the idea of my proposed approach.

5.3 Open Source Projects Evaluation

In this section, I focus on input validation and/or encoding vulnerabilities, as they are currently supported by code refactoring. My 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.

5.3.1 Establishing a Baseline Using an Open Source Project

I selected Apache Roller (a full-featured blog server) (66) 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 my colleagues, 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 (72), 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 db connections) as resources may be released in a non-standard way. Often times, warnings 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.

	<i>Critical</i>	<i>High</i>	<i>Medium</i>	<i>Low</i>
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.

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

<i>Severity</i>	<i>Category Name</i>	
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

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, I exclude them in our analysis to avoid duplications.

5.3.2 Vulnerability Coverage of ASIDE

I 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.

Forty one of the 143 are taint sources of composite data returned from APIs such as `org.hibernate.Criteria.list()` and `javax.servlet.ServletRequest.getParameterMap()`. ASIDE 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 5.

While I 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.

My 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.

5.3.3 False Positives for ASIDE

As I just demonstrated, ASIDE had good coverage of the input validation/encoding issues in Roller. In this section, I discuss the additional warnings that ASIDE generated. In analyzing false positives, I 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. Ninety four 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 an exploitable security vulnerability as the software evolves. Therefore, I 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 6 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 6: Untrusted input is logged through an Exception construction.

Thus, from a broad secure programming best practice perspective, I 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 I 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 7, 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 7: Untrusted input is used for logic test.

Another such case is shown in Figure 8, 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("excerpts") != null) {
126             this.excerpts = Boolean.valueOf(request.getParameter("excerpts")).booleanValue();
127         }
128     }

```

Figure 8: Untrusted input is parsed into harmless Boolean value.

Because the false positive rate often is positively correlated to accuracy, it is diffi-

cult to design a highly accurate tool without false positives. Both traditional analysis tools, such as Fortify SCA (72), and ASIDE will require manual inspection of warnings to eliminate false positives. However, my 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, I think 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 (72) 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, I 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, I conducted further research which will be detailed in Section 5.4 to understand how false positives in ASIDE impact developer behavior.

5.4 Developer 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 and to evaluate whether real developers could use ASIDE effectively, I conducted two comparison-based user studies to evaluate our approach for helping programmers to address potential security vulnerabilities in their code.

In designing ASIDE, I proposed to use interactive code generation as one method to help programmers easily modify their code and prevent security vulnerabilities. However, I also wanted to examine an alternate approach, currently provided in part by a commercial tool (17), providing a detailed explanation of a warning. Thus, for this study, I implemented and evaluated two different versions of ASIDE. The first, ASIDE *CodeGen*, performs automated code generation to fix the identified vulnerable code. More specifically, *CodeGen* offers a developer a list of suggestions as a set of input types/output encoding strategies upon his/her examination of a warning through either clicking on the warning icon or hovering over the highlighted code as shown in Figure 9. Upon choosing a type from the list, *CodeGen* then automatically inserts the corresponding code segment which performs the input validation or output encoding, as illustrated in Figure 10. The developer can also choose to dismiss the warning if s/he does not wish to modify the code.

My other alternative, ASIDE *Explanation*, instead provides only two options upon a developer's request to address the warning. When selecting "Guide Me Through", s/he will be presented with a detailed explanation, Figure 11, as to why the code is vulnerable, the consequences of not addressing the vulnerability, and the suggested remediation. Thus, *Explanation* does not automatically create code, but instead

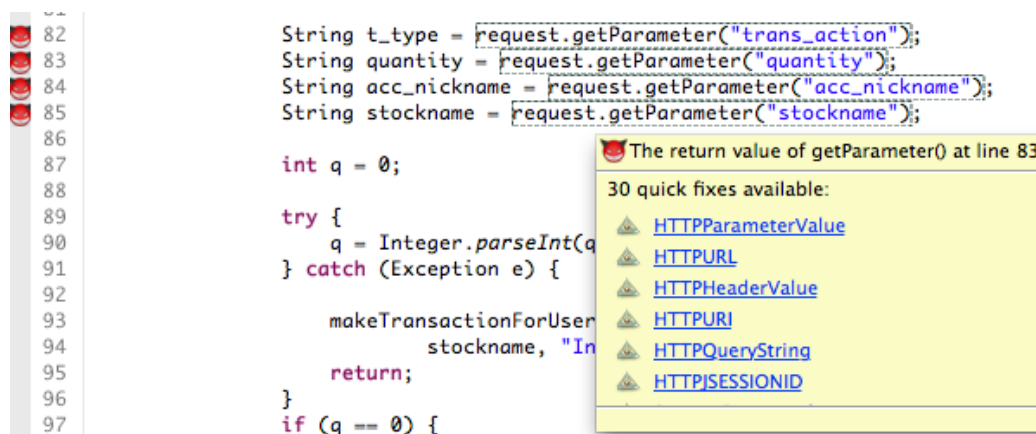


Figure 9: ASIDE CodeGen offers a list of suggestions that can be applied to address the selected warning.

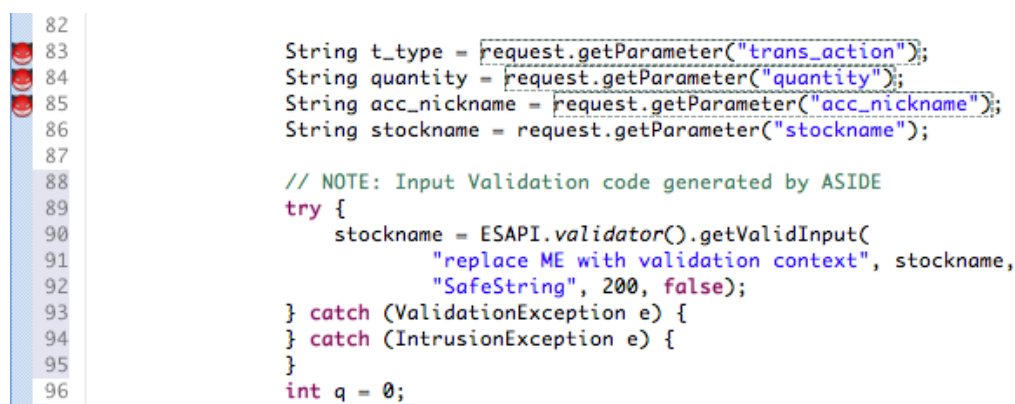


Figure 10: ASIDE CodeGen generates a code segment in response to the developer's selection of an input type.

attempts to help the programmer understand the problem and address it him/herself. The other option, "Ignore this" dismisses the warning, and removes it from view.

It is my hypothesis that *CodeGen* is more effective than *Explanation* in aiding programmers, given that *CodeGen* reduces a programmer's burden of creating his/her own validation/encoding routine. Furthermore, *CodeGen* provides a rich list of commonly-used, well-established input types from the security community, some of which may not be known to the developer. This, therefore, offers a chance for devel-

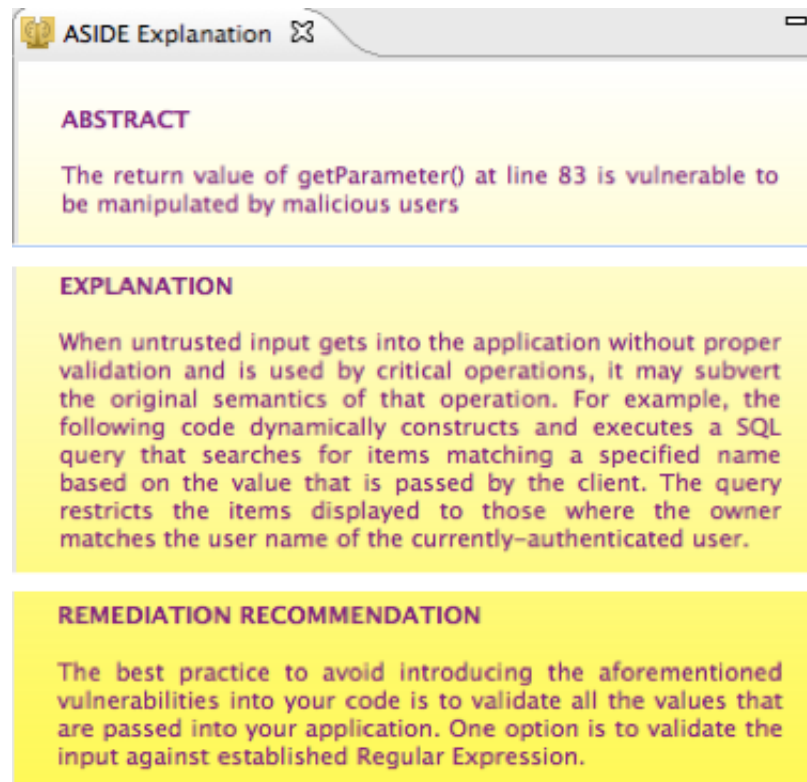


Figure 11: ASIDE Explanation provides details of the secure programming warning.

opers to expand their awareness of secure programming practices. However, programmers may not feel comfortable with code being generated for them, or understand how to modify that code if necessary. On the other hand, *Explanation* demands that the developer switch attention from programming to documentation reading, which may lead to ignoring the warning. Even worse, *Explanation* does not provide concrete solutions as to how to solve the problem beyond a more generic description of how to perform proper input validation/output encoding. As a result, the developer has to write the code from scratch, which may increase the developer's cognitive load and be challenging for someone without sufficient secure programming knowledge. However, *Explanation* would let the developer decide how to best incorporate the vulnerability

fix into the code. My studies examine the differences in these two approaches.

My approach aims to be useful for developers at all levels of their software development experience. However, novice programmers and expert programmers perceive and practice software development differently (5); thus, I believe that they possess different views towards the need of tool support for secure programming. I thus performed our study in terms of two different developer populations: novice developers who are at the stage of gaining both software development knowledge and experience; and expert developers who have years of professional experience in industry.

5.4.1 User Study I - Students/Novice Developers

5.4.1.1 Participants and Procedure

For this study, I recruited 18 students from a graduate level Java-based web application development course, which was offered by my college in the Spring 2011 semester. Twelve students were male and 6 female. As part of the course, students were briefly introduced to basic secure programming techniques such as input validation and output encoding. However, project grades were assigned only based on functional requirements, not on secure programming practices.

Part of the students' course work was to build an online stock trading system incrementally over four projects throughout the semester using Java Servlet technology. My 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 stock 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.

5.4.1.2 Study setup

I performed a controlled comparison study, where student participants were asked to come to a lab and work on their assignment for 3 hours. Participants were randomly assigned to work with either *CodeGen* or *Explanation*. Each study session took 3 hours of development and fifteen minutes of debriefing. Participants were offered a gift card as compensation for completing the study session.

The participants were first given a brief walkthrough of how the selected plugin works. In the mean time, a technical assistant helped set up the development environment and recording software on the lab machine. He also imported the participant's existing project into Eclipse and made sure it was compiled and could run on the local web server. The participant was then asked to launch the plugin before proceeding to development. Students were told to work as they wished on the assignment for 3 hours and respond to ASIDE warnings as they wanted to. The participants worked on various parts of their code and none completed the assignment during the study session.

5.4.1.3 Semi-structured Interview

Each interview lasted 10 to 15 minutes. I began by asking participants to describe their interactions with ASIDE, and the concrete actions they took to address the warnings and why. Next, I asked our participants to explain what they liked and disliked most about the tool that they interacted with. I also asked participants whether they thought they would likely pay attention to and fix vulnerable code

without such tool support. Finally, I asked each participant whether s/he gained any knowledge from interacting with the tool.

5.4.1.4 Result Analysis

For each participant, I collected a 3-hour screen recording of his/her application development and a recording of the interview. Additionally, I gathered a software log of the participant's interaction with ASIDE as a supplement to the screen recording. I analyzed the screen recording and logs, focusing on the many warnings ASIDE generated. For each warning, I analyzed how participants responded either through the tool or in their code.

The interviews provided the participants' explanations for their behavior with ASIDE. I transcribed all interviews into text, and performed open-coding on the transcripts using. I identified general themes and interesting cases about their perceptions towards tool support for secure programming.

Figure 12 depicts the results of the 9 participants who worked with ASIDE *CodeGen*. Over all nine participants, 101 distinctive *CodeGen* warnings were generated, resulting in 11.2 warnings per participant on average. There were 83 warnings clicked on by participants (82%), or 9.2 for each participant on average.

Out of the 83 warnings clicked, 63 were addressed (76%, or 7 per participant on average) by clicking on one of the input/output types provided by *CodeGen*, leading to code being generated. The remaining warnings were deliberately dismissed by participants through the provided "Ignore this" option. All participants used *CodeGen* to generate code and none of them wrote any customized validation or encoding rou-

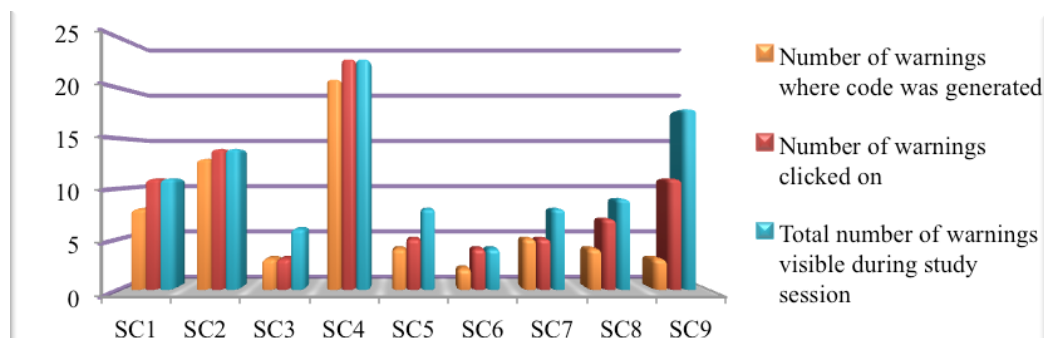


Figure 12: Metrics from students with *CodeGen*.

tines. Thus, *CodeGen*'s interactive code generation 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 acted upon. Some of the warnings were generated when the participant wrote debugging code, which was soon deleted. Perhaps participants ignored security warnings on code that they knew was transient. Other cases have to do with a bug in the version of *CodeGen* used, which falsely warned participants of the need for output encoding. I 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 *CodeGen* warnings. Thus, in cases where participants encountered false positives, they were able to recognize them quickly and dismiss them.

ASIDE *CodeGen*'s code generation is designed to be intuitive and unobtrusive. In most cases, it worked just as I expected, quickly providing useful code fixes. However, in two cases the generated code caused difficulties. In both cases, the participant was validating a password string passed from the client via an HTTP request. The rule used by the validation code enforces certain restrictions on that string. However, the

test data used by the students did not meet those strict restrictions (for example, one used the test string “password”). Thus, the participants were no longer able to execute and test their code because the test password failed validation. The solution would have been to create a new test password. However, this interrupted the participants flow of development and instead both participants deleted the generated validation routine.

Figure 13 depicts an overview of results of participants using ASIDE *Explanation*. The 9 participants generated in total 93 warnings, of which 68 were clicked on (73% or 7.5 per person). In contrast to CodeGen warnings, 87% (59/68) of *Explanation* warnings were deliberately dismissed by clicking on “Ignore this”. Throughout the sessions, only 20 warnings were examined by clicking on “Guide Me Through”, which then provides the more detailed explanation. While most participants read at least one warning, none of the participants wrote any validation routines to check the identified vulnerable code.

As shown by the results, the warnings shown by ASIDE during the programming process caught participants’ attention. No participant from the two groups just ignored all the warnings during his/her development, and users tended to deliberately dismiss any they were not going to attend to. However, only *CodeGen* resulted in any security validation being added to the code during the session.

Not only did students in both groups notice the warnings, they also reported welcoming the idea of being reminded of security vulnerabilities in their code in real time. Many expressed similar comments, such as:

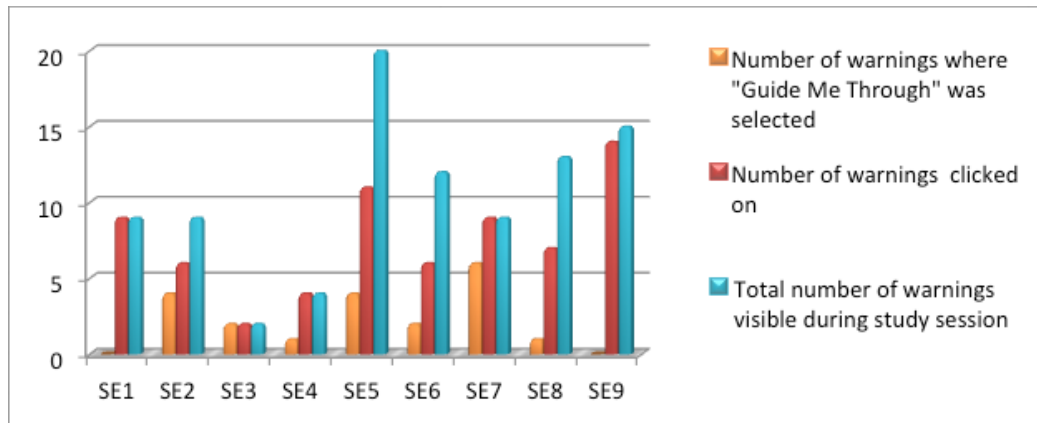


Figure 13: Metrics from students with *Explanation*.

[SE5] Helpful, definitely helpful. It's good at the moment to use it, they come up immediately, it's not like you have to understand them later on. At the moment, if you do something wrong, it shows a warning symbol on the left.

I was concerned that warnings may divert programmers from their development flow, and thus be considered annoying. But no one reported being bothered by the warnings because they are inline with the code, not obtrusive, and do not prevent programs from being compiled.

[SC4] No (it does not bother me). It gives me warnings so that I can write secure code.

Still, I observed participants deliberately dismissing warnings rather than simply ignoring them, and leaving them on the screen.

I purposefully used the red devil-looking icon to indicate the importance of the warning, and thus to effectively attract users' attention. Two participants expressed their like of the icon because it conveyed the seriousness of the issue. However, one

disliked the icon appearance, feeling that it is misleading since these warnings will not prevent programs from being compiled and thus should not be falsely presented to be as serious as compilation errors.

All but one of the participants indicated that if they were not given either *CodeGen* or *Explanation*, they would not have been aware of security vulnerabilities in their code. As one participant stated:

[SC1] That (the warning design of CodeGen) was good because hadn't it prompted me, I wouldn't have realized I have to inspect those input values.

Similar to the behavior I observed with the *Explanation* participants, the *CodeGen* participants reported that in the absence of auto-generated code, they would not be likely to take the initiative to write their own secure code, and would prefer the code generation. The participants who used *CodeGen* all reported trusting that using the tool would make their code more secure. *CodeGen* transparently showed the regular expressions used for each data type and the code can be viewed and further edited by users. Thus, most participants did report feeling like they had sufficient control over the code generation. In addition, several reported believing that the regular-expression input validation *CodeGen* used is more sophisticated than what they would come up with on their own.

[SC8] For example, the initial code for assignment 3, assignment 2, I myself did some password validation code in JavaScript for example. Simple code, something like password length and some special characters. I mean I used such thing but this was more sophisticated because it covered ev-

everything in a simple form. If you see my code it was there are a lot of if statements for each and every condition. So this one is much simpler.

However, one participant did feel uncomfortable with the auto-generated code because he was not able to understand why the code was generated and how the generated code was impacting his existing code. One participant also worried that the code generated by *CodeGen* would not work in a different development environment. He tried out the code generation and thought it worked fine, but later deleted it out of concern it would not work when he later finished his assignment in NetBeans (58).

Participants mentioned several aspects of *CodeGen* that need to be improved. For instance, the most wanted feature was more information about why code was vulnerable, and what the generated code does. One participant suggested providing a demo package to explain and illustrate why certain programming practices are not secure.

Even though none of the students who worked with *Explanation* wrote any customized validation/encoding routines, many of them still thought it was helpful since the explanation view gave them an idea as to why the warning showed up.

[SE5] So I realize that through out my code, there are a lot of points where possible, malicious users can take advantage of weakness in the code. So it's pretty interesting to know that, something looks so simple, someone can actually try to, find loophole and something as small as that. I didn't know these things; it's just interesting to see.

SE5 further acknowledged that he was motivated to learn more about input validation using regular expressions. Interestingly, he even took notes about what he read

on the explanation view.

While some thought *Explanation* was useful and helpful, they complained that the explanation view did not provide a concrete example as to how to address the problem. Instead, the suggestion given was too generic and abstract. Also, the content of the explanation contained too much security jargon. Despite using graduate students who had at least brief course content on input validation and output encoding, the participants still had less security knowledge than we expected, and thus the content needed to be written for a more novice audience. These results indicate a challenge that applies to either tool: how to explain the secure programming issues and present the content in language and examples that a broad programmer population will find understandable and useful.

5.4.2 User Study II - Professional Developers

The other type of potential users our approach targets is professional software developers who are experienced with application development. Thus, in addition to studying ASIDE on novice programmers, I ran a similar comparison study on a group of expert programmers.

5.4.2.1 Participants and Procedure

With the help from my advisor, Dr. Bill Chu, I was able to recruit in total 9 professional developers through personal contacts and referrals of personal contacts, 8 males and 1 female. The study was conducted over the course of 3 months during summer 2011. The participants' professional experience in the software industry ranged from one year up to 20 years, with an average of 10 years. All of them had professional

programming experience with developing web applications using Java technologies, although some of them do not currently directly work with Java Servlet (56) used in our study application development. Most (5/8) of my participants admitted that they do not pay additional attention or perform extra practices to ensure that the code written by them is secure. Only two participants were ever offered training by their employer on software security, but this did not include secure programming practices.

Instead of being brought into a controlled laboratory, these participants worked on one of 2 laptops provided by me borrowed from my university at their pace and schedule during a period of up to 7 days. However, I only required them to work on the development for a cumulative 3 hours regardless of how much functionality they could implement during the given time frame. The professionals worked with either ASIDE *CodeGen* or ASIDE *Explanation* depending on the laptop they worked on, which was circulated unpredictably. They were interviewed by phone after they returned the laptops. Participants were offered the same gift cards as the previous study when they completed the study session.

I wanted to sufficiently motivate the programmers to write code, not just interact with my tool. Unlike the students, they did not have intrinsic motivation to complete an assignment. Thus, to avoid priming the professional developer participants into thinking that I was only testing ASIDE, I informed them that the purpose of the study was to show students in my college how professional developers develop an application. I did require the participants to run ASIDE, and encouraged them to interact with it if they wished. But I did not provide any additional explanation of the tool. Thus, unlike the students, these participants had no brief tutorial and less

priming than the students as to the purpose of the study.

5.4.2.2 Task Structure

I asked the professional developer participants to develop the same stock application as described in the students' study. However, instead of composing the application from scratch, developers were provided with a base project, which had functionality including login, logout, registration and others already implemented. Static resources such as simplified html pages were also provided along with the base project. To create other web page interfaces, the participant only needed to copy and paste code and modify. Based on the lessons from the student study that 3 hours is far from enough to complete all designated functions, I arranged the specification in a way that the functionality they were asked to implement contained the most input code, and was thus most likely to have ASIDE warnings.

5.4.2.3 Study Results

I gathered the same forms of data from the professional developer's participation as was collected from the students, namely, a 3-hour development screen recording, an interview recording, an ASIDE log and the resultant project artifact. My results are based on 8 valid data points out of 9 professional developers' participation, of which 4 worked with *CodeGen* and the other 4 worked with *Explanation*. The ninth participant failed to use ASIDE at all, which meant her data was not useful for this study and she was unable to be interviewed about ASIDE. I performed the same data analysis procedure as we did with the student study.

The four professional developer participants who worked with *CodeGen* generated

in total 45 ASIDE warnings, out of which, 12 were clicked on (27% or 3 per person). This suggests that warnings were effective to at least attract some attention. However, only one participant (PDC3), as illustrated in Figure 14, used *CodeGen*'s code generation to validate the identified untrusted inputs.

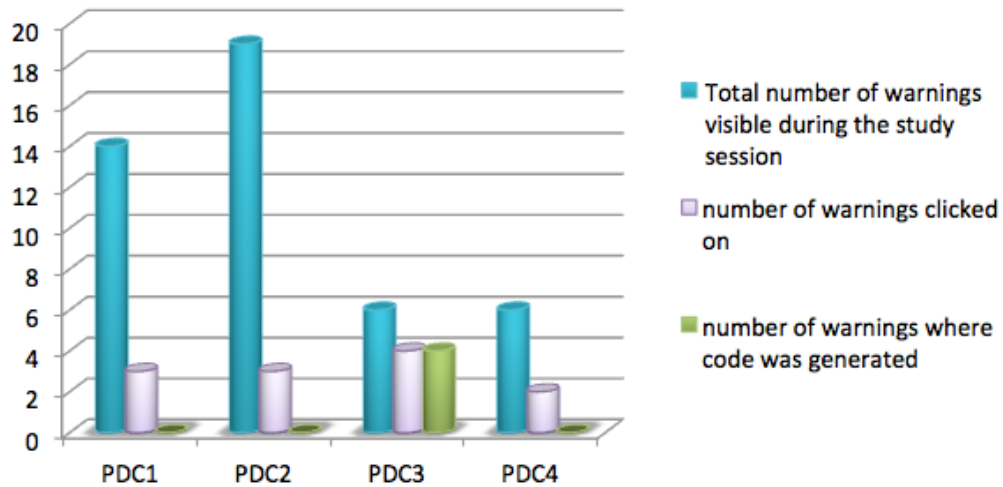


Figure 14: Metrics for *Professional Developers* with *CodeGen*.

Although PDC3 utilized *CodeGen* to address multiple warnings, he went through a fair amount of trouble to get everything working correctly. He selected the wrong input type and admitted during the interview that he was still not sure what the code does because he was not familiar with the ESAPI (59) library *CodeGen* uses to perform secure input validation and/or encoding. Both issues again indicate that *CodeGen* needs additional explanatory information to help users understand how it functions.

The other three participants from the *CodeGen* group neither used *CodeGen* to address the warnings nor put the effort into creating their own validation routines. A common reason given by them for not further interacting with *CodeGen* was that the

given 3-hour time frame was too stringent for the development. For example, PDC2 stated:

[PDC2] No, (I didn't take any actions to address the warnings). That's because of, that's what I said, the time, because I only had 3 hours. Had I had more time, I probably would've. I left a few comments that I should probably fix this but, I was just trying to get some of (the functionalities implemented).

As a substitute, these participants left comments in the code that indicated what needs to be done to address those warnings. Some of them added what they intended to do into the TODO list. Thus, the developers did seem to be influenced by the tool in perceiving that the validation should be added at some point, even though they chose not to address it immediately.

However, despite only being asked to implement as much functionality as they can, developers still felt compelled to achieve a functioning application at the cost of ignoring other concerns, which may also parallel the time pressures in the real world. This contrasted with the students who worked much longer than the 3 hours we observed them, and thus perhaps did not feel as much time pressure during the study.

The other four professional developer participants worked with Explanation. As shown in Figure 15, overall, they encountered 63 *Explanation* warnings and clicked on 14 of them (22% or 3.5 per person). All four participants interacted with *Explanation* through the “Guide Me Through” option at least once. Moreover, two participants

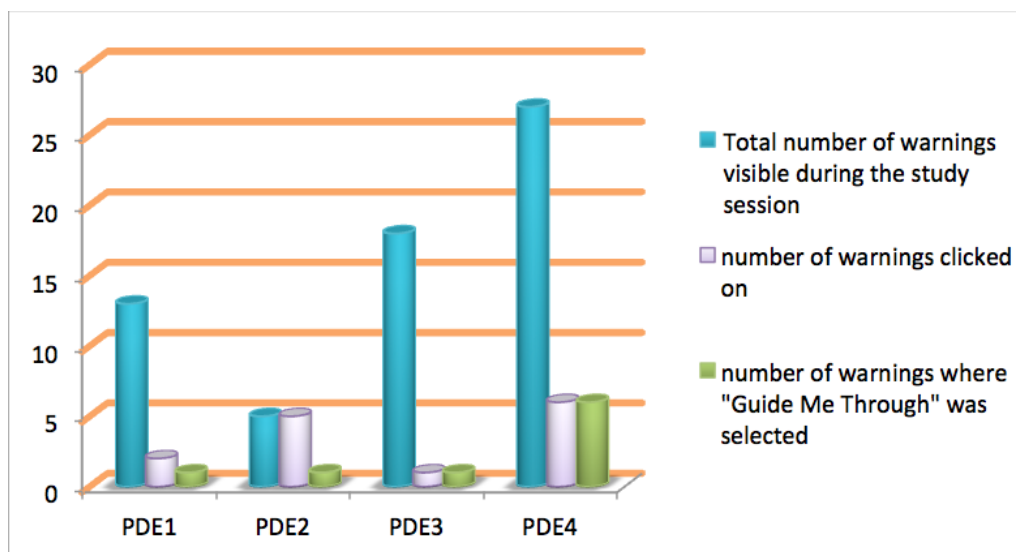


Figure 15: Metrics for *Professional Developers with Explanation*.

(PDE2 and PDE4) made the effort to write customized validation routines to check identified untrusted inputs. However, these routines were far from sufficient to prevent the applications from being compromised. Thus, their attempts were not fully correct.

The two who decided not to write validation code provided several reasons. For example, PDE1 pointed out that the application he developed for the study was not realistic, and in a realistic setting, other security controls in addition to just validating inputs against regular expressions (as suggested by *Explanation*) would be used and needed, such as limiting the login attempts and logging all the login attempts for later auditing. Furthermore, he failed to relate Java Servlet, the technology used in our study development, to the more current application development frameworks that he uses:

[PDE1] The industry does not use raw servlets anymore. Instead, the industry has other technologies such as Struts, Spring MVC, and they have components that can be directly invoked and used for validation purpose.

PDE2 also reasoned that the development process in an organization setting discourages developers from addressing code that has security implications individually because the development team may have more structured countermeasures that take care of the issues.

[PDE2] There are many ways to protect the application, even in a line of code, it looks like a security hole, but at other places we prevented it, so it will not be an issue to release this piece of code.

During the interview, all 8 participants did state that they appreciated the concept of real-time warning of secure programming issues and most acknowledged that the on-the-spot warnings made them aware of the need for input validation. For example, PDC2 left a comment in the source code about liking *CodeGen* while he was working, and another participant said that what he liked most is its ability to identify problematic areas:

[PDE2] If Explanation didn't give me warnings, I would not pay attention to the vulnerable code.

However, ASIDE is currently a demonstration prototype, working on only one type of secure programming issue, namely, lack of input validation/encoding. Thus, several participants could not generalize the idea of integrating secure programming support in the IDE from the current prototype implementation to their professional context. These participants expressed concerns over the lack of functionality in ASIDE and questioned the feasibility of applying the approach to more mainstream frameworks.

Thus, the developers' reactions were considerably more skeptical than the students' perceptions.

5.4.2.4 Study Discussion

The real-time detection and warning mechanism of ASIDE was universally accepted by both students and professional developers. Almost all participants responded in some way to the warnings and interacted with ASIDE, and perceived the warnings as a useful way to increase awareness of security vulnerabilities in the code. Participants also almost universally lacked detailed knowledge of secure programming practices, which meant that while they gained knowledge through using ASIDE, they also struggled to fully understand either the code generated or the explanation provided. Thus, any tool targeted at improving secure programming for developers needs to be able to be used and learned by those with little prior secure programming knowledge.

The real-time, interactive warnings were seen as sufficiently unobtrusive, and not annoying at least at the volume generated in my studies. Participants reported appreciating the instant feedback alongside the code they were currently working on. However, warnings were still deliberately dismissed when participants chose not to respond. So such warnings need to be easily removed from the view. This, however, could have negative consequences in that programmers will not be further reminded of any errors that they decided to ignore, and ASIDE does not currently provide any option of showing previously dismissed warnings. I may need to periodically remind users of unaddressed vulnerabilities, such as at the launch of the IDE, and to allow users to re-scan previous code.

My two studies confirmed a main challenge of secure programming: as a non-functional requirement, functionality will often trump security. Not surprisingly, all participants, but particularly the professional developers, were highly motivated to get the code functionally working, and not very motivated to spend time on secure programming. While several professionals did use ASIDE to make note of places that needed later validation, as one stated:

[PDE2] I looked at the requirements and they didn't say you have to pay attention to security problems. I just wanted to get it [the application] done.

Similar to the results from my interview study in Chapter 3, where security was attended to only when specifically dictated by the design, clients, or regulations, PDE2 did not find security specifically addressed in the requirements specification I provided, and so focused only on the functionality.

ASIDE *CodeGen* was not as effective as expected. One important lesson was that the tool was designed with an expectation that the user had at least minimal knowledge of secure programming. I expected the tool to serve as a reinforcement and reminder of this knowledge, and as a method to reduce the burden of producing code, making secure programming easier and more efficient. But neither group had sufficient knowledge for the current prototype, which caused confusion. Without prior knowledge of code security vulnerabilities, participants were not able to gain a full understanding of the warnings and how the generated code functioned. Users need functionality that provides more awareness, explanation, and training.

Interestingly, students seemed to trust the code generated by *CodeGen*, and used it more often than the professional developers. The professionals had difficulty relating the simplistic prototype to their real-world context. Given our small user population and experimental setup, it's still not clear how or when professional developers would trust and rely on generated code for security vulnerabilities. The professional developers also related similar concerns as my interview study in Chapter 3 noted, that there are other ways that organizations deal with such issues than the code fixes performed by or encouraged by ASIDE.

ASIDE *Explanation* was also seen as helpful, and the content of the warnings was read in some detail by most participants. However, this approach did not result in any successful secure code. No students attempted to create their own validation routines. While two professionals did, they did not get that functionality correct. Thus, explanation is not likely to be as effective as code generation in reducing warnings, particularly if the code generation technique can be augmented with more useful informational content.

The results of my study indicate that real-time warnings and code generation may be helpful for non-functional programming requirements, such as reducing secure programming vulnerabilities. The results of the studies also highlight several key design issues and necessary modifications to ASIDE.

- The increased efficiency and reduced cognitive burden do seem to be important for programmers to be willing to take the time to address security vulnerabilities while implementing functionality. Thus, automated code generation, where

possible, is likely to improve software security if it is quick and easy.

- Security and secure programming are concepts with many technical details. Many programmers have little to no background in specific vulnerabilities, tools, and practices. Thus, tool interactions and explanations need to help people learn and understand how and why to use the tool. This may also be necessary so that professionals trust any advice given and code generated. At the same time, once learned, efficiency will still be critical so explanations and help should be available when needed and unobtrusive when not.
- Users do not mind real-time warnings, but do not seem to want them to persist, even if they choose to ignore them.
- Even when creating secure code is relatively easy, such as through using *Code-Gen*, users still need to be motivated to make needed changes. This motivation may depend on organizational factors that encourage use amongst developers, and discourage developers from relying solely on other processes or people to handle all security concerns.

5.4.3 Limitation

My studies are experimental comparison studies, on non-production applications, which has a variety of limitations. First, the participants did not work in their normal development context, which may have added confusion and modified their behavior. All participants were also unfamiliar with ASIDE, and had very little training and exposure even during a 3-hour session. While students did use their own code, all participants understood that the application was a classroom exercise, and

was not going to be deployed with real users. As a result, participants may have paid less attention to ASIDE warnings as secure programming errors would have no real impact. Alternatively, the student participants may have paid more attention to the warnings because they were more aware that the study was related to ASIDE.

My studies were also relatively small, with 27 participants across both groups of participants. While this limits the general conclusions we can make, I believe that our observations on real programming tasks are still valuable and can inform further research. Finally, ASIDE is still only a limited functionality prototype. The unsophisticated implementation may have discouraged participants, particularly the professional developers, from interacting with and exploring the secure programming support.

Despite these limitations, I believe that such a lab-based study is a good first step at understanding the potential usefulness and impact of interactive warnings and code generation. The lessons I learned will help us improve the design of the interactive mechanisms, and encourage me to continue to develop ASIDE into a more functional and deployable system that I can evaluate on a larger and more realistic scale.

CHAPTER 6: INTERACTIVE CODE ANNOTATION

Interactive code annotation is a mechanism that helps developers to avoid more subtle security vulnerabilities where code refactoring described in Chapter 5 is not feasible, such as broken access control and Cross-Site Request Forgery (CSRF) (63).

Having developers providing programming considerations via annotations is very powerful. For instance, Microsoft discovered that having developers annotate limits of buffers effectively reduced buffer overflow vulnerabilities (28; 29). However, the annotation language used not only takes the form of extra textual syntax (e.g (28)), but also adds an additional task for the developers to perform, thus increasing developers' cognitive burden of developing software. Moreover, conducting code reviews to check the presence/absence of annotations is time consuming, especially in a large code base. It is also cost prohibitive to conduct face-to-face code reviews with developers.

Interactive code annotation works in a different fashion as follows. When potential vulnerable code that may lead to broken access control or CSRF vulnerabilities are detected, programmers are asked to indicate where the corresponding preventive practices were performed. The programmers may answer the questions by adding an annotation to the code that performs the practice. The relationship between a question and its answer is recorded for further analysis. This serves as both a reminder to perform best secure programming practices, and enables further analysis

and auditing.

6.1 Target Vulnerabilities Profile

It is fairly easy to provide a concise and general specification that captures the essential characteristics of the vulnerabilities that are described in Section 5.1, such as Cross-site Scripting, SQL Injection, Command Injection, Log Forging, etc. Given a programming environment, it is possible to specify a set of functions that read inputs that are potentially untrusted (called sources), a set of functions that represent security sensitive operations (called sinks), such as inserting information into a log, and a set of functions that check data for malicious content. However, not all vulnerabilities fit into this profile. In particular, I look at vulnerabilities that result from errors in the logic of a web application. Such errors are typically specific to a particular web application, and might be domain-specific. For example, consider an online store web application that allows users to use coupons to obtain a discount on certain items. In principle, a coupon can be used only once, but an error in the implementation of the application allows an attacker to apply a coupon an arbitrary number of times, reducing the price to zero, or even a negative number if another error such as missing check on whether the total price is positive exists.

Another example shown below in Figure 16 is a servlet that processes a request via method invocation `performActionOnCriticalData (HttpServletRequest , HttpServletResponse)` on a `User` instance only when the user is logged in and has certain privilege. In the code, however, the developer only checked to see whether the user has an active and valid session before invoking `performActionOnCritical`

```

23 @Override
24 protected void doPost(HttpServletRequest req, HttpServletResponse resp)
25     throws ServletException, IOException {
26     HttpSession session = req.getSession(true);
27
28     User user = (User) session.getAttribute("User");
29     if(user != null){
30
31         User.performActionOnCriticalData(req, resp);
32
33         // more code on process other logic
34     }
35
36     /*
37     * code performs other logic of the application.
38     */
39
40 }

```

Figure 16: An example with a broken access control vulnerability.

Data (req, resp) to process the request, which opens the door for non-privileged users to use privileged functions and may access privileged data of the application as a result.

If a certain function, in this case, the `performActionOnCriticalData (HttpServletRequest, HttpServletResponse)` method, should only be executed when an entity, in this context, the user with an admin privilege, the code should perform a check explicitly to see whether the user has the admin privilege in order to avoid access control bypass. This can be done by adding a test condition, `isPrivileged()` at line 29, as shown below in Figure 17.

6.2 Interactive Code Annotation

To illustrate how *interactive code annotation* works in practice, I use an example of access control as follows to show the workflow. Consider an online banking application with four database tables with their primary keys underlined in table 3: *user*, *account*, *account_user*, and *transaction*, where the tables *account* and *transaction* are specified

```

23 @Override
24 protected void doPost(HttpServletRequest req, HttpServletResponse resp)
25     throws ServletException, IOException {
26     HttpSession session = req.getSession(true);
27
28     User user = (User) session.getAttribute("User");
29     if(user != null && user.isPrivileged()){
30
31         User.performActionOnCriticalData(req, resp);
32
33         // more code on process other logic
34     }
35
36     /*
37     * code performs other logic of the application.
38     */
39
40 }

```

Figure 17: A solution to the broken access control issue in Figure 16.

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

Table 3: *Access control tables* for the example online banking application.

<i>user</i> (<u>username</u> , role, surname, givenName)
<i>account</i> (<u>accountNumber</u> , nickname, balance)
<i>account_user</i> (<u>accountName</u> , <u>username</u>)
<i>transaction</i> (<u>id</u> , accountNumber, date, payee, amount)

Figure 18(b) shows a highlighted line of code in the `doGet()` method of the accounts servlet, which contains a query to table *account*. *CodeAnnotate* would request the developer to identify the authentication logic in the program, for instance, using a red marker and highlighted text in the editing window. In this case, the developer would locate and highlight a test condition `request.getSession().getAttribute("USER") == null` as illustrated in Figure 18(a), which is saved by *CodeAnnotate* as an annotation to the query code. The annotations could be reviewed and modified in an additional view as shown in Figure 18(b) and Figure 18(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.

This annotation mechanism provides several benefits for developers. First, developers are reminded of the need to perform authentication and/or authorization. The annotating process may help a developer to verify that intended authentication and/or authorization logics have been included. The developer has an opportunity to add an intended access control logic should that be missing. Second, the logged annotations provide valuable information for code review. For instance, by looking at the annotations, a security auditor can get a picture of the application's access control logic without wading through the code base. Third, heuristics-based static analysis can be performed to provide more in-depth analysis of the access control logic. For such in-depth static analysis, I will specifically look 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. I 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.

6.3 Walkthrough Evaluation

In order to demonstrate that the proposed mechanism has the potential to help developers avoid writing insecure code that has broken access control and/or CSRF vulnerabilities, I have conceptually tested this idea on real world open source projects: *Apache Roller* (66) and *Moodle* (49).


```

43 response.setDateHeader("Expires", 0);
44 response.setHeader("Pragma", "no-cache");
45 response.setHeader("Cache-control", "no-store");
46
47 if (request.getSession().getAttribute("USER") == null) {
48     logger.warn("User not authenticated");
49     response.sendRedirect(request.getContextPath() + "/login.jsp");
50 } else {
51     User user = (User)request.getSession().getAttribute("USER");
52     if ("ADVISOR".equals(user.getRole())) {
53         logger.info("Role is ADVISOR");
54         request.setAttribute("MESSAGE", "Your role is ADVISOR. You will not be able to transfer funds.");
55     } else {

```

(a) Developer identifies authentication logic (highlighted text) upon request from the ASIDE (see marker and highlighted text of Figure 18(b)) and annotates it.

```

59     try {
60         session = DBUtil.getSqlMapper().openSession();
61         AccountMapper accountMapper = session.getMapper(AccountMapper.class);
62         logger.info("Getting accounts");
63         List<Account> myAccounts = accountMapper.myAccounts((User)request.getSession().getAttribute("USER"));
64         logger.debug("Got accounts of size {}", myAccounts.size());
65         request.getSession().setAttribute("ACCOUNTS", myAccounts);
66     } finally {
67         if (session != null) {
68             try { session.close(); } catch (Exception ex) {}
69         }
70     }
71     request.setAttribute("ACCOUNTS", myAccounts);

```

(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.

The screenshot shows the 'ASIDE Questions and Answers' window. It contains a question 'Q1 at line 63 in AccountsServlet.java' and an answer 'A1 at line 47 in AccountsServlet.java'. The question detail asks: 'Where is the authentication process for code List<Account> myAccounts = accountMapper.myAccounts((User)request.getSession().getAttribute("USER")); at line 63 in AccountsServlet.java?'. The answer detail provides a code snippet: 'Code Snippet if (request.getSession().getAttribute("USER") == null) { logger.warn("User not authenticated"); response.sendRedirect(request.getContextPath() + "/login.jsp"); } at line 37 in AccountsServlet.java does the authentication.'

(c) The *Annotation view* adds the annotated information as a response to the developer's annotating of the access control logic in above Figure 18(a).

Figure 18: An example showing how ASIDE interactive code annotation works.

Table 4: Security issues documented for Roller and Moodle.

	Fixed issues with detailed information	Code Refactoring	Code Annotation
<i>Roller</i>	4	3	1
<i>Moodle</i>	16	1	2

The security audit performed in Section 5.3.1 did not identify any broken access control or CSRF issues in *Roller*. Thus, I turned to bug tracking records to uncover previously discovered issues. Since *Apache Roller* only has a small number of fully documented security patches, I also included security patch information from *Moodle* (49), 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, as recorded in Table 4. Four of them are due to improper input validation and/or encoding, which 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* (66). The authenticator, as illustrated in Figure 20, gets a web request from the client 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. There is, however, another path from the web entry point to the data access point where the headers do not conform to the expected format, as shown in the control flow diagram in Figure 19.

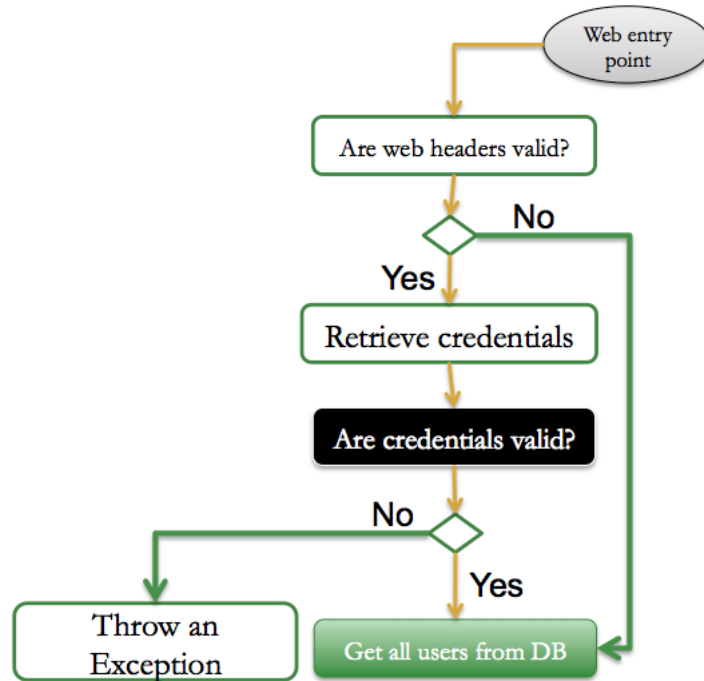


Figure 19: Control flow diagram of how an authentication request is processed.

According to the logic of ASIDE code annotation, when the application code accesses the protected database resource to get all users' information, *CodeAnnotate* 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 21. 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 20. In this case, there are three tests to be annotated.

Based on the developer's annotation, *CodeAnnotate* would be able to construct a graph, as illustrated in Figure 21, that has one path from a web entry point to a data access point with an annotated access control check on it, while another path from

```

46 protected void verifyUser() throws HandlerException {
47     try {
48         UserData user = getRoller().getUserManager().getUserByUserName(getUserName());
49         if (user != null && user.hasRole("admin") && user.getEnabled().booleanValue()) {
50             // success! no exception
51         } else {
52             throw new UnauthorizedException("ERROR: User must have the admin role to use
53     }

```

Figure 20: Annotate access control logics.

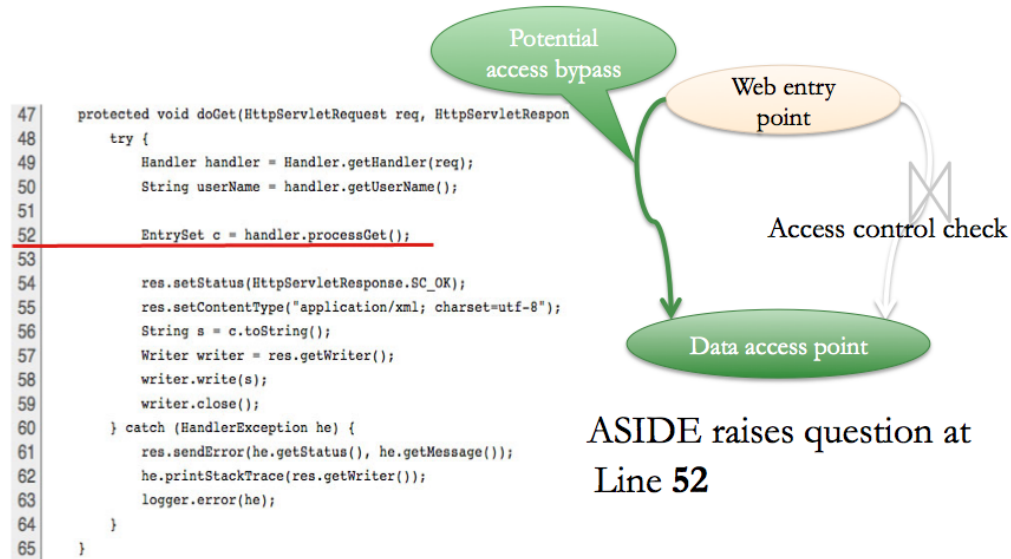


Figure 21: Java Servlet code for processing authentication request (left) and Access control check graph that involves processing the request (right).

the same entry point to the same data access point has no access control check on it. Therefore, *CodeAnnotate* would be able to provide a warning to the developer of a potential broken access control vulnerability.

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 development team is clearly well aware of the CRSF problem since they have implemented this strategy as a set of standard utility functions which drastically simplifies developers' tasks. However,

developers still missed using this CSRF protection, introducing serious vulnerabilities into the software.

CodeAnnotate 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. I describe one of the CSRF cases in Moodle, MSA-08-0013 (50), 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, I 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 22, 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.

6.4 CodeAnnotate

In this section, I describe a more comprehensive User Interface (UI) design which has been implemented in the prototype, *CodeAnnotate*, to illustrate the interactive

```

$userold = get_record('user','id',$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 22: A snippet of source code of the web transaction that changes user profile.

details of *interactive code annotation*.

6.4.1 CodeAnnotate User Interface Design

CodeAnnotate is activated when the user proactively issues the command on a selected project by clicking on the menu item “ASIDE CodeAnnotate” from the context menu, shown in Figure 23. This command only needs to be executed once for each active development session. The termination of a development session by shutting down the Eclipse workspace will terminate the process of *CodeAnnotate*.

Once activated, *CodeAnnotate* starts finding sensitive operations in the project that are matched with specified sensitive operation patterns. These sensitive operation patterns are based on predefined rules, coded as a XML configuration file, for common APIs that make changes to a datastore, application state, etc. An example of such an API is `java.sql.Statement.executeUpdate(String)` on Java platform. *CodeAnnotate* also allows one to plug in a custom rules class that reflects one’s company’s internal needs. In a large corporate setting, a SSG can be responsible for creating and plugging in the custom ruleset. For each sensitive operation, *CodeAnnotate* evaluates whether it can be reached by a program entry point, for in-

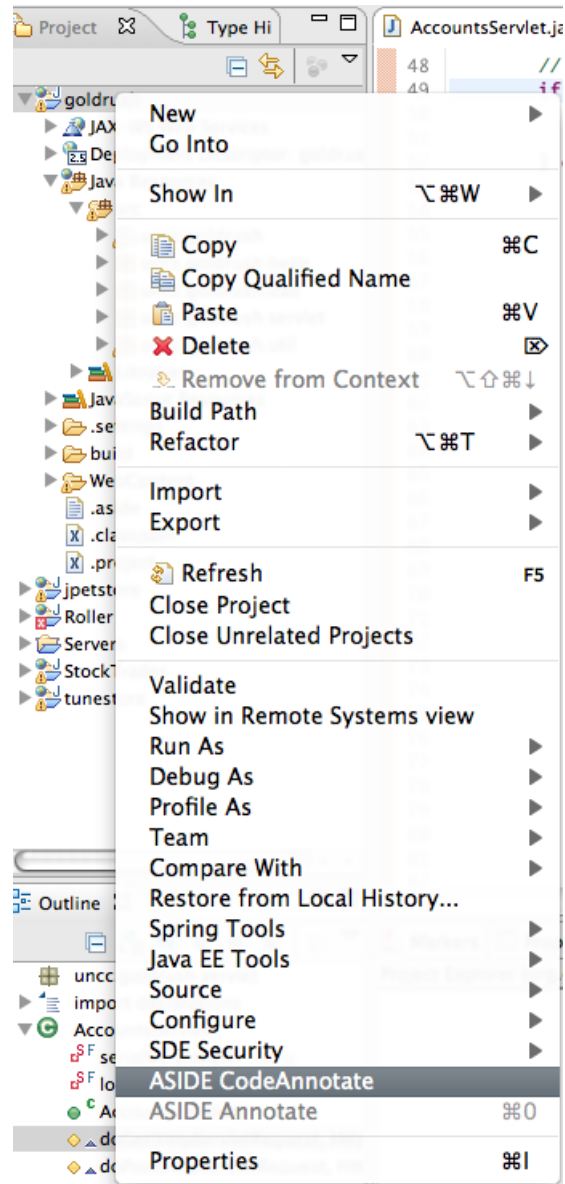


Figure 23: CodeAnnotate has a menu item on the context menu, through which it can be launched against a selected project.

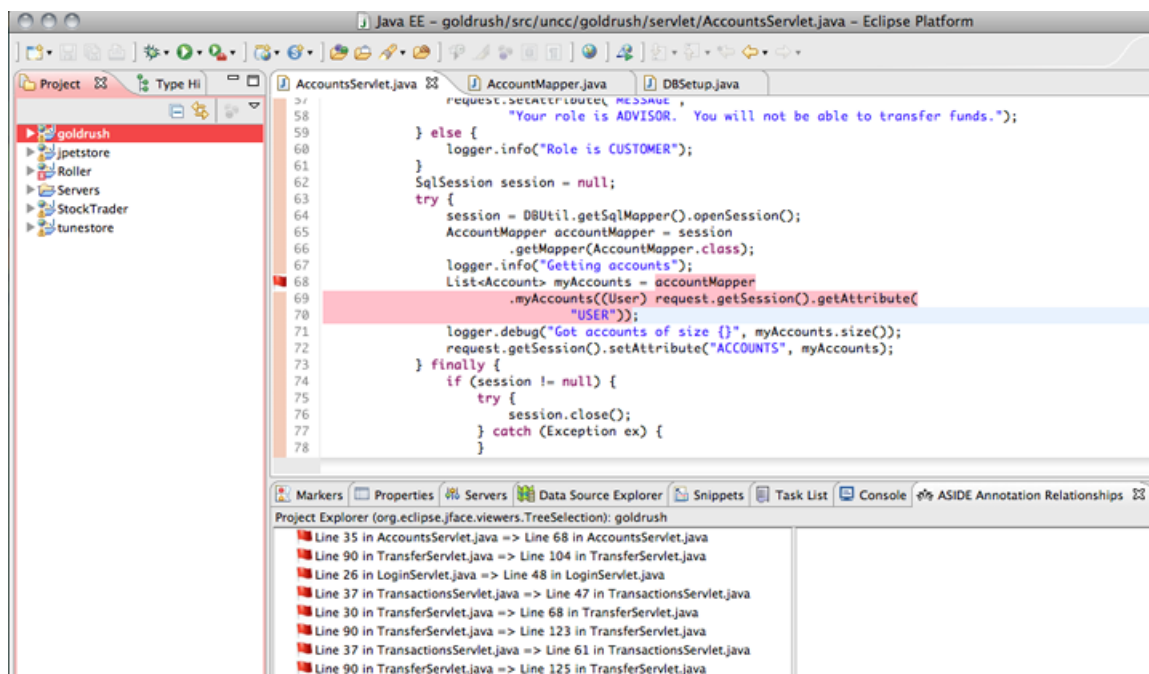


Figure 24: CodeAnnotate attaches a warning icon to the statement within a transaction that can be traced to an invocation of sensitive data operation within the application. In this case, a query to a database table Account.

stance, the `doGet` and/or `doPost` method in a Java Servlet through function calls. These program entry points are also predefined as rules in the format of XML. If a path can be established between a program entry point and a sensitive operation through a call graph of the application, *CodeAnnotate* reports a finding by attaching a warning to the statement that can be traced to the sensitive operation within the program entry point. In addition to the warning which is a red *flag* icon shown in Figure 24, *CodeAnnotate* also highlights the statement in pink.

To further examine the warning, the user can either click on the red flag warning icon or hover over the highlighted code. Both will bring up a prompt with a list of options, among which the top three are from *CodeAnnotate*. As shown in Figure 25, they are a) 1. Click me to read more about this warning; b) 2. Click me to annotate

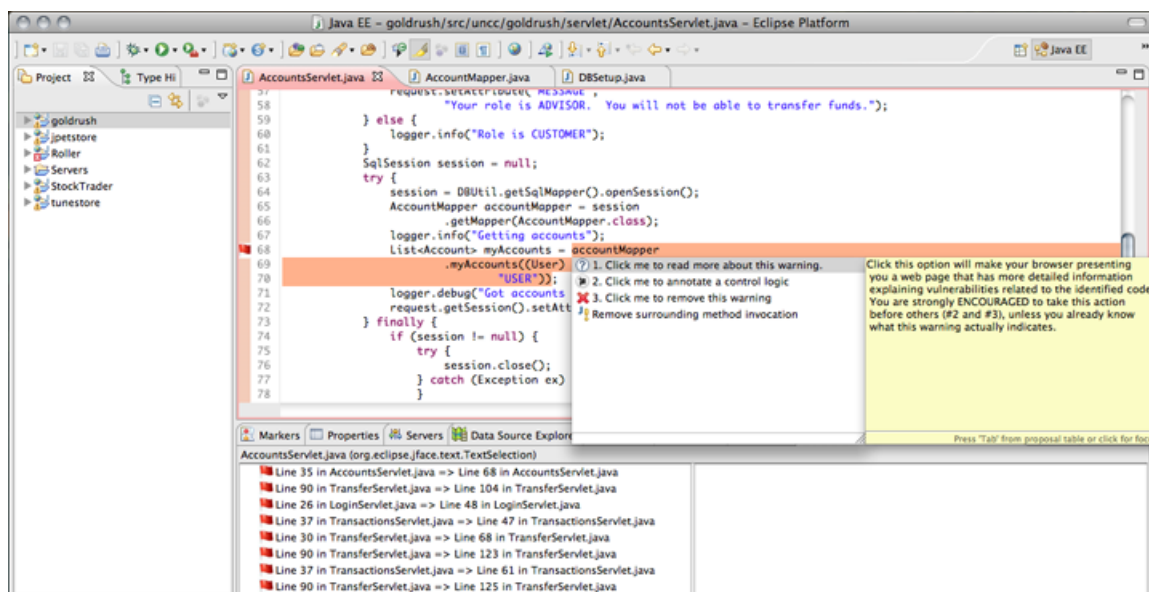


Figure 25: CodeAnnotate offers 3 options from which a developer can select.

a control logic; c) 3. Click me to remove this warning.

The selection of the first option “Click me to read more about this warning” brings up a page of detailed content explaining the warning in either a native Eclipse view or an external web browser. This page starts with explaining the high-level concept of vulnerability which is related to application access control. It then gives a scenario within which if certain access control is not performed, a vulnerability will be present which exposes the application to a certain attack, as shown in Figure 26.

The second option “Click me to annotate a control logic” is used by the developer to annotate code that checks access permission before executing the highlighted statement which will eventually perform certain sensitive operations. The selection of this option brings up a pink information box with textual information, shown in Figure 27, urging the developer to continue the process of locating and annotating code as well as instructing her how to do so. The information box is dismissed once

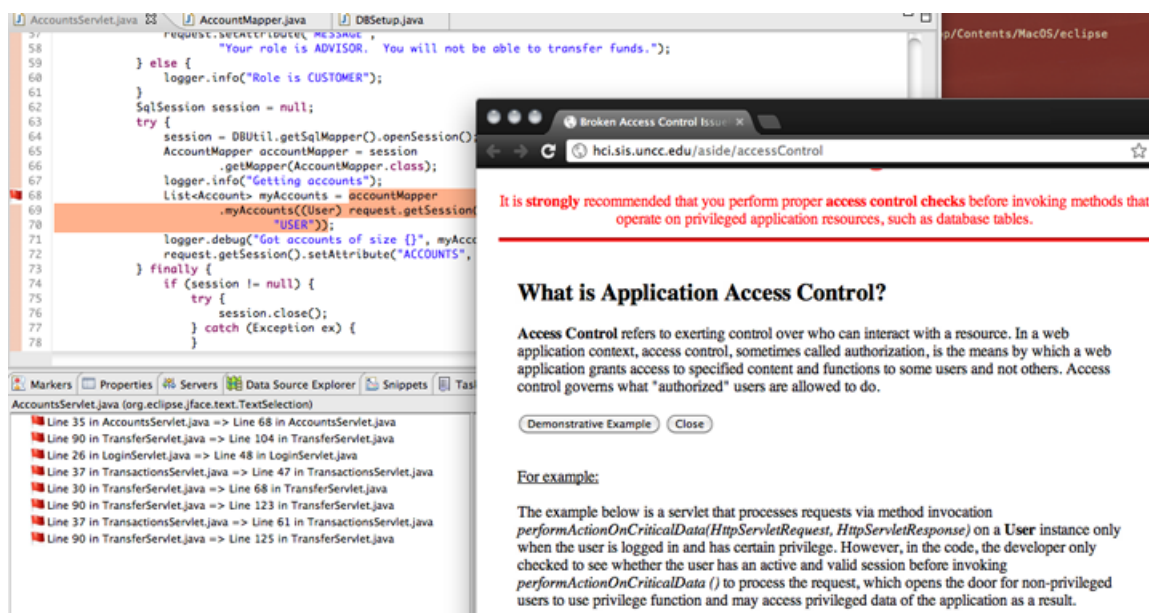


Figure 26: *CodeAnnotate* presents a developer a page of detailed content explaining the warning in either a native view or a web page when the developer chooses the *I want to read more* option.

an annotation is created for the selected highlighted statement.

Upon locating the appropriate control logic code, the developer can press keys CTRL + 0 to attach an annotation of a green *shield* with the selected piece of code. Figure 28 presents all the UI elements. This process also highlights the annotated control logic code with green. In the meantime, it turned the original pink highlighted code to yellow along with its attached warning icon to a yellow version. In some cases, multiple checks are needed for an operation to be performed. *CodeAnnotate* allows the developer to create and add multiple annotations to the yellow-highlighted statement.

To give developers a better understanding of the relationship of pieces of code, *CodeAnnotate* provides a Relationship view which visualizes the relationship in a tree structure. In the tree, a parent node is the path from a program entry point to a sensitive operation in the application. A child node represents the statement

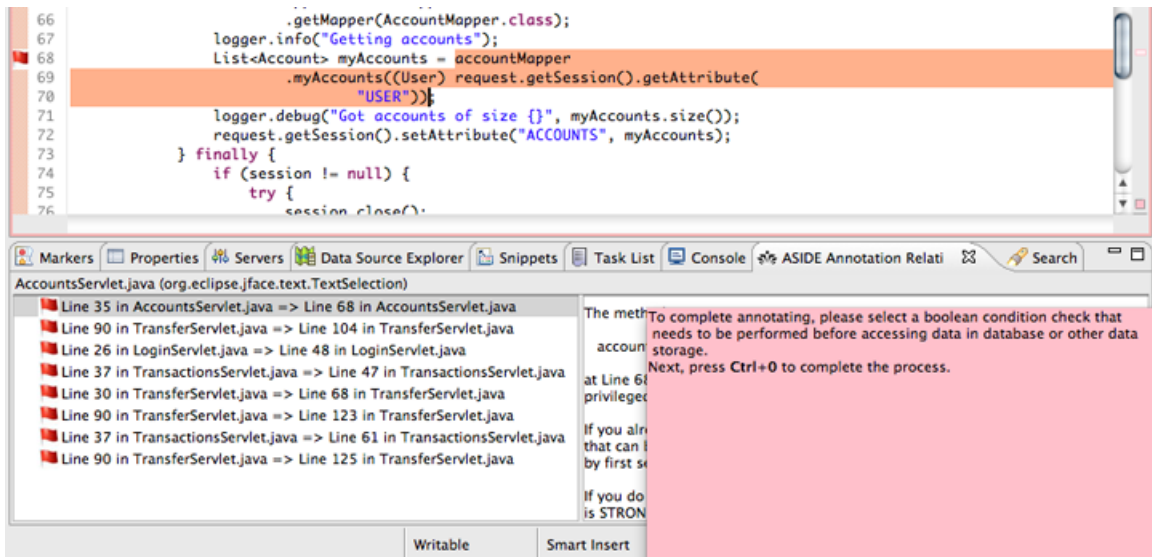


Figure 27: Upon selecting “Click me to annotate a control logic”, the developer was presented an information box at the bottom right corner as a reminder of completing the annotation process.

containing an annotated control logic check for the sensitive operation. A parent node can have more than one child when multiple annotations for multiple logic control tests are associated with a path. For example, Figure 28 shows a typical view in Eclipse which includes two groups of tabs. The upper group of tabs are opened files for editing, and the lower group of tabs are different Eclipse views displaying information of the application other than source code. The view being selected is the prototype implementation of the Relationship view, where a parent node is preceded by a red *flag* and a child node is preceded by a green *shield*. When a node is selected, its corresponding brief description is shown on the right text area.

In addition to adding a piece of control of logic on a path from an entry point to a sensitive operation, *CodeAnnotate* also gives developers the option to remove the control logic code from the path in cases where the developer considers it as a mistake. In order to detach the annotation from the highlighted code, the developer

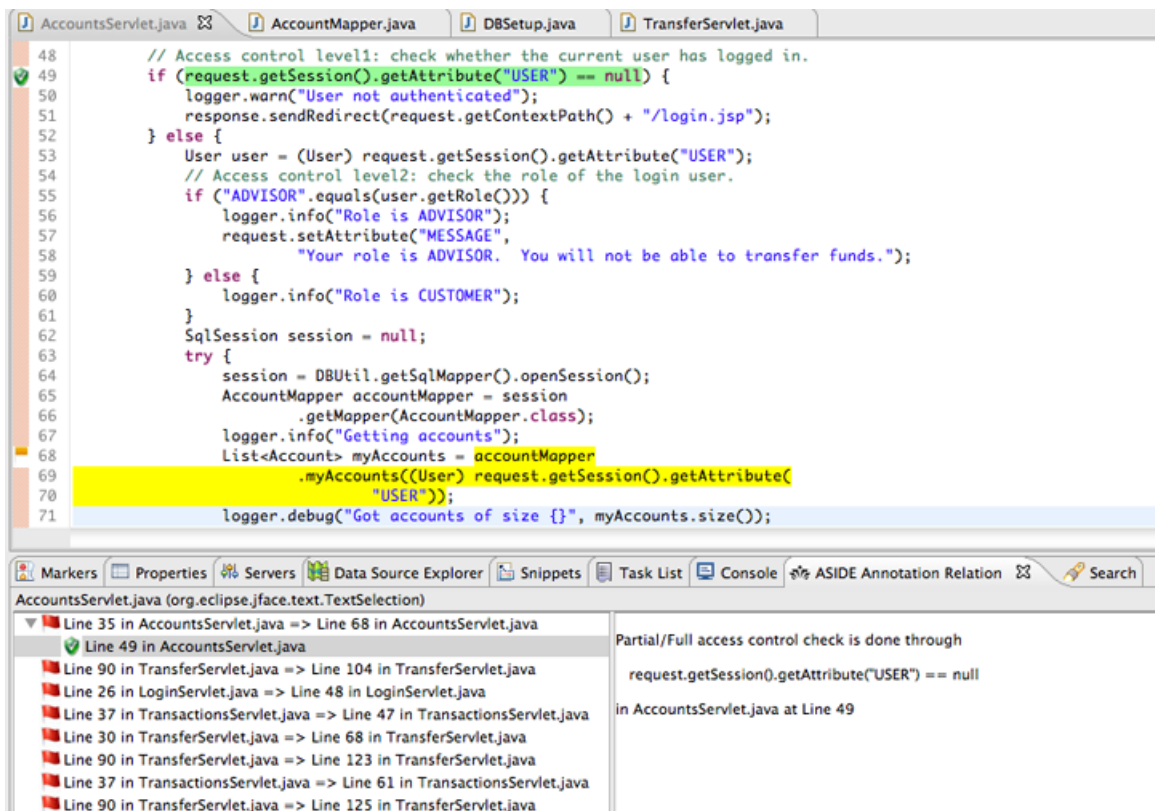


Figure 28: *CodeAnnotate* UI elements. Two different annotations (green and yellow) in the upper editor view. Tree structure in the bottom view visualizes the relationship between the annotations.

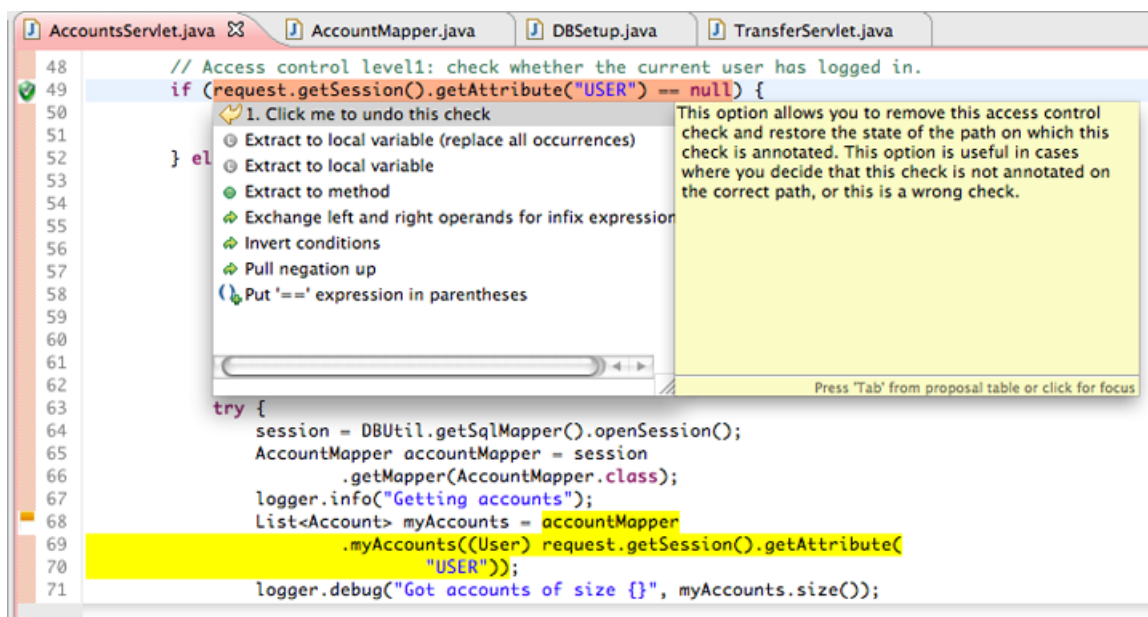


Figure 29: *CodeAnnotate* presents a developer an option to undo an annotation on a piece of control logic code.

simply needs to click on the warning and select the option on the top of the list to undo the check, shown in Figure 29.

The last option “Click me to remove the warning” provides the developer the ability to dismiss the warning in cases where the warning is a false positive.

6.5 Performance Measurement

To evaluate how effective the *interactive code annotation* technique is in detecting target vulnerabilities, I ran *CodeAnnotate* on two internally developed web applications that are immediately available: *Tunestore* and *Goldrush*. They were developed for web development and training for secure programming in my college.

6.5.1 Case I: Tunestore

Tunestore is a Java web application which provides its users the basic functionality of an online music store. A user can view all available songs in the store with or

without logging on to the application as a registered user. However, in order to purchase songs, a user has to register and log on as a registered user of the application. Once the user is logged in, s/he can comment on each song, make the purchase of a song, transfer money from other external banking account to the associated account and make changes to his/her personal profile. It also provides certain social networking functionality such as adding a friend and gifting a song to a friend.

The design of *Tunestore* follows the typical 4-tier architecture of a J2EE web application where an application presentation layer, control layer, business logic layer and persistence layer are logically separate processes. It implements the Model-View-Controller (MVC) design by utilizing the framework Apache Struts I. As a result, each use case of the application is implemented by an `org.apache.struts.action.Action` class. For instance, the login functionality is controlled by code in `LoginAction.java`. Its data management for the persistence layer relies on the relational database management system Apache Derby. Therefore, the operations on database are performed through Java SQL APIs such as `java.sql.Statement.executeUpdate(String sql)` to insert, update or delete a database table entry.

The measurement of performance on *Tunestore* is conducted from two aspects: Section 6.5.1.1 describes in detail the findings by running *CodeAnnotate* on *Tunestore*, and it especially focuses on analyzing and evaluating whether a warning is a false positive or a true positive. Section 6.5.1.2 focuses on an assessment of the complexity of the process of annotating the control logic code for a true positive as well as dismissing a false positive.

6.5.1.1 False positive analysis

As I described in Section 6.5.1, all the data manipulations of the application are done through Java SQL APIs. Thus, I mapped all those functions into *CodeAnnotate* sensitive information accessor specification. Based on the specification, *CodeAnnotate* knows whether an application accesses a database table and where this access occurs in the application code. Given this information, *CodeAnnotate* identified 29 cases where an entry point is able to reach a sensitive data operation through program function calls. As shown in Figure 30, the entry point is `ActionForward.execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` from `LoginAction.java`. The reason that this function is defined as an entry point is because it overrides the `execute()` method of Struts I `Action` abstract class. The path ends at a statement that invokes `Statement.executeQuery(String sql)`, a function that retrieves data from a database table, within `LoginAction.java`.

A *false positive* for *CodeAnnotate* is identifying an invocation of a data manipulation function that does not require any type of access control logic check within the context of *Tunestore*. To analyze false positives that were generated by this process, I manually inspected all the findings reported by *CodeAnnotate*. Out of all 29 identified issues, ten are easily classified as *false positives*. The effort involved to locate and annotate the corresponding access control checks for the 19 true positives is presented in following section on the complexity of annotating control access code. In this section, I focus on analyzing the *false positives*.

```

28     private static Log log = LogFactory.getLog(LoginAction.class);
29
30     public void setDataSource(DataSource dataSource) {
31         this.dataSource = dataSource;
32     }
33
34     public ActionForward execute(ActionMapping mapping, ActionForm form,
35         HttpServletRequest request, HttpServletResponse response)
36         throws Exception {
37         DynaActionForm df = (DynaActionForm) form;
38         String login = (String) df.get("username");
39         String password = (String) df.get("password");
40         Boolean stayLogged = df.get("stayLogged") == null ? new Boolean(false)
41             : (Boolean) df.get("stayLogged");
42         ActionMessages errors = getErrors(request);
43         ActionMessages messages = getMessages(request);
44
45         Connection conn = null;
46         try {
47             conn = dataSource.getConnection();
48             String sql = "SELECT USERNAME, PASSWORD, BALANCE FROM TUNEUSER"
49                 + " WHERE TUNEUSER.USERNAME = '" + login
50                 + "' AND PASSWORD = '" + password + "'";
51
52
53             Statement stmt = conn.createStatement();
54             stmt.setMaxRows(1);
55             ResultSet rs = stmt.executeQuery(sql);
56             if (rs.next()) {
57                 messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
58                     "login.successful"));
59             }
60         } catch (SQLException e) {
61             log.error(e);
62             messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
63                 "login.failed"));
64         } finally {
65             if (conn != null) {
66                 conn.close();
67             }
68         }
69         return mapping.findForward();
70     }

```

Figure 30: *Tunestore* Login Action Servlet

There are 10 cases, shown in Figure 31, where the reported warnings represent data access or manipulation operations that do not require access control. One type of false positives which includes 5 cases, highlighted in dark green (row 7-11) in Figure 31, is where the data that is being accessed is of non-sensitive nature. For example, all the song details and the comments of a song. Conceptually, these false positives can be easily prevented with a feature that allows a SSG to mark tables as sensitive and non-sensitive before this information is provided to *CodeAnnotate*. However, this is not implemented in current *CodeAnnotate* prototype. Another 5 cases (row 2-6) were involved with use cases such as login, logout and registration, as highlighted in orange in Figure 31. The data manipulations happen in these use cases do not require any access control, since these are the starting point where different users will be granted different access rights towards data in the application.

1	User Case	Entry Point	Path	Reason
			Sensitive Operation	
2	Login	LoginAction.java:34:execute(...)	LoginAction.java:55:stmt.executeQuery(...)	Application logic does not require access control
3		LoginAction.java:34:execute(...)	LoginAction.java:77:stmt.executeQueryUpdate(...)	
4	Logout	LogoutAction.java:29:execute(...)	LogoutAction.java:29:stmt.update(...)	
5	Registration	RegisterAction.java:29:execute(...)	RegisterAction.java:56:executeQuery(...)	
6		RegisterAction.java:29:execute(...)	RegisterAction.java:65:executeUpdate(...)	
7	View Songs	ViewCDsAction.java:20:execute(...)	ViewCDsAction.java:110:executeQuery(...)	CDs table is of no sensitive nature
8	Purchase a Song	BuyAction.java:32:execute(...)	BuyAction.java:59:stmt.executeQuery(...)	CDs table is of no sensitive nature
9	View Comments	CommentsAction.java:35:execute(...)	CommentsAction.java:47:stmt.executeQuery(...)	Comments table is of no sensitive nature
10		CommentsAction.java:35:execute(...)	CommentsAction.java:63:stmt.executeQuery(...)	
11	Gift Setup	GiftSetupAction.java:34:execute(...)	GiftSetupAction.java:69:stmt.executeQuery(...)	CDs table is of no sensitive nature

Figure 31: 10 false positive cases of *Timestore*

6.5.1.2 Complexity of annotating control access code

CodeAnnotate identified 19 paths that need access control checks within the context of *Tunestore*. Some of them already have the appropriate checks implemented in code while the rest do not. *CodeAnnotate*'s effectiveness in preventing broken access control vulnerabilities during the programming process is not only dependent upon the accuracy in detecting paths that need access control checks, but also relies on a developer's success in locating and annotating proper checks for identified paths.

To describe the difficulty involved in locating and annotating a control logic check for an identified path, I use the measurement of the *distance* between the location of the control logic check code and the entry point in terms of *within a method*, *within a class*, and *within a package*. When a control logic check is enclosed within the entry point method, the distance is within a method, which I assign a value 1. When an entry point does not contain the control logic check, but shares the same enclosing class, the distance is within a class, which is assigned a value 2. When the control logic check is out of the scope of the enclosing class of the entry point, the distance is within a package with a value 3. In cases where a path has several control checks, the corresponding complexity is represented by the accumulative value of all checks. Figure 32 shows the result of applying the distance metric to the 19 true positives.

User Case	Path		Control Check(s) Distance	Required Check(s)
	Entry Point	Sensitive Operation		
Change Password	PasswordAction.java:31:execute(...)	PasswordAction.java:58:stmt.executeQuery(...)	2	WHERE clause CSRF check
Add Balance	AddBalanceAction.java:56:execute(...)	AddBalanceAction.java:93:stmt.executeUpdate(...)	2	WHERE clause CSRF check
Purchase a Song	BuyAction.java:32:execute(...)	BuyAction.java:73:stmt.executeQuery(...)	1	WHERE clause
	BuyAction.java:32:execute(...)	BuyAction.java:90:stmt.executeQuery(...)	1	WHERE clause
	BuyAction.java:32:execute(...)	BuyAction.java:95:stmt.executeQuery(...)	2	WHERE clause CSRF check
Comment a Song	LeaveCommentAction.java:30:execute(...)	LeaveCommentAction.java:46:executeUpdate(...)	1	CSRF check
Add a Friend	AddFriendAction.java:32:execute(...)	AddFriendAction.java:45:stmt.executeQuery(...)	2	WHERE clause CSRF check
	AddFriendAction.java:32:execute(...)	AddFriendAction.java:57:stmt.executeQuery(...)	1	Previous SQL execution
	AddFriendAction.java:32:execute(...)	AddFriendAction.java:77:stmt.executeQuery(...)	1	Previous SQL execution
View Friends	FriendsAction.java:33:execute(...)	FriendsAction.java:45:stmt.executeQuery(...)	1	WHERE clause
	FriendsAction.java:33:execute(...)	FriendsAction.java:65:stmt.executeQuery(...)	1	WHERE clause
Gift Setup	GiftSetupAction.java:34:execute(...)	GiftSetupAction.java:60:stmt.executeQuery(...)	1	WHERE clause
	GiveAction.java:32:execute(...)	GiveAction.java:56:stmt.executeQuery(...)	2	WHERE clause CSRF check
Gift a Friend	GiveAction.java:32:execute(...)	GiveAction.java:74:stmt.executeQuery(...)	1	Previous SQL execution
	GiveAction.java:32:execute(...)	GiveAction.java:88:stmt.executeUpdate(...)	1	Previous SQL execution
	GiveAction.java:32:execute(...)	GiveAction.java:98:stmt.executeUpdate(...)	1	Previous SQL execution
	GiveAction.java:32:execute(...)	GiveAction.java:107:stmt.executeUpdate(...)	1	Previous SQL execution
Display Account Balance	LeftAction.java:25:execute(...)	LeftAction.java:34:stmt.executeQuery(...)	1	WHERE clause
View Songs for User	ListAction.java:17:execute(...)	ListAction.java:21:DBUtil.getCDSForUser(...)	3	WHERE clause

Figure 32: Complexity of annotating for *Timestore*

According to *Tunestore* application logic, ten out of the 19 cases require a user to login and maintain an active session. *Tunestore* enforces this type of access control check throughout the application by conditioning all the SQL statements with a WHERE clause to see whether the user from the current session is null or not. E.g. `WHERE TUNEUSR = "request.getSession(true).getAttribute("USERNAME")"`. Within the context of *Tunestore*, all these checks are within a method distance to their entry points, thus, it is perceived to be easy to locate and annotate. In addition to checking whether the current session is active, six cases need CSRF check, shown in the last column of Figure 32. *Tunestore* does not provide any defense against CSRF, thus the developer would need to write the corresponding defense code in these 6 cases, she then may annotate the checks for corresponding paths. A slightly more complicated case is gifting a song to a friend where 5 different types of data manipulations are involved. Except the first data manipulation statement, each succeeding manipulation depends on the success of the preceding accessing. Despite that these checks are not as direct as a Boolean test condition which is straightforward to be identified, they are within the same method with the warned statement.

6.5.1.3 Discussion

With pre-defined program entry points and sensitive operations that change program data, *CodeAnnotate* is capable of identifying all paths that require access controls with a false positive rate of 34%(10/29) in the above case study. Further improvements such as allowing SSG to set control metadata for sensitive data tables

can remove the 5 cases that are related to reading from non-sensitive tables: *CDs* and *Comments*. This can further lower the rate to 20.8%(5/24). It is a low false positive rate considering that *Tunestore* is a small application with only 16 use cases including login, logout and registration.

For *Tunestore*, it is not a complex process to locate the control logic checks since most checks are within the same class as the entry points. These checks, however, are not of a direct and uniform format such as a `Boolean` conditional test. There are other more implicit logic controls such as `WHERE` clause for a SQL query, a preceding SQL execution, a segment of code that performs certain logic, which complicate the annotating of control checks. *CodeAnnotate* has implemented the most intuitive checks, which are conditional tests that are often implemented as `boolean` test. Implementing annotating the aforementioned implicit logic checks and exploring more variants can be interesting future work.

6.5.2 Case II: Goldrush

Goldrush is another in-house built web application based on J2EE platform, which simulates a banking application with basic functionality such as login, logout, display account information, transfer funds from one account to another. The application implements a role-based access control mechanism which assigns different access rights to 2 major roles involved: customer and financial advisor. More specifically, a customer can only view detail information of an account that belongs to her, she can only make a transfer from an account that belongs to her to another account. A customer can have only one financial advisor. On the other hand, an advisor can be

advising multiple customers. She is allowed to see the accounts information of her client. She, however, is not allowed to make a transfer on the behalf of her client.

Goldrush is also a 4-tier based web application. The presentation layer is implemented using JSP; the control layer is based on Java Servlet; All data is stored in a database managed by a lightweight Database Management System (DBMS) HSQLDB. However, instead of embedding SQL within Java to interact with database, *Goldrush* employs Apache iBatis which is a data mapper framework that couples objects with stored procedure or SQL statement using a XML descriptor or annotations. For instance, the `loginUser(String username, String password)`, shown in Figure 33, will return the SQL execution result of the statement `SELECT username, password, role, surname, givenName FROM user WHERE username = #{username} AND password = #{password}` because of the mapping shown in Figure 33 in iBatis mapping file. Therefore, *CodeAnnotate*'s default sensitive information accessor rules are not sufficient to cover the patterns involved in *Goldrush*, since *Goldrush* does not have native SQL execution statement function calls. Thus, I expanded the default accessor rule specification to cover the functions that are mapped to SQL statements. For the previous example, the function call is `loginUser(User user)` with a return type of `User`.

6.5.2.1 False positive analysis

Table 5 shows all the paths found in *Goldrush* by *CodeAnnotate*. My examination confirmed that 3 of them, shown in Figure 34, are false positives, while the other 5 are true positives. Out of the 3 false positives, one (row 2) takes place in the login

```
<select id="loginUser"
  parameterType="uncc.goldrush.bean.User"
  resultType="uncc.goldrush.bean.User">
  <![CDATA[
    SELECT username, password, role, surname, givenName
    FROM user WHERE username = #{username}
    AND password = #{password}
  ]]>
</select>
```

Figure 33: iBatis implementation for `loginUser()` method

use case when a user is being granted access to the application. The other two (row 5 and row 9) are logically dependent on the access control checks for their preceding sensitive operation. For instance, row 5 retrieves all the transactions performed on a given account for a given user. This is legitimate and safe if the account given belongs to the user who requested the information. This check, however, should be performed before the detailed account information is retrieved, which takes place at row 4. Since the warning for retrieving account detail at row 4 is a true positive, the warning at row 5 is a false positive. For the same token, row 9 which inserts a transaction of a given source account is dependent upon row 8, where updating the source account happens, shown in Figure 35.. To update the source account, two access checks are needed to ensure that the user has right to view and modify the account. Therefore, it has been reported as a true finding.

Table 5: The 8 paths in *Goldrush* found by *CodeAnnotate*

<i>Use Case</i>	<i>Enclosing Class (.java)</i>	<i>Entry Point Function</i>	<i>Sensitive Access Point</i>	<i>Enclosing Class (.java)</i>
Login	LoginServlet	Line 30: doPost(...)	Line 48: mapper.loginUser(...)	LoginServlet
Display Accounts Information	AccountsServlet	Line 40: doGet(...)	Line 68: mapper.myAccounts(...)	AccountsServlet
Display Transactions	TransactionsServlet	Line 41: doPost(...)	Line 47: mapper.getAccount(...)	TransactionsServlet
	TransactionsServlet	Line 41: doPost(...)	Line 61: mapper.getTransactionForAccount(...)	TransactionsServlet
Transfer Money	TransferServlet	Line 34: doGet(...)	Line 68: mapper.getAccount(...)	TransferServlet
	TransferServlet	Line 94: doPost(...)	Line 58: mapper.myAccounts()	TransferServlet
	TransferServlet	Line 94: doPost(...)	Line 123: mapper.updateAccount(...)	TransferServlet
	TransferServlet	Line 94: doPost(...)	Line 125: mapper.insertTransaction(...)	TransferServlet

User Case	Entry Point	Path	Sensitive Operation	Reason
Login	LoginServlet.java:30:doPost(...)		LoginServlet.java:48:mapper .loginUser(...)	
Display Transactions	TransactionsServlet.java:41:doPost(...)		TransactionsServlet.java:61: mapper.getTransactionForAc count(...)	Application logic does not require access control
Transfer Money	TransferServlet.java:94:doPost(...)		TransferServlet.java:125:ma pper.insertTransaction(...)	

Figure 34: 3 false positives from *Goldrush*

6.5.2.2 Complexity of annotating control access code

To measure the complexity of annotating the control logic checks for the 5 identified true positives, I applied to *Goldrush* the same methodology described in analyzing the complexity of annotating control access code for *Tunestore*. Figure 35 presents the results. *Goldrush* has existing access control checks for some of the functionality but is lacking for other. Therefore, for identified paths, the developer would need to come up within her own check logic for each path. Since the existing checks were done with the same methods of the warned statements, new checks could also be added in within the same methods.

User Case	Path		Control Check(s) Distance	Required Check(s)
	Entry Point	Sensitive Operation		
Display Accounts Information	AccountsServlet.java:40:doGet(...)	AccountsServlet.java:68:mapper.myAccounts(...)	1	user from session
Display Transactions	TransactionsServlet.java:41:doPost(...)	TransactionsServlet.java:47:mapper.getAccount(...)	2	user from session user has account
Transfer Money	TransferServlet.java:34:doGet(...)	TransferServlet.java:58:mapper.getAccount(...)	2	user from session user has account
Transfer Money	TransferServlet.java:94:doPost(...)	TransferServlet.java:68:mapper.myAccounts(...)	1	user from session
Transfer Money	TransferServlet.java:94:doPost(...)	TransferServlet.java:123:mapper.updateAccount(...)	2	user has account user can make transfer

Figure 35: Complexity of annotating for *Goldrush*

6.5.2.3 Discussion

The result generated by *CodeAnnotate* on *Goldrush* is similar to *Tunestore*. It is able to identify all broken access control issues with a similar false positive rate of 37.5%(3/8). Since login and registration account for 40% of the total 5 use cases, this is a low false positive rate. One of the false positives is related to the login function of the application, which is the starting point of interactions with the application. It is also clear that, for *Goldrush*, locating the control logic checks is not a complex process since most checks are within the same class as the entry points. It, however, shares the same difficulty that some control checks are not as intuitive as a `Boolean` condition test, as described in the *Tunestore* case.

The current prototype implementation of *CodeAnnotate* seems to be effective in identifying broken access control issues with a relatively low false positive rate. Such a result, however, applies only to small web applications with very limited functionality and simple implementations. It is not clear at this point whether it will be as effective when used on more complex web applications that involve more functionality and more advanced technologies.

6.6 User Study

Although my prototype implementation is limited in scope, it has demonstrated the feasibility and potential of the theory, which is interactive code annotation for detecting insufficient access control issues. As described in Section 6.4.1, *CodeAnnotate* involves more complicated interactions with its end users. Therefore, a good user interface and interaction design is perceived to increase *CodeAnnotate*'s effective-

ness. To evaluate the current design as well as to acquire more design requirements, I conducted user studies on programming tasks using *CodeAnnotate*.

The immediate goals of the studies are as follows:

- Can participants always annotate the correct control logic for a sensitive information access?
- If participants were able to annotate correct control logic, how long, on average, did it take for a participant to establish a relationship between access control logic code and sensitive information accessing code?
- If participants were NOT able to annotate correct control logic, what were the major difficulties/obstacles preventing participants from succeeding?
- If there was any, what were the good design/implementation aspects that facilitated participants to interact with *CodeAnnotate*?

6.6.1 Study I - Controlled Lab Study

For this study, I, along with my colleagues, recruited in total 20 students from a graduate level Java-based web application development course offered by our college in Spring 2012 semester. This time, the course used for evaluating *CodeGen* and *Explanation* does not cover any secure programming practices in terms of building web applications.

Part of the students course work was to build an online stock trading system incrementally over four projects throughout the semester using Java Servlet technology. My study focused on the last increment of this project, where students were asked to

implement functionality including display stock details, make a buy/sell stock 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 functionality.

This was a controlled study. Student participants were asked to come to our lab and work on their assignment for 3 hours. I installed two machines with similar hardware and software configurations. They had Eclipse installed with both ASIDE *CodeRefactoring* and *CodeAnnotate*. Each study session took up to 3 hours of development and up to 20 minutes of debriefing ensued.

In addition to investigating whether *CodeAnnotate* is effective in preventing developers from writing code that has broken access control issues, this study also intended to assess whether my design and implementation of ASIDE on *CodeRefactoring* from Chapter 5 has an impact on improving novice programmers' secure programming knowledge and awareness. Therefore, upon a participant's arrival at our study lab, s/he was first instructed to fill out a pre-survey which is an essential part of assessing the impact of *CodeRefactoring*. Then, one of my co-investigators or me gave him/her a brief walkthrough of how the two software tools work, respectively. In the meantime, my other co-investigator helped set up the development environment and screen recording software on the machine. He also imported the participant's existing project into Eclipse, and made sure it compiled and could run on the local web server installed in Eclipse. This process prevented participants from wasting time or being frustrated on configuration of an unfamiliar IDE - Eclipse, since most participants were only familiar with Netbeans. The participant was then asked to

launch the tools before proceeding to development. Students worked as they wished on the assignment. They were told, however, that they have to examine at least one of the warnings show up throughout the development. When 3 hours was up, we asked the participants to stop and instructed them to fill out a post-survey, another essential part of evaluating impact of *CodeRefactoring*. Afterwards, we brought our participants to our interview room for debriefing.

Each interview lasted from 15 to 20 minutes. We began by asking participants to describe the types of warnings they encountered throughout the development session. In the cases where participants encountered *CodeAnnotate* warnings, we made extra effort on discussing their interactions with those warnings in terms of the difficulties they ran into. The discussion was based on the 3 options provided by *CodeAnnotate* to address a warning: read more about this warning; annotate a control logic; remove this warning. For each option, we asked for the difficulties they had during the interactions. Then, we switched our discussion to focus on general questions, such as whether s/he gained any knowledge from interacting with the tools.

The study was not able to give answers to the intended research questions, however it did provide interesting insights for designing tool evaluation studies.

First off, a study evaluating a tool should avoid involving software tools that offer similar interactions to users. In this study, I aimed to evaluate two software tools: *CodeRefactoring* and *CodeAnnotate*, which offer similar user interface elements, such as warnings, and list of solutions. It was too much a burden presenting too much information all at once to participants/students who were new to the development environment. On average, each participant encountered more than 10 warnings raised

by *CodeRefactoring* of 3 types. For instance, the warning could be related to an input validation issue, an output encoding issue, or a dynamic SQL issue.

Further complicating the case, each type of warning would involve different interactions. For example, an input validation warning gives a user 17 options. These options in turn can produce 3 effects. For example, one provides a user a web page full of textual content; another generates code for the user; and the third one removes the warning from view. An output encoding warning has similar formats for presenting information. The dynamic SQL warning, however, demands that the user to change his/her code manually in order to remove the warning from view.

The integration of *CodeAnnotate* requires users to juggle more issues during the study. *CodeAnnotate* brought in one different warning icon, two information icons, a relationship view and an information box. As a design option, I chose a red flag icon for *CodeAnnotate* warnings which resembles *CodeRefactoring* warning icon. This led to the belief that those warnings were of the same type. On average, each participant encountered more than 10 warnings of various kinds. Thus, most participants failed to examine *CodeAnnotate*.

Another interference from *CodeRefactoring* is caused by the warnings on statements that execute SQL queries. By design, one of *CodeRefactorings* detections is about dynamic SQL execution which is a common vulnerable code pattern for SQL Injection vulnerabilities. A warning shows up whenever there is a method invocation of one of the Java SQL execution functions, e.g. `java.sql.Statement.execute(String sql)` with a red devil head icon attached on the left ruler of the java editor. *CodeAnnotate* reports warnings on the same statements where these SQL execution method

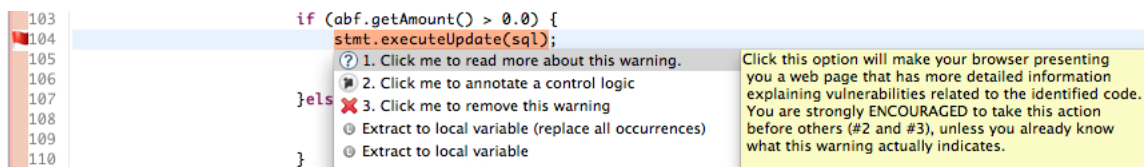


Figure 36: *CodeAnnotate* warns a sensitive information accessing without proper control check with a red flag icon.

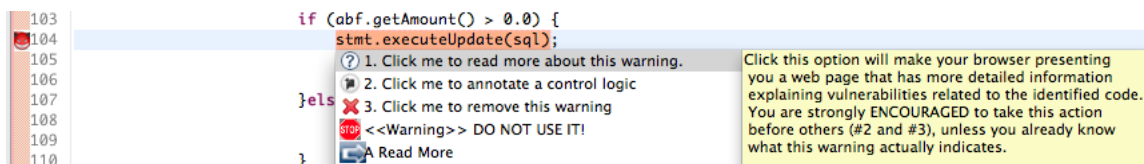


Figure 37: *CodeRefactoring*'s dynamic SQL warning overshadows *CodeAnnotate*'s warning on the same statement.

invocations are called with a red flag icon attached on the left ruler of the java editor, shown in Figure 36. However, when these two software tools are used together, a statement of interest would have two different warnings with only the *CodeRefactoring* warning visible to the developers, as shown in Figure 37.

As mentioned above, students were introduced to use Netbeans as the primary development tool throughout the course. For creating Java servlets, Netbeans generates template code which is common for any Java class that inherits `javax.servlet.HttpServlet`. More specifically, it creates a common method `void processRequest (HttpServletRequest request, HttpServletResponse response)` invoked by both entrance methods `void doGet (HttpServletRequest request, HttpServletResponse response)` and `void doPost (HttpServletRequest request, HttpServletResponse response)`, which renders the body for both entrance methods has only one statement. Since *CodeAnnotate* issues a warning at the transaction level, in this context, it is within the body of entrance methods. As a result, the first participant encountered cases shown in Figure 38. The warning

```

39 @Override
40 protected void doGet(HttpServletRequest request,
41                      HttpServletResponse response) throws ServletException, IOException {
42     processRequest(request, response);
43 }

```

Figure 38: *CodeAnnotate* issues a warning within an entrance method with a single statement.

in this case offers no extra context that is essential for a user to understand. As a result, the participant gave up any further interaction with the warning based on the rationale that the code was generated by Netbeans.

To increase the visibility of *CodeAnnotate* warnings as well as to reflect the relationship among entry point, information access point and access control checks, I provided a Relationship view that displays all existing *CodeAnnotate* warnings of a project. However, the view was aligned with the Eclipse Console view that displays the application execution output. The Console view is usually the primary focus of a developer during development. Therefore, the *CodeAnnotate* Relationship view was not visible at all times throughout the development session for 2 of the participants. As a result, they failed to be aware of the existence of *CodeAnnotate* warnings showing up in their code.

6.6.2 Study II - Observation + Think Aloud

To evaluate the current user interface design of *CodeAnnotate* as well as to acquire more requirements, I devised an exploratory study in which I observed each participant using *CodeAnnotate* on a functioning web application, which was either developed for assignment 4 used in previous study by the participant, or being introduced to the participant as a target for practicing secure programming against

common web application vulnerabilities. I asked all participants to think aloud while they were dealing with *CodeAnnotate* warnings in the applications. During the observation, I also interacted with participants when I found an action worth being probed further by asking them questions. I recorded the screens as well as my conversations with participants for data analysis.

6.6.2.1 Participants and Procedure

I recruited in total 8 volunteers from 3 different sources. Three of them were from the 20 students who participated in the previous study but failed to interact with *CodeAnnotate*. Two were from a secure web application course which aims to teach students ethical hacking and secure programming. It has been offered to both undergraduate and graduate students by my college every semester since year 2006. The other 3 students were from a special security program at the college where students are competitively selected to receive full cyber security scholarships. Participants recruited from different sources were at different experience levels of web development and secure programming. The 3 from the web development course were the least experienced in programming, they had no prior exposure to secure programming. The 2 from the secure programming course were relatively more experienced in programming, and were in the process of acquiring secure programming knowledge. The 3 from the special 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 *CodeAnnotate*.

For the 3 participants recruited from the web development course, they exercised *CodeAnnotate* on the project, online stock trading system, they were working on for the assignment. For the other 5 participants, however, I used *Tunestore* described in Section 6.5.1, which is a Java servlet based web application with a variety of known vulnerabilities, developed for the purpose of ethical hacking for students who take this course. Prior to the study, students already had knowledge about the functionality and corresponding source code since they were asked to find the vulnerabilities of *Tunestore* for one of their assignments and fixed them in another.

Upon a participant's arrival, I gave an introduction of *CodeAnnotate* on a sample web application by addressing a warning through the 3 different options provided by *CodeAnnotate*: read more about this warning; annotate a control logic; remove this warning. Then, I imported the participant's project or *Tunestore* into a new workspace of Eclipse, and started screen and voice recording. Next, I let the participant examine the warnings shown in his/her code. Each session lasted from 30 to 40 minutes.

6.6.2.2 Data Analysis

While I obviously had a small sample of participants, their performance, however, varied significantly. I extracted 3 characteristics that heavily influenced the performance patterns emerged across the interactions with *CodeAnnotate* among 8 participants with different backgrounds of application development and application security.

I sought out participants with different backgrounds with respect to building secure

applications. The 3 participants who were from the graduate level web application development course were only presented, for the first time, with the potential abuse of functionality in the previous study through *CodeRefactoring*. The other 5 either had been or were taking a course on penetration testing and attacking/breaking web applications by exploiting code level vulnerabilities, such as SQL Injection, Cross-site Scripting, Privilege Escalation, Cross-site Request Forgery, etc. They were very familiar with common web application vulnerabilities during the secure web application course.

All 3 participants without established attacker mentality, the mindset of manipulating regular functions to do unintended things, failed to comprehend the problem presented in explanatory material which illustrates a typical broken access control that can result in a vertical privilege escalation attack. Moreover, none of them was 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 had already read through. When confronted to explain what is the problem described in the page, however, she murmured and then switched the topic to that using `PreparedStatement` to prevent SQL Injection issues, which she learned from participating in the previous study.

Participant P3 was on his own exploring the functionality of *CodeAnnotate* for

dealing with *CodeAnnotate* warnings on his code for the first part of the study. He glanced quickly through the explanatory page explaining broken access control issues, 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 to the above 3 participants, the other 5 participants who possess an attacker mentality were able to understand the explanatory material and attempted to use the information to guide their examination of *CodeAnnotate* warnings generated in *Tunestore*. All 5 participants were able to make a quick and accurate decision on the false positives on both 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 I mentioned the possibility of broken access control.

Having an attacker mentality while building an application enables a developer to think beyond just legitimate use cases of a functionality, and to be aware of possible illegitimate abuse cases. However, a typical web application involves a variety of attack surfaces to which a developer needs to be aware. In many cases, a developer does not have the mental capacity to keep in mind every single type of attack possibility. For instance, a broken access control can be an authentication bypass, a privilege escalation, an access to protected information, or a Cross-site Request Forgery issue, etc. Therefore, having an attack mentality but not mental models of common attacks is not sufficient for a developer to realize and prevent common vulnerabilities.

Among all 5 participants who possess an attacker mentality, none of them im-

mediately spotted the Cross-site Request Forgery (CSRF) vulnerabilities that were indicated by *CodeAnnotate* warnings. P4 explained that the exemplar case, which is a role-based access control issue, given in the explanatory material is hard to be applicable to CSRF scenarios in the application:

[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]...

All other participants also pointed out that the explanatory material did not give an example of such type of broken access control scenario. Thus, they had difficulty to establish the mental model of such an attack under the context of *Tunestore*.

Participant P7 also gave his explanation to how he understood and viewed the material:

[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.

An improvement to incorporate as many examples of common vulnerabilities that were caused by lack of or insufficient access control checks in the explanatory material was called for by all 5 participants. Like Participant P8 said:

[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 one example of a type of check and an example of this could be Cross-site Request Forgery.

Although it is impossible to provide explanation or case scenarios for application specific broken access control issues, it might be helpful to provide as many exemplar cases for possible different vulnerabilities that are caused by lack of or insufficient access control checks.

Once a CSRF attack mental model was established in the context of *Tunestore*, four out of the 5 participants successfully annotated a corresponding access control check following the given instructions provided by the tool.

Three of the 8 participants either had experience writing professional code or were about to be a professional developer. They were able to perform such annotation successfully and correctly on the first trial, while the other 5 participants had trouble to successfully locate and annotate a Boolean control check for a warning even after given detailed description of the potential vulnerability for a warning. When asked about whether the phrase “Boolean control check” makes sense to them, they responded positively. For instance, F8 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 its 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 other 5 found it difficult to accurately explain the phrase. It is hard to see whether there is any correlation between the programming capability of a user with his/her perception and usage of *CodeAnnotate*, but it will be interesting to investigate more comprehensively on this aspect.

Overall, *CodeAnnotate* 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. Figure 39 gives a summary of the results from the study.

6.6.2.3 Study Discussion

Throughout the study, participants identified several User Interface issues that can be improved to help developers use *CodeAnnotate* more effectively. I summarize the possible improvements into the following items in response to participants experience with *CodeAnnotate*.

- Highlight all `Boolean` logic checks in the current active/visible editor view.
The current design requires users to look for and locate code. The study shows that users have difficulty locating as well as selecting the right code piece. For instance, it's an error prone process to select an expression that has parenthesis.
- Attach the information presentation box to the cursor and have the box move around along with the cursor. The current design has the information box sitting statically at the bottom right corner of the Eclipse window. It can be

		Attacker Mentality	
		Medium	Low
Programming Capability	High	Understood explanatory material easily; Acknowledged the usefulness of <i>CodeAnnotate</i> warnings; Identified broken access control issues; Used <i>CodeAnnotate</i> to annotate control checks successfully; Elaborated confusion and pointed out design requirements.	
	Medium	Understood explanatory material easily; Acknowledged the usefulness of <i>CodeAnnotate</i> warnings; Identified broken access control issues with help from experimenter;	
	Low		Could not understand provided explanatory material; Could not understand the purpose of <i>CodeAnnotate</i> warnings; Could not identify any broken access control issues that exist.

Figure 39: Summary of Think-aloud study

easily overlooked since it is far away from the code the user is focusing on at the moment of figuring out the control check code.

- Present a sample attack that exploits broken access control problems associated with an identified issue, in addition to what should be done to countermeasure, in the explanatory material.
- Visualize the relationship between path and control check graphically.
- Give more examples showing more varieties of broken access control vulnerabili-

ties. Participants were able to understand the given example, but were not able to transfer the knowledge to the application context at hand since the example problem was far from being relevant to the application issues.

- Change the visualization of a processed path in the relationship view to indicate which path has been dealt with. A possible solution will be to gray out the item.
- Replace the information box with a different control for displaying information.

The current information box gives a false sense that the user can type notes.

An alternative design would be a controlled comparison study in which the experiment group would work on the programming tasks with Eclipse equipped with *CodeAnnotate*, while the control group would work within the same development environment without *CodeAnnotate*. While it seems to serve the goal of obtaining an answer as to whether *CodeAnnotate* is helpful in helping developers writing code with fewer broken access control vulnerabilities, it is insufficient for investigating whether a tool like *CodeAnnotate* is more effective than traditional knowledge dissemination, which is teaching in class. This problem can be addressed by adding in another group of participants who would work without *CodeAnnotate*, but would be taught about similar knowledge by an expert before the study.

This study, however, has several challenges which make the study impractical during the period of time allotted. The foremost is how to control the equivalence of knowledge being taught by an expert and knowledge provided by *CodeAnnotate*. The second challenge lay in the complication of the vulnerabilities which could lead to the high risk of not being able to get any data from the group taught by an expert.

This is also backed by the study in Section 6.6.2, which shows that participants who have been extensively trained in secure programming might still not be able to catch broken access control issues. The third challenge is that with intended size of participants divided into more than two groups, the data generated would not be sufficient to drawing any conclusions.

CHAPTER 7: CONCLUSION AND FUTURE WORK

7.1 Restatement of Contributions

Software vulnerabilities originating from insecure code are one of the leading causes of security problems people face today. Unfortunately, many developers have not been adequately trained in writing secure programs that are resistant from attacks violating program confidentiality, integrity, and availability, a style of programming which I refer to as *secure programming*. Worse, even well-trained developers can still make programming errors, including security ones.

Much work on software security has focused on detecting software vulnerabilities through automated analysis techniques. While they are effective, they are neither sufficient nor optimal. For instance, 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. What is worse, these approaches exclude programmers from the

security loop, and therefore, do not prevent them from continuing to write insecure code.

I dedicated my dissertation to investigating an approach to helping software programmers write secure code by interactively providing secure programming support while they are writing code to implement software. This dissertation started from understanding why developers make security errors that put software systems at risk of malicious attacks. It then described the concept of interactive secure programming in the IDE, helping developers write code that has fewer common security vulnerabilities. It presented my two implementations of tool support in the form of Eclipse IDE plugins for tackling two different classes of code vulnerabilities: vulnerabilities that are caused by improper or insufficient input validation or output encoding and vulnerabilities originate from lack of or inadequate access control. For each, this dissertation gives comprehensive evaluation in terms of its effectiveness in finding targeted vulnerabilities and its usability from potential end users' point of view. The evaluations demonstrate that interactive secure programming support is needed and helpful to develop more secure software.

To summarize, in this dissertation, I provide an in-depth understanding of why software programmers make security errors during programming with support of empirical evidence. I also devise a new approach that reminds software programmers of potential insecure code and provides them secure programming support during program construction to help them write secure code, in order to eventually develop more secure software. I develop two techniques, *interactive code refactoring* and *interactive code annotation*, to assist programmers in producing code with less common code vul-

nerabilities. Moreover, I implement prototype software in the form of a plugin for the Eclipse platform and conducted an extensive open source projects study to evaluate the effectiveness of the proposed techniques in addressing common vulnerable code. Additionally, I conducted comprehensive user studies to evaluate the current design of the implemented prototype as well as gain insights on how developers perceive this new approach.

7.2 Future Work

Throughout the dissertation, I have pointed out areas where more work is needed to advance and improve the interactive secure programming support approach. In this section, I give an overview of these open areas that may become interesting and valuable future research directions.

7.2.1 Web Frameworks

The implementation of *Interactive Code Refactoring* in this dissertation targets a foundational technology, Java Servlet, for developing Java web applications. While it is still used widely among universities for teaching students to build Java web applications, it is considered an obsolete technology in industry. Mainstream technologies that are used, instead, are web frameworks that abstract common characteristics of applications and provide developers implementations of these common characteristics on top of Java servlet. Such web frameworks, including Apache Struts I, Struts II, Spring MVC, JavaServer Faces and Grails, hide some interactions among an application and its end users in configurations or conventions, thus adding a layer of complication in analyzing applications.

One of the prominent examples is a concept called auto-binding, which allows a web framework to bind web request parameters into an application's model properties, without explicitly invoking `javax.servlet.http.HttpServletRequest.getParameter(String)`. This increases the difficulty of identifying the entry points where untrusted inputs get into an application. To further complicate the implementation of a more sophisticated *CodeRefactoring*, different web frameworks have different implementations of functionality supporting common web application characteristics. In addition, each web framework adds in a variety of programming elements into an application. The current implementation solution is far from sufficient in supporting them.

7.2.2 Larger Scale Evaluation on CodeAnnotate

In section 6.6.2, I have studied a small sample of developers using *CodeAnnotate*. The study generated interesting results, such as whether a developer possesses an attacker mentality has influence on whether she can use *CodeAnnotate* effectively. It, however, is not able to pinpoint whether *CodeAnnotate* can help them reduce broken access control vulnerabilities in their code. Therefore, it is interesting and of great value to implement a more robust and sophisticated *CodeAnnotate* and conduct user studies of a larger scale that investigate how developers use the tool.

7.2.3 Integration with Static Analysis Tools

Another development that can significantly increase the value of the interactive secure programming support approach is to integrate with current static analysis tools to reduce secure code review cost. In reality, there is a communication gap

between application developers and security auditors. In many cases, developers do not understand what security auditors need and security auditors are not able to get from developers what they want. Therefore, it is highly desirable if auditing results from static analysis tools can be consumed by both *CodeRefactoring* and *CodeAnnotate* to reduce noisy warnings. In the same vein, developers may provide feedback about the application context that can drive customized security analysis.

Integrating IDE secure programming support with static analysis tools provides a channel for developers to communicate their application logic to security auditors, and for security auditors to convey their security knowledge to application developers.

7.2.4 Support Secure Programming Education in the IDE

This dissertation has taken the first step studying students using my prototype implementation of interactive secure programming approach, and shown that it is embraced by students and perceived as a useful tool in helping them write more secure code. It is natural to be curious about whether such interactive secure programming reminding and assistance will have any impact on novice developers' security awareness. Furthermore, this can be extended to investigate whether integrating educational support into an IDE overcomes some of these challenges and provides effective training throughout a student's education.

Therefore, an interesting extension of this dissertation may be to (a) develop and deploy a usable tool that can serve a wide range of students and courses; (b) improve student awareness and understanding of security vulnerabilities in software; (c) increase utilization of secure programming techniques in assignments; and (d) have

minimal impact on other course objectives and instructors. Through which, one can evaluate how students use the tool, the impact on assignments, students' vulnerability awareness, and the impact on the course instructors.

7.3 Closing Remarks

Developers writing insecure code that lead to vulnerabilities in software is a reality. In this dissertation, I examine a new approach that provides interactive programming support in the IDE to help developers write more secure code that fewer common vulnerabilities. In particular, I investigated two techniques that aim to address vulnerabilities in web applications that are caused by untrusted inputs get consumed without proper validation, and vulnerabilities that result from lacking of proper access control for sensitive application operations. The technical evaluations have demonstrated that the techniques are able to successfully find in large open source projects with low false positive rates. The user studies have shown that my design of the approach is highly appreciated by developers. In addition, the research in this dissertation opens a door for additional research, as presented in the future work section.

REFERENCES

- [1] Shanai Ardi, David Byers, Per Hakon Meland, Inger Anne Tondel, and Nahid 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] ARS Technica. Anonymous speaks: the inside story of the hb-gary hack, 2011. <http://arstechnica.com/tech-policy/news/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack.ars>.
- [3] Davide Balzarotti, Marco Cova, Vika Felmetzger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni 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] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security, CCS '08*, pages 75–88, New York, NY, USA, 2008. ACM.
- [5] Allan Bateson, Ralph A. Alexander, and Martin D. Murphy. Cognitive processing differences between novice and expert computer programmers. *Int. J. Man-Mach. Stud.*, 26:649–660, June 1987.
- [6] Bill Pugh. Findbugs, 2011. <http://findbugs.sourceforge.net/>.
- [7] 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.
- [8] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrisnan. Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13:14:1–14:39, March 2010.
- [9] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.
- [10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE Interna-*

tional Conference on Software Engineering - Volume 1, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

- [11] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In Proceedings of the 27th international conference on Human factors in computing systems, CHI '09, pages 1589–1598, New York, NY, USA, 2009. ACM.
- [12] CERT. CERT Secure Coding. www.cert.org/secure-coding, year = 2011.
- [13] CERT. Top 10 Secure Coding Practices, 2011. <https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices>.
- [14] Avik Chaudhuri and Jeffrey 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, New York, NY, USA, 2010. ACM.
- [15] Brian Chess and Gary McGraw. Static analysis for security. IEEE Security and Privacy, 2:76–79, November 2004.
- [16] Brian Chess and Jacob West. Secure programming with static analysis. Addison-Wesley Professional, first edition, 2007.
- [17] Cigital. Whitebox SecureAssist. <http://www.cigital.com/solutions/secureassist/>.
- [18] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: an approach for the anomaly-based detection of state violations in web applications. In Proceedings of the 10th international conference on Recent advances in intrusion detection, RAID'07, pages 63–86, Berlin, Heidelberg, 2007. Springer-Verlag.
- [19] Coverity Inc. Coverity static analysis, 2011. <http://www.coverity.com/products/static-analysis.html>.
- [20] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni 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, Berkeley, CA, USA, 2010. USENIX Association.
- [21] Xiang Fu and Kai 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, New York, NY, USA, 2008. ACM.
- [22] Mark G. Graff and Kenneth R. Van Wyk. *Secure Coding: Principles and Practices*. O'Reilly & Associates, Inc., 2003.
- [23] Munawar Hafiz, Paul Adamczyk, and Ralph Johnson. Systematically eradicating data injection attacks using security-oriented program transformations. In *ES-SoS '09: Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, pages 75–90, Berlin, Heidelberg, 2009. Springer-Verlag.
- [24] William G. J. Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05*, pages 174–183, New York, NY, USA, 2005. ACM.
- [25] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14*, pages 175–185, New York, NY, USA, 2006. ACM.
- [26] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. Vida: Visual interactive debugging. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 583–586, Washington, DC, USA, 2009. IEEE Computer Society.
- [27] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2002.
- [28] Howard, Michael. A brief introduction to standard annotation language, 2011. http://blogs.msdn.com/b/michael_howard/archive/2006/05/19/602077.aspx.
- [29] Howard, Michael. Windows vista security a bigger picture, 2011. http://blogs.msdn.com/b/michael_howard/archive/2006/06/12/windows-vista-security-a-bigger-picture.aspx.
- [30] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen 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.
- [31] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen 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.
- [32] IBM Eclipse Foundation. Eclipse, 2011. <http://www.eclipse.org/>.
- [33] SANS Institute. SANS Institute, 2011. www.sans.org.
- [34] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In Proceedings of the 2006 IEEE Symposium on Security and Privacy, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.
- [35] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *J. Comput. Secur.*, 18:861–907, September 2010.
- [36] Kaarina Karppinen, Lyly Yonkwa, and Mikael 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.
- [37] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14, pages 1–11, New York, NY, USA, 2006. ACM.
- [38] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.
- [39] Andrew J. Ko and Brad 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.
- [40] Andrew J. Ko and Brad A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In Proceedings of the SIGCHI conference on Human Factors in computing systems, CHI '06, pages 387–396, New York, NY, USA, 2006. ACM.
- [41] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In Proceedings of the 30th international conference on Software engineering, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

- [42] Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou. Sqlprob: a proxy-based architecture towards preventing sql injection attacks. In Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, pages 2054–2061, New York, NY, USA, 2009. ACM.
- [43] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In Proceedings of the 14th conference on USENIX Security Symposium - Volume 14, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.
- [44] Wim Maes, Thomas Heyman, Lieven Desmet, and Wouter Joosen. Browser protection against cross-site request forgery. In Proceedings of the first ACM workshop on Secure execution of untrusted code, SecuCode '09, pages 3–10, New York, NY, USA, 2009. ACM.
- [45] Michael Martin and Monica 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, Berkeley, CA, USA, 2008. USENIX Association.
- [46] Michael Martin, Benjamin Livshits, and Monica 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.
- [47] G. McGraw. Software security. *Security Privacy, IEEE*, 2(2):80 – 83, mar-apr 2004.
- [48] G. McGraw, B. Chess, and S. Miguez. Building security in maturity model, 2011. www.bsimm2.com.
- [49] Moodle. Moodle, 2011. <http://moodle.org>.
- [50] Moodle. MSA-08-0013, 2011. <http://moodle.org/mod/forum/discuss.php?d=101405>.
- [51] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In Proceedings of the 5th international symposium on Software visualization, SOFTVIS '10, pages 5–14, New York, NY, USA, 2010. ACM.
- [52] Gleb Naumovich and Paolina Centonze. Static analysis of role-based access control in j2ee applications. *SIGSOFT Softw. Eng. Notes*, 29:1–10, September 2004.

- [53] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In Ry-
oichi Sasaki, Sihang Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *Security
and Privacy in the Age of Ubiquitous Computing*, volume 181 of *IFIP Advances
in Information and Communication Technology*, pages 295–307. Springer Boston,
2005. 10.1007/0-387-25660-1_20.
- [54] NYTimes.com. Thieves found citigroup site an easy entry, 2011. [http://www.
nytimes.com/2011/06/14/technology/14security.html?pagewanted=all](http://www.nytimes.com/2011/06/14/technology/14security.html?pagewanted=all).
- [55] Open Web Application Security Project. Owasp top ten project, 2011. [https:
//www.owasp.org/index.php/Category:OWASP_Top_Ten_Project](https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project).
- [56] Oracle. Java Servlet. [http://www.oracle.com/technetwork/java/javaee/servlet/
index.html](http://www.oracle.com/technetwork/java/javaee/servlet/index.html).
- [57] Oracle Corporation. Java, 2011. <http://www.java.com/en/>.
- [58] Oracle Corporation. Netbeans ide, 2011. <http://netbeans.org/>.
- [59] OWASP. ESAPI. [https://www.owasp.org/index.php/Category:OWASP_
Enterprise_Security_API](https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API).
- [60] OWASP. Open Web Application Project. [https://www.owasp.org/index.php/
Main_Page](https://www.owasp.org/index.php/Main_Page).
- [61] OWASP. OWASP secure coding practices.
- [62] OWASP. ESAPI Validator API, 2011. [http://owasp-esapi-java.googlecode.com/
svn/trunk/doc/latest/org/owasp/esapi/Validator.html](http://owasp-esapi-java.googlecode.com/svn/trunk/doc/latest/org/owasp/esapi/Validator.html).
- [63] OWASP Foundation. Cross-site request forgery, 2011. [https://www.owasp.org/
index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)).
- [64] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis
methods for identifying security vulnerabilities in software systems. *IBM Syst.
J.*, 46:265–288, April 2007.
- [65] James Reason. *Human Error*. Cambridge University Press, Cambridge, UK,
1990.
- [66] Apache Roller. Apache Roller. <http://roller.apache.org>, year = 2011.
- [67] SAFECode.org. Fundamental practices for secure software development, 2008.

<http://www.safecode.org/publications/>.

- [68] SANS Institute. Cwe/sans top 25 most dangerous software errors, 2011. <http://www.sans.org/top25-software-errors/>.
- [69] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, Mar 2010.
- [70] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.
- [71] Helen Sharp, Yvonne Rogers, and Jenny Preece. Interaction Design: Beyond Human-Computer Interaction. Wiley, 2 edition, March 2007.
- [72] Fortify Software. Fortify SCA, 2011. <https://www.fortify.com/products/fortify360/source-code-analyzer.html>.
- [73] SourceForge.net. Pmd, 2011. <http://pmd.sourceforge.net/>.
- [74] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.
- [75] Blair Taylor and Shiva 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.
- [76] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.
- [77] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), pages 123–140, Vienna, Austria, July 2005.
- [78] VERACODE. State of Software Security Report Volume 1, 2, and 3, 2011.

<http://www.veracode.com/reports/index.html>.

- [79] John Viega and Gary McGraw. Building secure software: how to avoid security problems the right way. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [80] John Viega, Matt Messier, and Genen Spafford. Secure Programming Cookbook for C and C++. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1 edition, 2003.
- [81] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.
- [82] G. M. Weinberg. The psychology of computer programming. Van Nostrand Reinhold Co., New York, NY, USA, 1988.
- [83] David A. Wheeler. Secure programming for linux and unix howto, 2003.
- [84] Irene M. Y. Woon and Atreyi Kankanhalli. Investigation of is professionals' intention to practise secure development of applications. Int. J. Hum.-Comput. Stud., 65:29–41, January 2007.
- [85] Jing Xie, Heather Richter Lipford, and Bill Chu. Why do programmers make security errors? In Proceedings of 2011 IEEE Symposium on Visual Languages and Human Centric Computing, pages 161–164, 2011.
- [86] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In Proceedings of the 15th conference on USENIX Security Symposium - Volume 15, Berkeley, CA, USA, 2006. USENIX Association.
- [87] Xie, Jing and Chu, Bill and Melton, John. Owasp aside, 2011. https://www.owasp.org/index.php/OWASP_ASIDE_Project.

APPENDIX A: RULES

A ASIDE Code Refactoring Default Trust Boundary Rules

Figure 40 is a trust boundary rule for Java Servlet API `javax.servlet.ServletException.getParameter(String)`. Table 6 explains the mapping in detail.

```
3 <TrustBoundary type="MethodInvocation" attr="input">
4   <DeclarationClass><![CDATA[
5     javax.servlet.(http.Http)?ServletRequest
6   ]]></DeclarationClass>
7   <MethodName><![CDATA[getParameter]]></MethodName>
8   <ReturnType><![CDATA[java.lang.String]]></ReturnType>
9 </TrustBoundary>
```

Figure 40: ASIDE trust boundary rule for Java Servlet API
`HttpServletRequest.getParameter(String)`

Table 6: Trust boundary rule mapping.

Rule Element	Attribute	Value	Note
TrustBoundary			
	type		denote the rule is applied to identify program constructs that introduce untrusted inputs in to applications or render untrusted outputs from the applications
	att		denotes the type of program constructs this rule should be applied to. There are two types of constructs defined: MethodInvocation maps to function calls; MethodDeclaration maps to a declaration or a definition of a function
Declaration Class		javax.servlet. (http.Http)? ServletRequest	denotes the type of untrusted data associated with this boundary rule: input or output
MethodName		getParameter	denotes the namespace within which method is defined
ReturnType		java. lang. String	denotes the defined name of the method
			denotes the namespace to which the return object of the method belongs

B ASIDE Code Refactoring Default Input Validation Rules

Figure 41 is an input validation rule for data type **Email**. Table 7 explains the mapping in detail.

```

18  <ValidationPattern label="Email">
19  <Pattern><![CDATA[
20      ^[\w\-\+\&\*]+(?:\.[\w\-\+\&\*]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}$
21  ]]></Pattern>
22  <Default><![CDATA[
23      admin@gmail.com
24  ]]></Default>
25  </ValidationPattern>

```

Figure 41: ASIDE input validation rule for OWASP ESAPI **Email**

Table 7: Trust boundary rule mapping.

Rule Element	Attribute	Value	Note
Validation Pattern	label		denotes a rule is an input validation pattern denotes the type of data
Pattern		regular expression for Email	denotes the regular express to be used for validating against an input. If the input can be matched, it is considered a valid email address. If it cannot be matched, it is considered an invalid one
Default		admin@gmail.com	denotes the default value to be set if the validation fails

C ASIDE CodeAnnotate Sensitive Accessor Specification

Figure 42 is a sensitive accessor rule that defines the action updating a database table via `java.sql.Statement.executeUpdate(String)`. Table 8 explains the mapping in detail.

```
3 <accessor id="java.sql.Statement.executeUpdate(String)">  
4   <sensitiveLocation>0</sensitiveLocation>  
5   <sensitiveType><![CDATA[DB_Table]]></sensitiveType>  
6   <sensitiveTarget><![CDATA[USER]]></sensitiveTarget>  
7 </accessor>
```

Figure 42: ASIDE *CodeAnnotate* sensitive accessor rule example

Table 8: Sensitive accessor rule mapping.

Rule Element	Attribute	Value	Note
accessor	id		denotes a rule is a sensitive accessor rule denotes the fully qualified name of a method that accesses sensitive information. In this case, it is method <code>executeUpdate(String)</code> method within the namespace <code>java.sql.Statement</code>
sensitiveLocation		0	denotes the argument index at which the sensitive information is
sensitiveType		DB-Table	denotes the type of sensitive information. In this case, it is a database table
sensitiveTarget		USER	denotes the reference of the sensitive information. In this case, the name of the database table, which is USER

APPENDIX B: STUDY MATERIALS

A Secure Programming Errors Study


UNC CHARLOTTE
 The University of North Carolina at Charlotte
 9201 University City Boulevard
 Charlotte, NC 28223-0001

Woodward Hall Room 330B
 Department of Software and Information Systems
 College of Computing and Informatics

Telephone: 704-687-8388

**Informed Consent for
Investigating expert security programmers' mental models toward secure programming**

Project Title and Purpose

You are being asked to participate in a study to investigate how expert security programmers think about and perform secure programming. This study is part of research being conducted at the University of North Carolina at Charlotte in the College of Computing and Informatics, Software and Information Systems Department. This study will provide valuable information to help us to design and develop better tool support for programmers to develop secure software.

Primary Investigators

Jing Xie – Software and Information Systems
 Dr. Bill Chu – Software and Information Systems
 Dr. Heather Lipford – Software and Information Systems

Eligibility

You may participate in this study if you are at least 18 years old and employed as a software developer, and have knowledge about programming secure software. You should be fluent in English. Anyone who does not have experience with developing secure software is not eligible for this study.

Overall Description of Participation

If you choose to participate, you will be interviewed by one of our investigators about your programming experiences and practices related to security. The entire interview will be recorded via a voice recorder. Upon the completion of the interview, you will be compensated for your time and effort with a \$20 gift card. However, if you end the interview before 15 minutes, you will not be provided with a gift card.

Length of Participation

This is a one-time interview. The session will last approximately 30 minutes. You are free to leave for any reason at any time and will not be treated differently.

Risks and Benefits of Participation

The study will not result any known risks to the participants, but there may be risks unknown at present.

Volunteer Statement

You are a volunteer and willing to be recorded by a voice recorder. The decision to participate in this study is completely up to you. If you decide to be in the study, you may stop at any time. You will not be treated any differently if you decide not to participate in the study or if you stop once you have started.

Confidentiality Statement

Any information about your participation that is collected by us will be completely confidential. The following steps will be taken to ensure this confidentiality: A unique code will be generated to label your voice recording for anonymity. The generated code will have no relation to your real identity. All the recorded data will be encrypted immediately by the investigator via strong encryption algorithm PGP and stored on the password protected

Figure 43: Consent Form

research desktop located in a keyed office to which only the investigators have access in Woodward Hall at UNC Charlotte main campus. Your voice recording will be destroyed within two years.

Statement of Fair Treatment and Respect

UNC Charlotte wants to make sure that you are treated in a fair and respectful manner. Contact the university's Research Compliance Office (704-687-3309) if you have questions about how you are treated as a study participant. If you have any questions about the actual project or study, please contact Dr. Bill Chu (704-687-8661, billchu@uncc.edu) or Jing Xie (704-687-8388, jxie2@uncc.edu) or Dr. Heather Lipford (704-687- 8376, heather.lipford@uncc.edu)

Approval Date

This form was approved for use on July 06th 2010 for use for one year.

I have read the information in this consent form. I have had the chance to ask questions about this study, and those questions have been answered to my satisfaction. I am at least 18 years of age, and I agree to participate in this research project. I understand that I will receive a copy of this form after it has been signed by me and the principal investigator of this research study.

Participant Name (PRINT)

DATE

Participant Signature

Investigator Signature

DATE

Figure 44: Consent Form

Duration: 30mins**Section 1: Opening**

Questions:

1. Could you please give me a brief description about your current job? Such as what are the activities involved, what are the general goals that you want to achieve, how many hours do you spend on it per day?
2. How much programming is involved in this job? What kinds of programming languages you use? what kinds of software you develop?
3. How long have you been working on this job? Prior to this, what did you do?
4. When did you start to be aware of security in software?
5. Have you had any experience with software security incidents? Can you talk about it? Such as what happened, why it happened and how did it end?

Section 2:

Questions:

1. While you are coding, at what granularity you think about security implications? For instance, you think about security while you complete each statement or a block, a method, a class, a package or a project?
2. If you don't consider security while you are coding, when do you think about it? (After you are done with intended functions?)
3. Do you reuse code often, for example, reuse existing code that was written by yourself for other projects, or sometimes reuse other people's code for similar functions? If yes, do you examine the code for potential security vulnerabilities?
4. In your past experience, have you ever employed countermeasures which were recommended by other people for mitigating or solving vulnerabilities that you were facing? Did you have any suspicion about the security of the solutions? If yes, could you please give me an example?
5. Do you think ensuring security for software is challenging? If yes, why do think it is challenging?
6. How do you keep up with security information updates?
7. Are there any established routines that you use to ensure security of the software that you develop from either yourself or external entities such as the company, your team, or maybe colleagues? If there is any, can you please tell me what it is? How good do you think it is? (coverage, cost-effectiveness, reliability, correctness, ease of use). (If subject has a negative evaluation on it/them), what will you do to improve it/them if you are allowed to do so?
8. Do you always follow design specifications? Do you have experience with modifying specification to better accommodate your implementation? If yes, while you were modifying established specification, did you think about things such as whether this modification would take away security considerations or add in security considerations? Could you please give me an example of such experience?

Figure 45: Interview Questions

9. what are the triggers: what makes the programmer going back and forth to fix the vulnerabilities.

Section 3:

Questions:

1. What are the software security issues that you consider most in your job/programming practices? How do you define issue A, B and C?
2. Take issue A as an example, what do you think are the causes of such issue, how do you manage to prevent it from arising?
3. Have you ever resorted to external help? Such as going online to forums, or asking colleagues, etc? Were those resources helpful?
4. What is the coverage of your solution towards issue A in the given example? Is it a localized solution which targeted only the issue in question or it is a systematized solution which can thwart similar issues from emerging? (Do you consider it as a silver bullet?)
5. When you come across a security issue, to what extent do you get a mitigation? What factors do you take into consideration to weigh among different possible solutions? If possible, could you please give an example with your past experience?
6. How do measure your solution to the issue? Such as in terms of correctness, effectiveness, overhead, understandability and etc.
7. Do you think that you do enough to ensure your code is immune from all potential attacks? If not, what is insufficient? Could you please give me some examples?
8. With respect to the inadequates, do you think that you are the one to blame?
9. While you were reviewing your teammates' code after you checked in the project repository, did you have moments thinking about the security implications of their code or just assumed their code is secure?
10. How do you deal with common security issues in code such as code injection vulnerabilities?
11. What is the percentage of time you spend on security overall a normal developing session?
12. Do you feel struggling to juggle security into your programming practices? If yes, why is that? If not, could you please give me an example to show me how easy it is?

Section 4: Closing

Questions:

1. What are the existing techniques and tools in use to ensure the overall security of the software in your company?
2. Do they work well as expected? If not, what are the general problems?
3. How will you improve them if possible?

Figure 46: Interview Questions

**Informed Consent for
Evaluate Eclipse plugin ASIDE**

Project Title and Purpose

You are being asked to participate in a study to evaluate one of our two software tools called ASIDE and Whitebox. They are Eclipse plugins designed to help developers write more secure code. This study is part of research being conducted at the University of North Carolina at Charlotte in the College of Computing and Informatics, Software and Information Systems Department. This study will provide valuable information to help us to design and develop better tool support for programmers to develop software.

Primary Investigators

Jing Xie – Software and Information Systems
 Dr. Bill Chu – Software and Information Systems
 Dr. Heather Lipford – Software and Information Systems

Eligibility

You may participate in this study if you are currently taking computing course IT IS 4166/5166 offered by College of Computing and Informatics, UNC Charlotte. Anyone who does not meet the criteria is not eligible for this study.

Overall Description of Participation

If you choose to participate in our study, you will work on your assignment 4 of IT IS 4166/5166 on a lab computer that has either ASIDE or Whitebox installed in Room 337. You will be given 3 hours to work on your assignment on that computer using Eclipse. Any interaction with ASIDE or Whitebox will be logged. Your development will be screen recorded via screencasting software Camtasia from Techsmith. A technical assistant will be available for you should you have any trouble with ASIDE or Whitebox. After 3 hours, or you complete the assignment, we will interview you about your experience with ASIDE or Whitebox. The interview will be recorded via a voice recorder. Upon the completion of the interview, you will be compensated with a \$30 Amazon.com gift card.

If you withdraw from the study at any point, you will not be provided with a gift card. Your participation will be kept confidential to all the others including the course instructor but the research investigators and yourself.

Length of Participation

The study will take you up to 3 and half hours. We do not plan for any follow-up study.

Risks and Benefits of Participation

The study will not result in any known risks to you, but there may be risks unknown at present. Choosing to use ASIDE or Whitebox should not impact your completion or grade on your assignment. You are offered a gift card to compensate for the time and effort.

Volunteer Statement

You are a volunteer and willing to be recorded by a voice recorder during the interview. You also agree to be recorded by screencasting software during your development. Your interaction with either ASIDE or Whitebox will be captured in the form of log and screencasting. The decision to participate in this study is completely up to you. If you decide

Figure 47: Consent Form

to be in the study, you may stop at any time. You will not be treated any differently if you decide not to participate in the study or if you stop once you have started.

Confidentiality Statement

Any information about your participation that is collected by us will be completely confidential. The following steps will be taken to ensure this confidentiality: A unique code will be generated to label your voice recording for anonymity. The code will also be used to associate your voice recording with the application you developed and ASIDE/Whitebox log and screen recording. The generated code will not have any relation to your real identity. All the recorded data will be stored on the password protected research desktop located in a keyed office to which only the investigators have access in Woodward Hall. Your records will be destroyed after 3 years.

Statement of Fair Treatment and Respect

UNC Charlotte wants to make sure that you are treated in a fair and respectful manner. Contact the university's Research Compliance Office (704-687-3309) if you have questions about how you are treated as a study participant. If you have any questions about the actual project or study, please contact Dr. Bill Chu (704-687-8661, billchu@uncc.edu) or Jing Xie (704-687-8388, jxie2@uncc.edu) or Dr. Heather Lipford (704-687- 8376, heather.lipford@uncc.edu)

Approval Date

This form was approved for use on XXX for use for one year.

I have read the information in this consent form. I have had the chance to ask questions about this study, and those questions have been answered to my satisfaction. I am at least 18 years of age, and I agree to participate in this research project. I understand that I will receive a copy of this form after it has been signed by me and the principal investigator of this research study.

Participant Name (PRINT) _____
DATE

Participant Signature

Investigator Signature _____
DATE

Figure 48: Consent Form.

B CodeRefactoring Study

The following materials were used for study described in Section 5.4

Homework Four
Due: April 04, 2011
Points: 100

Overview:

This assignment is intended to familiarize students with more advanced functionality for maintaining server states in user applications, with a focus on transaction functionality.

For this assignment, you will need to continue working on the servlet-based e-trade application from Assignment 3. The assignment is meant to build upon the previous assignment's user registration, login, and session management functionality. Just as before: **HTML content must be generated directly and programmatically by the servlets and not by, for example, redirecting to a static HTML page.**

Changes on the HTML pages are needed for this assignment's functionalities. For a new set of sample screenshots, please click [here](#) to download.

This is an **individual** assignment.

Functionality:

- **Data Persistence Functionality**
 - Stock holdings and portfolio balances must remain persistent even after the web server has been restarted. This persistence will be provided by an external library, which you can download it [here](#).
 - The details of the external library can be found in its [API documentation](#).
 - Files that store persistence data such as users, stocks and stock categories can be found [here](#).
 - Check [this document](#) for the details of prepopulated users, stocks and stock categories.
 - View the [instructions](#) for help with proper configuration.
- **Integration with previous Assignment**
 - This assignment's functionality is intended to integrate with and enhance the functionality from Assignment 3. Assignment 3 functionality that is not explicit/updated in this assignment should be present and fully functional, but will not represent a significant proportion of the grade for this assignment. If your Assignment 3 submission was not fully functional, depending on the degree, you should finish it up if it will not detract from progress on this assignment.
- **Registration, Login, and Logout**
 - This functionality should have been completed in Assignment 3.
- **Viewing Stock Categories**
 - When viewing the categories of stocks, the sum of the funds of each stock for each category must be displayed beside the link to the individual stock category page. See the screen shots accompanying Assignment 1 for an example. Note that this needs to be dynamic, so that when the user buys or sells stock in that category, this value will change accordingly.
- **Viewing Stocks**
 - The individual category pages must dynamically display the logged-in user's currently owned shares of each stock. The total value column must also be dynamically generated, so that the value will change once the user buys or sells a particular stock (owned shares * price per share = total value).
 - The action drop-down box, quantity text field, bank account and submit button must be contained in a separate form for each stock in the list.

Figure 49: Description of Programming Task

- **View Accounts**
 - After login, the user should be able to view the nicknames and corresponding balances of the accounts that are associated with the user. Each nickname should link to a page that displays all the transactions that have been performed through the account. When viewing the list of accounts, the user should be able to delete an account(s). Deleted accounts should not be available for further stock transactions.
- **Add Banking Account**
 - The User must be able to enter new banking information, which includes account nickname, account holder name, routing number, account number and initial balance. Each account nickname should be **unique**. After submitting the account information the user should be able to use this account for further stock transactions.
 - All pages except for login and registration should link to the accounts page.
- **Transaction**
 - The selected stock information must be displayed to the user with the ability to change the quantity of the transaction. Each quantity change field and button must be in an individual form that will link back to the transaction page and update the quantity and total price of the selected stock.
 - Display current transaction information, **allowing for selection of existing bank accounts (e.g. from a drop down list). If no account has been associated with the user, redirect to the page that can add a banking account for the user.**
- **Confirmation**
 - This page must display all the information from the Transaction page with the nickname of the account, through which this transaction has been done. The quantity should not be an input form on this page and there should not be a change quantity button present.
- **View Transactions by Account**
 - This page must display all transaction performed through the selected account. The list of transactions must be dynamically generated.

Supplemental Requirements for Graduate Students (15 bonus points)

- Data persistence implementation through University Oracle Database Management System.
 - This requires students to swap the implementation of the data manipulation functions using Database operations. All function specifications should be preserved the way they are.

Graduate students addressing the supplemental requirement correctly will receive 15 extra credit points. Undergraduate students implement the supplemental requirement earn 20 extra credit points.

Assignment Submissions:

Your completed application must be loaded and available for public viewing on the school's servlet server.

What to submit using Blackboard (Email submissions will NOT be accepted):

1. **HW4.war** - An archive of the entire web application (project) stored in a standard war file. When creating the war file you must include your java source files. This war file will be imported into NetBeans for grading.
2. **info.doc** - Document with the following assignment information:
 - Link to your Login page on the servlet server (for example <http://coit-servlet01.uncc.edu:8080/username/Login>)
 - Explanation of status and stopping point, if incomplete.
 - Explanation of additional features, if any.

Figure 50: Description of Programming Task

Interview Questions:

1. Can you tell me what ASIDE/Whitebox did while you were developing your application?
2. Did you respond to ASIDE/Whitebox's reminding?
If yes,
 What did you do?
If no,
 Why not?
3. What kind of features of ASIDE/Whitebox have you used during your whole development?
4. How's the design of those warnings generated by ASIDE/Whitebox?
5. What do you like ASIDE/Whitebox most? If possible, ask participants why they like.
6. What do you dislike ASIDE/Whitebox most? If possible, ask participants why they dislike.
7. In your opinion, what can be done to make ASIDE/Whitebox better?
8. Did you learn anything from using ASIDE/Whitebox?
If yes, what did you learn exactly?
If no, continue the interview with other questions.
9. Would you pay attention to the possible vulnerable code should ASIDE/Whitebox not give you warnings?
10. Would you be able to fix the potential vulnerable code should ASIDE not provide you with possible fixes?

Figure 51: Interview Questions

Risks and Benefits of Participation

The study will not result in any known risks to you, but there may be risks unknown at present. Participants are offered a gift card to compensate for their time.

Volunteer Statement

You are a volunteer and willing to be recorded by a voice recorder during the interview. You also agree to be recorded by screencasting software during your development. Your interaction with either ASIDE or Whitebox will be captured in the form of log and screencasting. The decision to participate in this study is completely up to you. If you decide to be in the study, you may stop at any time. You will not be treated any differently if you decide not to participate in the study or if you stop once you have started.

Confidentiality Statement

Any information about your participation that is collected by us will be completely confidential. The following steps will be taken to ensure this confidentiality: A unique code will be generated to label your voice recording for anonymity. The code will also be used to associate your voice recording with the application you developed and ASIDE/Whitebox log and screen recording. The generated code will not have any relation to your real identity. All the recorded data will be stored on the password protected research desktop located in a keyed office to which only the investigators have access in Woodward Hall. Your records will be destroyed after 3 years.

Statement of Fair Treatment and Respect

UNC Charlotte wants to make sure that you are treated in a fair and respectful manner. Contact the university's Research Compliance Office (704-687-3309) if you have questions about how you are treated as a study participant. If you have any questions about the actual project or study, please contact Dr. Bill Chu (704-687-8661, billchu@uncc.edu) or Jing Xie (704-687-8388, jxie2@uncc.edu) or Dr. Heather Lipford (704-687- 8376, heather.lipford@uncc.edu)

Approval Date

This form was approved for use on **06/15/2011** for use for one year.

I have read the information in this consent form. I have had the chance to ask questions about this study, and those questions have been answered to my satisfaction. I am at least 18 years of age, and I agree to participate in this research project. I understand that I will receive a copy of this form after it has been signed by me and the principal investigator of this research study.

Participant Name (PRINT)

DATE

Participant Signature

Investigator Signature

DATE

Figure 52: Consent Form

Woodward Hall Room 330B
 Department of Software and Information Systems
 College of Computing and Informatics


UNCCHARLOTTE
 The University of North Carolina at Charlotte
 9201 University City Boulevard
 Charlotte, NC 28223-0001

Telephone: 704-687-8388

**Informed Consent for
 Evaluate Eclipse plugin ASIDE**

Project Title and Purpose

You are being asked to participate in a study to evaluate a software tool called ASIDE. It is an Eclipse plugin designed to help developers write more secure code. This study is part of research being conducted at the University of North Carolina at Charlotte in the College of Computing and Informatics, Software and Information Systems Department. This study will provide valuable information to help us to design and develop better tool support for programmers to develop secure software.

Primary Investigators

Jing Xie – Software and Information Systems
 Dr. Bill Chu – Software and Information Systems
 Dr. Heather Lipford – Software and Information Systems

Eligibility

You may participate in this study if you are currently working as a software developer and using Java as one of the programming languages on a regular basis. You should be capable of communicating in English. Anyone who does not meet the criteria is not eligible for this study.

Overall Description of Participation

If you choose to participate in our study, you will be provided with a laptop that has Eclipse and either ASIDE or Whitebox installed and functioning. You will work on the laptop to develop a Java-based web application with given design specification for up to accumulated 3 hours in a 7-day time frame with Eclipse and one of our software tools. Your interaction with ASIDE or Whitebox will be logged. Your development will be screen recorded via screencasting software Camtasia from Techsmith which is also installed on the provided laptop.

At the time you first meet with the primary investigator, you will be provided with the laptop and design specification. Once you complete the development by accumulating 3 hours development or implementing all the required functionalities, whichever comes first, you should contact the primary investigator and schedule a time to meet, during which you will be interviewed by the investigator about your experience with ASIDE or Whitebox. The interview will be recorded via a voice recorder. Upon the completion of the interview, you will be compensated with a \$30 Amazon.com gift card. The investigator will take back the laptop along with all data on it. We will give you a copy of the application you developed upon your request.

If you withdraw from the study at any point, you will not be provided with a gift card. Your participation will be kept confidential to all but the research investigators and yourself.

Length of Participation

The study will take you up to 3 and half hours. We do not plan for any follow-up study.

Figure 53: Consent Form.

Dear Participant,

First, please allow me to say thank you for participating in our study, we sincerely appreciate your help on our research.

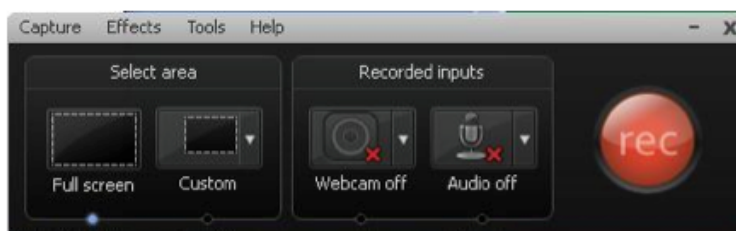
From now on, you will be in full charge of the execution of the study. To ensure a better quality of the results produced by your participation and the data for our research, we provide a set of instructions that may help you to better perform the tasks.

1. Screen Recording

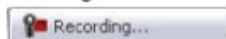
Each time you are about to start/resume your development, you should first start the screen recorder by



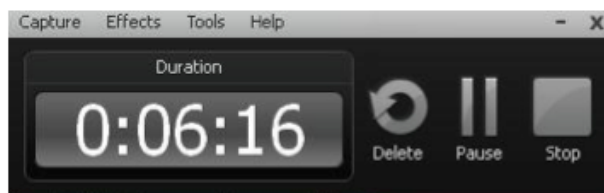
double clicking on this icon on the desktop. You will be prompted with a dialog like the one as follows:



You should then click on the red **rec** button on the above view. It will minimize to the task bar and the recording starts after 10 seconds. You can retrieve the dialog by clicking on the task icon



on the task bar. The dialog now should look like the one as follows:



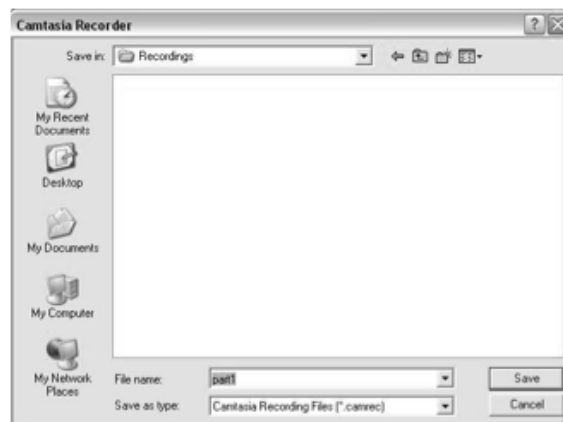
If you want to pause the development, you should just click on the **Pause** button of the retrieved recording dialog shown above, the dialog will change to one that has a **resume** button for you to click and resume the recording. **In order to get most out of your study, I highly recommend and would appreciate that you pause the recording when you are digressed from the study by, for instance, talking on the phone, going to the restroom, etc, and then resume the recording while you are ready to focus on the study.**

Figure 54: Study Instructions

Likewise, if you want to stop the development, please click on the **Stop** button. The click of the **Stop** button will bring up a **preview** view that has your recording playing. At the **bottom** of this view, you



should see a button labeled as **Save and Edit** like this: . You then click on that arrow pointing down which will bring up a menu item says **Save As...** when you click on it, it will prompt you a typical file locating dialog. I would like you to save the recording to the folder on your desktop named **Study Content->Recordings** and name the corresponding file as part1 or part2 or part3...



Since the recordings are very important data to our research, we would like to ask you to take a little bit extra care. Thank you.

2. Development

After you start **Camtasia Recorder** and position it at its recording status as instructed above, you are off to the development of the target web application. We provide a brief description of the provided development environment in this [document](#). The design specification that details the functionalities to be implemented is located [here](#).


3. ASIDE

The very first time after you launch Eclipse but before you start to write any code, you should run our software tool **ASIDE** by **right clicking** on the project **StockTrade** in the project explorer, and navigating to **ASIDE menu** on the popped up context menu, and then clicking on **Run ASIDE**. **ASIDE** will be working in the background during your whole development.

ASIDE marks code with an icon and dash-box like this

```
String username = request.getParameter("username");
String password = request.getParameter("password");
```

Figure 55: Study Instructions

You can hover over the dash-boxed text and retrieve a list of information. You can also choose to click on this devil icon  to obtain similar information about this warning. **But we strongly encourage you to explore ASIDE and we would love to hear whatever opinions you have on it.**

4. Conclusion

During your development, if you have any questions regarding the software and study, please contact me via my email jxie2@uncc.edu. I will try to respond to you as soon as I can. If you would like to do so, you can also call me at 704-425-0423.

Once you finish the study by implementing all functionalities or accumulating 3 hours development, whichever comes first, please contact me so that we can schedule a time and location for me to pick up the laptop and do the interview.

Sincerely,

Jing Xie


Figure 56: Study Instructions

Java SDK is installed at: C:\Program Files\Java\jdk1.6.0_26

Eclipse is installed at: C:\Documents and Settings\CCILOANER35\Desktop\eclipse

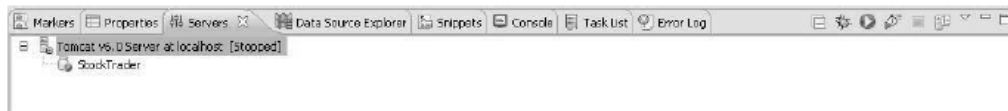
There are multiple ways to launch Eclipse, among others, you can:





1. Double click on icon  on the desktop;
2. Go to directory C:\Documents and Settings\CCILOANER35\Desktop\eclipse and double click on eclipse.exe;
3. On the task bar, single click on Start->run, then type in eclipse in the textbox and click OK.

Web server Tomcat is installed at: C:\Program Files\apache-tomcat-6.0.32

You can control it through the server pane on Eclipse:



To start tomcat when it is at *stopped* status, press ; Pressing this icon when tomcat is running will restart it.

To stop tomcat when it is running, press .

To deploy a web application project on the server, you right click on the project on the project explorer pane and navigate to **Run As-> Run on Server** and click on the selected menu item.

Figure 57: Development Environment Setup

The ultimate product is a servlet-based e-trade application. At present, this application has login, logout, registration and view stock categories of a given user functionalities implemented. You are asked to implement the following functionalities. Although we do not restrict the order of the functionalities they are implemented, we recommend they are implemented according to the given order in the specification as follows. The interface design of this application is in the given project as an attachment, here is the [link](#) to it.

Data Persistence Requirements:

Stock holdings and portfolio balances must remain persistent even after the web server has been restarted. This persistence is provided by an external library, which has already been placed properly under the [lib](#) directory of the base project in your Eclipse [workspace](#). But just in case that you need to re-import it, it can be retrieved [here](#). The source code can be viewed through Eclipse, for instance, if you navigate from **StockTrader->Java Resources->Libraries->Web App Libraries->edu.uncc.sis.base.jar->persistence** to **DataStore.class** and double click on it, you would have Eclipse open a new editor that has the read-only source code of this class.

The details of the external library APIs can be found in its [API documentation](#).

Files that store persistence data such as users, stocks and stock categories have already been placed at the proper location under the home directory of this machine. You can view them by clicking [here](#). The data stored in these files are bytes, thus you cannot view them through notepad/wordpad|

For testing purpose, we provide [this document](#) that visualizes the pre-populated users, stocks and stock categories that have been stored in the provided *dat* files.

Major Functionalities:

Login (Implemented)

Typical login function.

Registration (Implemented)

Typical registration function.

Logout (Implemented)

Typical logout function.

Viewing Stock Categories (Implemented)

This functionality allows a logged in user to view the categories of stocks, the sum of the funds of each category displayed beside the link to the individual stock category page. The displayed amount of fund is

Figure 58: Description of Programming Task

dynamic, so when the user buys or sells stock in that category, this value will change accordingly. **This functionality is already implemented in the base project.**

View Accounts

After login, the user should be able to view the nicknames and corresponding balances of the accounts, if there are any associated with the user. For the existing users we provide [and](#) also newly registered users, there is no account associated with them just yet. You need to implement the next functionality—Add Banking Account to populate account data. Each nickname should link to a page that displays all the transactions that have been performed through the account. When viewing the list of accounts, the user should be able to delete an account(s). Deleted accounts should not be available for further stock transactions.

Add Banking Account

The User must be able to enter new banking information, which includes account nickname, account holder name, routing number, account number and initial balance. Each account nickname should be **unique**. After submitting the account information the user should be able to use this account for further stock transactions.

All pages except for login and registration should link to the accounts page.

Viewing Stocks

The individual category pages must dynamically display the logged-in user's currently owned shares of each stock. The total value column must also be dynamically generated, so that the value will change once the user buys or sells a particular stock (owned shares * price per share = total value).

The action drop-down box, quantity text field, bank account and submit button must be contained in a separate form for each stock in the list.

Transaction

The selected stock information must be displayed to the user with the ability to change the quantity of the transaction. Each quantity change field and button must be in an individual form that will link back to the transaction page and update the quantity and total price of the selected stock.

Display current transaction information, **allowing for selection of existing bank accounts (e.g. from a drop down list). If no account has been associated with the user, redirect to the page that can add a banking account for the user.**

Confirmation

Figure 59: Description of Programming Task

This page must display all the information from the Transaction page with the nickname of the account, through which this transaction has been done. The quantity should not be an input form on this page and there should not be a change quantity button present.

View Transactions by Account

This page must display all transaction performed through the selected account. The list of transactions must be dynamically generated.

If you have any questions regarding this specification, please contact **Jing Xie** by email at jxie2@uncc.edu or by phone at 704-425-0423.

Figure 60: IDescription of Programming Task

Interview Questions:

1. Demographics: Can you please tell me a bit about your professional background?
2. What is your employer's attitude toward the security quality of the software you developed? Are there any concrete actions taken by your employer to ensure
3. Did you have any training that focuses on writing code that has less vulnerabilities?
4. Does your employer provide any incentives for writing secure code or any punishment for not writing secure code?
5. Do you personally take actions to make the code you write more secure?
6. Can you tell me what ASIDE/Whitebox did while you were developing your application?
7. Did you respond to ASIDE/Whitebox's reminding?
 - If yes,
 - What did you do?
 - If no,
 - Why not?
8. What kind of features of ASIDE/Whitebox have you used during your whole development?
9. What do you like ASIDE/Whitebox most? If possible, ask participants why they like.
10. What do you dislike ASIDE/Whitebox most? If possible, ask participants why they dislike.
11. In your opinion, what can be done to make ASIDE/Whitebox better?
12. Did you learn anything from using ASIDE/Whitebox?
 - If yes, what did you learn exactly?
 - If no, continue the interview with other questions.
13. Would you pay attention to the possible vulnerable code should ASIDE/Whitebox not give you warnings?
14. Would you be able to fix the potential vulnerable code should ASIDE not provide you with possible fixes?
15. Do you have any expectations on tools that help you write more secure code, produce more secure software?
16. From a developer's perspective, what do you think are needed to help you on the same subject matter, write more secure code?
17. To your knowledge, are there any tools that function similarly to ASIDE available?

Figure 61: Interview Questions