INTERACTIVE STATIC ANALYSIS FOR APPLICATION SECURITY

by

Jun Zhu

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

2015

Approved by:

Dr. Bill (Bei-Tseng) Chu

Dr. Heather Richter Lipford

Dr. Xintao Wu

Dr. Mohamed Shehab

Dr. Mark Pizzato

ABSTRACT

JUN ZHU. Interactive static analysis for application security. (Under the direction of DR. BILL (BEI-TSENG) CHU)

Software vulnerabilities have become increasingly pervasive and result in severe data and financial loss to organizations and individuals. One leading source of software vulnerabilities is the insecure code written by developers. Although vulnerabilities could be addressed through secure programming practices and there have already been a collection of well documented secure programming practices, developer continues to make same mistakes. With the rapidly growing complexity of software, security bugs are difficult to avoid.

This dissertation presents interactive static analysis as a developer-oriented hybrid framework for vulnerability detection and mitigation. This approach integrates static analysis into Integrated Development Environment (IDE) as a plug-in, facilitating two-way interaction between static analysis and developer. The goal is to assist developer in detecting and mitigating vulnerabilities during code construction phrase and solicit application specific knowledge from developer to customize static analysis as well as enable automatic placement of application sensor. Developers are not required to have any knowledge of static analysis, nor are they security experts.

To demonstrate the effectiveness of interactive static analysis, this dissertation focuses on access control vulnerability detection. This dissertation finds implicit assumptions of previous research techniques to automatically detect access control vulnerabilities might be unrealistic for most web applications through studying six open source PHP web applications. It demonstrates that a hybrid approach, such as interactive static analysis, is a

much more reasonable for detecting access control vulnerabilities.

This dissertation presents an interactive static analysis prototype for access control vulnerability detection as a plug-in in Eclipse PHP IDE [19], called ASIDE-PHP (Application Security plug-in for the Integrated Development Environment for PHP). It also presents an extensive evaluation of the prototype with six open source PHP web applications including a large project named Moodle [71]. The prototype detected 20 zero-day access control vulnerabilities in addition to finding all access control vulnerabilities detected in previous works.

Based on the interactive static analysis framework, this dissertation proposes an approach for automatic placement of application sensors to enable application-based intrusion detection systems. This work focuses on using application sensors to detect events of failed access control to detect privilege escalation attacks. It presents a proof of concept analysis of two open source projects to evaluate the effectiveness of the approach. In addition, it illustrates a model for automatically inserting application sensors into applications to detect access control events, based on an extensive case study involving six open source PHP projects.

ACKNOWLEDGMENTS

This dissertation is not possible without the guidance of my committee members, help from my friends, and support from my family.

To begin with, I would like to express my deepest gratitude to my advisor, Dr. Bill Chu, for his excellent guidance, caring and patience all the way through this dissertation research journey. It was him, who led me onto this wonderful and fulfilling research journey of application security. He not only provided me with the excellent atmosphere for doing Ph.D. research, but also gave me great advice about various aspects beyond research. I thank him for having faith in me from the beginning, and challenging me while remaining encouraging all the way through, especially the final months when I needed his support the most. I am deeply indebted to him.

I am very grateful to Dr. Heather Lipford, who has collaborated all the way through my dissertation research and advised me on every project, especially the interactive support for secure programming education project. I thank her for always offering her constructive feedback that polishes the idea and the work.

I am also very grateful to Dr. Xintao Wu, who advised me on the research of privacy preserving data mining during the first year of my Ph.D. study. I thank him for helping me develop very valuable research skills during my first year of study, and also his valuable feedback on my dissertation research as my committee member.

I would like to thank other members of my dissertation committee, Dr. Mohamed Shehab and Dr. Mark Pizzato, who have invested valuable time on me. Their valuable advice has made me improve portions of this dissertation for the better.

I would like to thank Jing Xie for her help when I started working on Eclipse plugin development. Many thanks to Erik Northrop, Jing Xie for their assistance in conducting the user study. I also would like to thank Tyler Thomas and Spike E. Dog for their help in part of the project evaluation. Thanks to all participants for their time in the user study.

I would like to thank my friends and labmates at UNC Charlotte for their help and making my Ph.D. study life much more enjoyable. Thank you Li Qi, Kang Ni, Lukun Zheng, Chen Fu, Jieyan Kuang, Ning Zhou, Mingming Fan, Michael Whitney, Berto Gonzalez, Alex Adams, Tyler Thomas and more.

Words cannot be enough to express my thanks to my parents, Wen Xiao, Huaiyi Zhu, for their years of unconditional love and support. They have been always supporting my goal and encouraging me to do my best. I dedicate this dissertation to them.

Finally, I am deeply grateful to my best friend, and wife, Chenjun Ling, who has always been supporting me and inspiring me to explore and grow in ways I could never have imagined.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Software is widely used to serve a range of applications from electronic medical records management to financial services. Software vulnerabilities have become increasingly pervasive targets for attackers resulting in data and financial loss. A recent Internet security threat report by Symantec states that an average security incident costs $591,780 for businesses [36]. In a 2013 report, WhiteHat Security analyzed over 600 corporate websites and found that 86% of them contained at least one serious security vulnerability. The number of serious vulnerabilities per website was 56 on average, and it took on average 193 days to resolve a vulnerability from the time it was discovered [22]. To achieve more secure software, defense in depth needs to be enforced for each phase of the software development lifecycle.

Roughly 50% of all security vulnerabilities result from programming mistakes [67]. Programmer errors, including security ones, are unavoidable even for well-trained programmers. One major cause of programming mistakes is software developer's heavy cognitive load dealing with a multitude of issues, such as functional requirements, runtime performance, deadlines, and security [54, 77, 95]. Many software vulnerabilities can be addressed with relatively straightforward coding practices, referred to as secure programming [31], such as performing validation on user input to prevent various forms of injection attacks. While secure programming practices have been well documented [87, 60, 79, 48, 20] to help developers, developers continue to make the same mistakes, either forgetting to apply

secure programming practices or performing them incorrectly. To make it worse, software complexity has been growing rapidly recent years, especially for web applications. Software is having ever more complicated business logic or access control logic, making it even harder for developers to write secure code.

Interactive support for secure programming in the Integrated Development Environment (IDE) [93, 92, 94] was proposed to provide reminders to developers to help them write secure code as they construct the application, similar to how a grammar checker in word processing software helps someone writing. Initial user studies showed the promise of this approach for validating untrusted input to prevent injection attacks [94]. Interactive code annotation was proposed as a conceptual idea [93, 92] as a mechanism in which developers are asked to indicate where secure programming practices were performed. However, no thorough evaluation of the effectiveness of interactive annotation was performed.

Static analysis tools are widely used to detect software flaws in code. Widely used static analysis tools include open source ones from the research communities such as Findbugs [8] and PMD [26], and commercial static analysis tools from vendors such as Fortify SCA [25], Veracode [18], and Coverity [47]. Most of their use is focused on detecting injection vulnerabilities [50, 63, 65, 97]. Access control vulnerabilities, another very important class of software vulnerabilities, are much less researched. There are some recent research works on automatic detection of access control vulnerabilities [83, 86, 37, 81, 43, 70, 40]. Although these works could detect both known and zero-day access control vulnerabilities, they have significant limitations due to their implicit assumptions.

Although ideally static analysis tools should be run regularly in the software development lifecycle to help developers fix vulnerabilities as they appear, in reality these tools

often are not used by regular developers because they require special training. In addition, a study of developer [49] revealed that false positives of static analysis tools and developers' overload during the programming phase are major contributors to the dissatisfaction and low adoption of static analysis tools by developers. Static analysis tools are usually used by security specialists in the security code review phase late in the development cycle. Upon detection of vulnerabilities, security specialists need to collaborate with developers for vulnerabilities reporting and mitigation, which means additional steps in the software development process and fixing vulnerabilities late in that process.

1.1 Interactive Static Analysis

Based on interactive code annotation [93, 92], this dissertation proposes interactive static analysis, a developer-oriented framework that integrates static analysis into the Integrated Development Environment (IDE) as a plug-in. The primary advantage of this approach is to facilitate two-way interaction between static analysis and developers and assist developers in detecting and mitigating vulnerabilities during the code construction phrase. *Plug-in to developer:* the plug-in analyzes the source code in the background and reminds developers about vulnerable code through security warnings in-situ. Developers could interact with the warnings, get the contextual explanations and mitigation suggestions about the vulnerable code, and resolve them with the assistance from the plug-in. *Developer to plug-in:* the developer is prompted by the plug-in and asked to identify and annotate application-specific logic critical for security, which was named interactive code annotation [93, 92]. The annotations are then leveraged to customize the static analysis enabling more effective static analysis, which helps discover vulnerabilities and reduce false positives without the intervention by someone with special training in static analysis to write customized rules. In

addition, by relying on the annotations, application sensors could be automatically placed, which could potentially significantly widen the adoption of using application sensors for intrusion detection.

Developers are not required to have any knowledge of static analysis, nor are they security experts. However, they do have to be familiar with basic security concepts (e.g. access control, encryption) as well as basic secure programming techniques (e.g. input validation, prepared SQL statements).

What distinguishes interactive static analysis from other forms of static analysis is that it is continuously customized and refined based on the application specific knowledge obtained from the two-way interaction between static analysis and the developers. Similar to other common static analysis tools, the static analysis of my approach utilizes a set of techniques that are often used to detect software vulnerabilities, such as data flow analysis and control flow analysis. Data flow analysis is often leveraged to detect vulnerabilities originating from improper handling of untrusted data, such as Cross-site Scripting (XSS), SQL injection. Control flow analysis can be used to detect authentication bypass and privilege escalation.

Similar to other static analysis tools, an interactive static analysis tool can be configured by an organization's Software Security Group (SSG), which is responsible for ensuring software security as identified by best industry practice [46]. SSG promotes organizational and/or application-specific programming standards through writing security specifications. For example, an SSG may list operations that are considered sensitive (e.g. inserting a weblog into the database) and that the application must provide an access control check.

I performed an in-depth evaluation of interactive static analysis with six open source

PHP web applications, including a large project named Moodle [71]. I detected 20 zero-day access control vulnerabilities in addition to finding all access control vulnerabilities detected in previous works. Based on this study I discovered the limits of related works in automatic detection of access control vulnerabilities and demonstrated the effectiveness and benefits of my approach.

1.2 Placing Application Sensors for Application Based Intrusion Detection and Prevention

No techniques or tools could detect all the vulnerabilities in a piece of software. For example, it can be easily shown that finding all buffer overflow vulnerabilities in a program using static analysis is computationally undecidable. Intrusion detection systems (IDS) [29] have been leveraged to detect signs of intrusions or attacks. For intrusion detection, most of the research efforts [28, 38, 39, 56, 72, 84, 59, 27, 35, 42, 45, 52, 53, 55, 61, 64, 69, 74, 88, 96, 78] have been focused on host-based or network-based intrusion detection, but none of them is effective in detecting application specific attacks, because they are unable to capture the application context information. For example, privilege escalation is a common attack against high value applications. In general it cannot be detected by existing intrusion detection mechanisms.

There have been some attempts, both in the research community as well as in the open source community, to achieve application layer intrusion detection. One way of application layer intrusion detection is through Web application firewall (WAF) [76], which acts as a filter and applies preconfigured rules to an HTTP conversation. WAF is able to detect common actions of a known attack sequence such as basic SQL injection or Cross-site Scripting attacks. However, because it has no insights about application specific logic, it is

unable to detect many application specific attacks, and thus it is not a sufficient prevention approach for critical high value applications, such as financial applications.

As another way of application layer intrusion detection, Watson et al. [89] proposed to use application sensors to detect attacks. It puts sensors inside the application and obtains application context information so that it is able to detect application specific attacks. OWASP AppSensor [90] provides a reference implementation of sensors to detect intrusions based on inputs, which requires developers to manually put the sensor code in places that need sensors. Requiring developers to manually write sensor code would prevent wide adoption of this approach. Little research has been performed toward how to automatically instrument applications with application sensors for intrusion detection.

Based on the interactive static analysis framework, this dissertation proposes an approach for automatic placement of application sensors to enable application-based intrusion detection systems. This work focuses on using application sensors to detect events of failed access control to detect privilege escalation attacks. It presents a proof of concept analysis of two open source projects to evaluate the effectiveness of the approach. In addition, it illustrates a model for automatically inserting application sensors into applications to detect access control events, based on an extensive case study involving six open source PHP projects.

1.3 Summary of Contributions

Contributions of this dissertation include,

1. I propose a framework, interactive static analysis, that helps developers write more secure code and mitigate access control vulnerabilities during code construction phase.

It also enables customized static analysis for vulnerability detection and automatic placement of application sensors for intrusion detection.

2. I found implicit assumptions of previous research techniques to automatically detect access control vulnerabilities is unrealistic for most web applications. This is established through studying six open source PHP web applications. I demonstrate that a hybrid approach, such as interactive static analysis, is a much better approach for detecting access control vulnerabilities.

3. I build an interactive static analysis prototype for access control vulnerability detection as a plug-in in Eclipse PHP IDE [19], called ASIDE-PHP (Application Security plug-in for the Integrated Development Environment for PHP). I conduct an extensive evaluation of the prototype using six open source PHP web applications including a large project named Moodle [71], and detect 20 zero-day access control vulnerabilities in addition to finding all access control vulnerabilities detected in previous works.

4. The scalability of application-based intrusion detection systems (IDSs) is significantly limited by the lack of an easy way for placing application sensors, I propose automatic placement of application sensors based on interactive static analysis. Focusing on using application sensors for intrusion detection of privilege escalation attacks, I propose to detect privilege escalation attacks by relying on application sensors associated with access control failures, and conduct a proof of concept analysis of two open source projects which demonstrates the effectiveness of my approach. In addition, I illustrate a model for automatically inserting application sensors into ap-

plications to detect access control events, based on an extensive case study involving six open source PHP projects.

1.4 Dissertation Structure

The rest of this dissertation is organized as follows.

Chapter 2 provides a survey of research fields that are closely related to this dissertation. These include vulnerabilities detection techniques, static analysis tools, annotations, and interactive support for secure programming in the IDE.

Chapter 3 describes a comparative study of six open source PHP applications that finds that implicit assumptions of previous research techniques can significantly limit their effectiveness to automatically detect access control vulnerabilities.

Chapter 4 presents a more effective hybrid approach to mitigate access control vulnerabilities. Developers are reminded in-situ of potential access control vulnerabilities, where self-review of code can help them discover mistakes. Additionally, developers are prompted for application-specific access control knowledge, providing samples of code that could be thought of as static analysis by example. These examples are turned into code patterns that can be used in performing static analysis to detect additional access control vulnerabilities and alert the developer to take corrective actions. This chapter presents an implemented prototype, a plug-in in Eclipse PHP IDE [19] named ASIDE-PHP (Application Security plug-in for the Integrated Development Environment for PHP).

Chapter 5 describes the evaluation of six open source applications with the ASIDE-PHP, which detected 20 zero-day access control vulnerabilities in addition to finding all access control vulnerabilities detected in previous works.

Chapter 6 presents an approach, based on interactive static analysis, to automatically place sensors in applications to detect privilege escalation, a common type of application level attack. This chapter presents a proof of concept analysis of two open source projects and demonstrates the effectiveness of privilege escalation detection by relying on application sensors associated with access control failures. I illustrate a model for automatically inserting application sensors into applications to detect access control events, based on an extensive case study involving six open source projects.

Chapter 7 summarizes dissertation outcomes and outlines some future research directions.

CHAPTER 2: RELATED WORKS

Related works closely relevant to this dissertation falls into 5 strands: Static analysis tools, automatic software vulnerability detection techniques, code annotation for vulnerability detection, interactive support for secure programming, and intrusion detection and prevention. Among them, the related works on intrusion detection and prevention are less related to the generalized framework interactive static analysis, and thus I put the related works on it under the specific chapter on application based intrusion detection.

2.1 Static Analysis Tools

Static analysis tools are widely used to detect software defects in code. In this section, I will focus on surveying static analysis tools, research works on static analysis techniques will be surveyed in next section. Widely used static analysis tools include open source ones from the research community such as Findbugs [8] and PMD [26], and commercial ones from industry vendors such as Fortify SCA [25], Veracode [18], and Coverity [47]. According to McGraw [66], the bulk of the benefits of static analysis come from customization: such as writing custom rules or custom detectors. Effective detection of software vulnerabilities often requires application-specific knowledge for several reasons. First, default rules for static analysis often generates many false positive warnings. One empirical study [57] revealed that both Fortify SCA [25] and Coverity Prevention [47] had significant false positive rates that imposed serious impact on the effectiveness of analysis. Static analysis false positives are generally caused by the lack of application specific knowledge. For

example, default rules in static analysis tools warn developers to validate untrusted input, regardless of whether proper input validation code has already been added. To suppress these warnings in cases when input validation functions have been provided, application specific knowledge must be incorporated into the static analysis. Second, applications often must satisfy certain security invariants or constraints, e.g. access control requirements. Such invariants or constraints are by nature application specific. In order to accurately detect missing security invariants or constraints, application specific knowledge must be incorporated into the analysis.

Static analysis tools provide different ways for incorporating application specific knowledge. Commercial static analysis tools such as Fortify SCA [25] support it through the writing of custom rules. Some open source static analysis tools such as PMD [26] support it through the writing of custom checkers. Putting specific textual annotations in the source code is also a way supported by many static analysis tools such as Fortify SCA [25], Coverity [47] and Klocwork [17], for incorporating application specific information. All of these ways requires special training for developers. The first two ways require training in writing custom static analysis rules or detectors, and thus usually the rules or detectors are written by security specialists requiring close collaboration between security specialists and developers, because developers are the only ones that know the application specific information. As a result, these two ways render high cost to achieve the customization. The third approach, writing specific textual annotations in source code, requires the learning of a specific annotation language and how to apply the correct annotations for different cases, and remembering to write the corresponding annotations when encountering cases where annotations are needed. This not only puts an extra training requirement on developers

but also an extra cognitive load requirement that developers are unlikely to always fulfill, since developers have already been heavily loaded with many priorities and tend to forget or make mistakes [54, 77, 95]. And thus, all the major available ways of incorporating application specific information into the static analysis have significant drawbacks preventing their wide use in practice.

My interactive static analysis approach attempts to overcome those drawbacks. My approach reminds developers about vulnerable code and asks for specific code annotations by displaying warnings alongside the corresponding lines of code. Developers could annotate a code snippet in a point-and-click fashion under the guidance of a concise contextualized explanation and instructions attached to the warnings. There is no need to learn a new specification language or textual annotation language and with little requirements for close collaboration between security specialists and developers in the process. Moreover, developers are not required to initiate the annotation process; they are not required to always keep in mind they need to write annotations for particular cases when they are writing their code. Developers would be prompted with a warning requesting for specific annotations when annotations are needed.

Ideally static analysis tools should be run regularly in the software development lifecycle to help developers fix vulnerabilities as they appear. However, in practice these tools often are not used by regular developers because they require special training, instead, they are usually used by security specialists in the security code review phase late in the development cycle. And thus, upon detection of vulnerabilities, security specialists need to collaborate with developers for vulnerability reporting and mitigation, which means additional steps in the software development process and fixing vulnerabilities late in the

development process. Recently, with the aim of helping developers detect and fix vulnerable code within the development environment early in the software development process, several vendors including Coverity [47] and GrammaTech [16] integrated their static analysis with an Integrated Development Environment (IDE) such as Eclipse, which displays static analysis generated warnings besides vulnerable code in the IDE's editor and a code issues review dashboard in the problem view of the IDE through a plugin in the IDE. Although their integration might be a valuable attempt to help developers detect and mitigate vulnerabilities in the IDE, it does not provide any benefits for static analysis customization, and thus a lot of vulnerabilities, such as logic flaws that relate to application specific information, may still remain unsolved. The proposed interactive static analysis approach is able to obtain application specific knowledge from developers directly and enables the customization of static analysis, which could make it more easily realize the full benefit of using static analysis to detect software vulnerabilities.

2.2 Automatic Software Vulnerability Detection Techniques

Automatic vulnerability detection techniques include static analysis, dynamic analysis, and hybrid analysis.

2.2.1 Static Analysis to Detect Software Vulnerabilities

Injection vulnerabilities usually result from improper handling of untrusted data. Common injection vulnerabilities include SQL injection, Cross-site Scripting (XSS), Command Injection, etc. Research into injection vulnerability detection has a long history, as many static analysis techniques have been proposed to detect injection vulnerabilities via data flow analysis [50, 63, 65].

Access control vulnerabilities stem from failure to properly check access credentials

before granting access to sensitive resources. Compared with injection vulnerabilities, detection of access control vulnerabilities using static analysis is much more difficult because access control logic varies a lot across different applications. A number of techniques have been proposed to address specific types of access control models such as vulnerabilities associated with role-specific access control [40, 81, 82, 85]. Son et al. [82] proposed a static analysis and code transformation approach to detect access control errors along with providing fixes automatically. It constructs an access control template (ACT) which can be based either on user input or program analysis. ACT is then used to find access violations and propose fixes. Doupe et al. [40] proposed a control flow path based approach to detect a specific type of vulnerability called Execution After Redirect (EAR), which causes the application to continue execution after intended redirection and thus leads to violation of intended access control and unauthorized execution of security-sensitive code. There are other research works on access vulnerability detection, I will describe them in the related work comparison section in Chapter 3 and compare them with my approach there.

These approaches detect access control vulnerabilities via some automatic program analysis techniques. Strong assumptions (e.g. a limited model of role-based access control or SQL-based database access) are often made for the analysis to be effective, making them difficult to apply to real-world applications, which often involve exceptions. In contrast, my interactive static analysis approach relies on developers to provide application specific information to help with static analysis and does not impose any assumptions on how applications are built, which represents a much wider effectiveness and practical impact in vulnerability detection.

2.2.2 Dynamic Analysis to Detect Software Vulnerabilities

Different from static analysis, dynamic analysis requires the instrumentation and execution of the target application in order to perform the analysis. It observes applications' runtime behaviors through execution.

Felmetsger et al. [41] proposed an approach to detect application-specific logic flaws. The approach first extracts potential invariants for function parameters and session variables through dynamic execution, and then identifies violations for these inferred invariants through model checking together with symbolic execution of application source code. Specifically, it only utilizes the most likely real invariants that corresponds to explicit checks on the execution control path in the code and involves comparison between persistent database object and session variables. Newsome et al. [73] automatically detects overwrite attacks via dynamic taint analysis towards compiled binary program, it rewrites binary at runtime and can detect majority of overwrite attacks with no false positives. Bisht et al. [32] focused on detecting a special logic vulnerability which results from the inconsistent validation of the parameters between server-side code and client-side code in forms of web applications. It is a black-box analysis approach. It generates malicious inputs as well as benign inputs based on the parameter constraints found by its analysis, and feeds both inputs into the web application to detect vulnerabilities. One vulnerability is identified if the responses from the web application with both inputs are the same. [33] improves the analysis precision of [32] through leveraging white-box analysis.

Since my approach is based on static analysis techniques, my approach differs from dynamic analysis in the same aspects as static analysis differs from dynamic analysis. Static

analysis does not require the instrumentation and execution of the target application, and thus it can be applied anytime when developers are writing their code. Static analysis conservatively identifies all potential vulnerabilities which produce false positives. Dynamic analysis has better analysis precision compared with static analysis, but it can not guarantee the completeness of its analysis.

2.2.3 Hybrid Analysis to Detect Software Vulnerabilities

Hybrid analysis combines the advantages of both static and dynamic analysis in order to further increase the accuracy of the analysis.

Balzarotti et al. [30] targets detecting weak or faulty sanitization, which they argued could not be detected by solely static analysis or dynamic analysis. They described Saner, a tool that verifies the correctness of sanitization routines. It first captures and builds a model for how untrusted input is sanitized through conservative static string analysis, and identifies weak or faulty sanitization by generating a large collection of malicious inputs as attack vectors to exploit the sanitization routines that are regarded as suspicious.

Monica et al. [58] proposed a holistic approach that combines model checking, dynamic checking, runtime detection and static analysis. Specifically, model checking is utilized to increase the accuracy of static analysis. Given a property or specification, model checking is able to verify the correctness of a system by exploring the space of the finite-state system.

Hybrid analysis techniques focus on vulnerability detection with increased accuracy while my approach focuses on providing in-situ support to help developers in detecting and mitigating vulnerabilities while they are writing their code.

2.3 Code Annotation for Vulnerability Detection

Asking developers to provide security related annotations has been shown to be very effective at detecting security vulnerabilities (e.g. Microsoft SAL [68], FindBugs [8] annotation language, and annotation toolkit [75]). However, majority of existing annotation approaches are text-based and rely on the initiatives of the developers, which means the developer must learn the annotation language and then remember to write those annotations under appropriate circumstances when the developer is writing code. Given that omission is a common cause of error by developers, developers would benefit from assistance in remembering and performing security annotations.

2.4 Interactive Support for Secure Programming

Interactive support for secure programming in the Integrated Development Environment (IDE) [93, 92, 94] was proposed to provide reminders to developers to help them write secure code as they construct the application, similar to how a grammar checker in a word processing software helps someone writing. Initial user studies showed the promise of this approach for validating untrusted input to prevent injection attacks [94]. Interactive code annotation was proposed as a conceptual idea in [93, 92] as a mechanism in which developers are asked to indicate where secure programming practices were performed. However, no thorough evaluation of the effectiveness of interactive annotation was performed.

CHAPTER 3: EMPIRICAL STUDY OF RELATED WORKS ON AUTO DETECTION TECHNIQUES FOR ACCESS CONTROL VULNERABILITY DETECTION

Access control vulnerabilities due to developer mistakes have been consistently ranked amongst the top software vulnerabilities [21]. Previous research efforts have primarily concentrated on using automatic program analysis techniques to detect access control vulnerabilities [83, 86, 37, 81, 43, 70]. While these works have led to discovery of access control vulnerabilities in open source projects, there has not been any study evaluating their limits. I conducted a comparative study using six open source PHP applications, many of them used by related works as part of their evaluations. I found that the implicit assumptions made by these previous research techniques have significant limitations.

3.1 Summary of Related Works on Auto Detection Techniques for Access Control Vulnerability Detection

To detect access control vulnerabilities, one must first have a correct access control model. Since usage of formal access control model specification languages are not widespread, such a model could be constructed either manually or inferred using data mining techniques. Previous researchers have attempted to automatically learn a correct access control model from source code and use it to detect likely vulnerabilities [83, 86, 37, 81, 43, 70]. I collectively refer to these approaches as auto detection.

Tan et al. [86] detects access control vulnerabilities through mining the program for program-wide patterns and considers deviations or anomalies as bugs. Dalton et al. [37]

detects authentication and access control vulnerabilities by performing dynamic information flow tracking on user credentials. It ensures that only properly authenticated users could access privileged resources, and prevents authentication bypass and access control attacks based on untrusted user credentials.

Son et al. [81] attempted to detect access control vulnerabilities automatically by exploiting several heuristics about the way programs are structured. First, code that implements distinct user role functionality and its access control checks reside in distinct methods and files. They compute commonality values between different files and partition the files into roles using a commonality threshold. Second, they compute branch asymmetry values, the ratio of statements in the successful branch vs. failed branch. The heuristic is that a security check is likely to have a large asymmetry number. So an asymmetry threshold value might suggest candidate access control checks. Based on the above two heuristics, they then infer correct access control checks as repetition of access control checks above a certain threshold in the same role.

Gauthier et al. [43] proposed to detect security weaknesses and vulnerabilities based on the heuristic that syntactically similar security-sensitive code fragments should be protected by similar checks. They first detect security-sensitive code clones based on a query-radius threshold value, and then based on the heuristic that access control checks are usually correctly enforced in majority cases within these code clones while wrongly enforced in minority cases, they identify the minority cases as security weaknesses or vulnerabilities.

Monshizadeh et al. [70] defines a four-tuple authorization context representing the access control rules, including developer identification of (global) access control variables. It detects privilege escalation vulnerabilities based on the consistency analysis on authoriza-

tion contexts. Inconsistencies in authorization context are regarded as vulnerabilities.

3.2 Access Control Vulnerability Taxonomy

To better characterize approaches for access control vulnerability detection, I developed a taxonomy of access control software flaws. This taxonomy is developed based on the point of view of vulnerability detection and mitigation. I use the term Security Sensitive Operation (SSO) to refer to an operation that requires access control checks (e.g. updating a database table). The types of access control vulnerabilities targeted by previous research can be described by the following taxonomy.

(1) *Missing*: If there is no access control checks on an execution path leading to an SSO.

Below is an example which has a *Missing* flaw. Because *query("INSERT INTO account...")* is an SSO, but there is no access control checks on the execution path leading to it.

```
query("INSERT INTO account ...");  //SSO
```

(2) *Inconsistent*: There exist duplicate instances of the same SSO in the code but the control checks for them are not implemented consistently while their intended access control policy should be equivalent.

Listing 1, 2 are two code examples sharing the same SSO *query("INSERT INTO blog...")*, their access control check policy should be equivalent, but their implementations shown in italics are different, and thus there exists an *Inconsistent* flaw.

Listing 1: Code example with wrong access control checks (access control checks are shown in italics)

```
require_login();
require_capability('insertBlog', $context);
query("INSERT INTO blog ...");  //SSO
```

Listing 2: Code example with correct access control checks(access control checks are shown in italics)

```
require_login();
if (isguestuser()) {
print_error('noguests','chat');
}
query("INSERT INTO blog ...");  //SSO
```

(3) *Untrusted data*: access control check is based on untrusted variables, such as variables coming from untrusted data sources that can be manipulated by an attacker.

Below shows a code example which has an *Untrusted data* flaw. The Boolean expression in italics *session_id='$id'* is an access control check for the SSO DELETE FROM sessions, it is constraining the *session_id* field with a variable *$id*, which is an untrusted variable because it comes from a common untrusted data source *$_GET*, and thus, this example has an *Untrusted data* flaw.

```
$id=(int)$_GET['session_id'];  //untrusted data
query("DELETE FROM sessions WHERE session_id='$id'");  //SSO
```

Table 1: Types of access control vulnerabilities that each technique could solve.

	ASIDE-PHP	**[37]**	**[81]**	**[43]**	**[70]**
Missing	✓				
Inconsistent	✓		✓	✓	✓
Untrusted data	✓	✓			✓
Logic errors					

(4) *Logic errors*: Other logic errors in an access control check. These are logic errors for access control that have no known method for automatic detection.

Below shows a code example which has a *Logic error*. The Boolean expression in italics is the access control check for the SSO INSERT INTO blog, which consists of two Boolean expressions concatenated by an operator "Or". One Boolean expression checks whether the username input by the user matches the username stored in the database; the other checks whether the password input by the user matches the password stored in the database. It has a logic error, because it uses a wrong operator "Or" to concatenate the two Boolean expressions. The correct way is to use "And", because it should check whether both the username and password input by the user match the record stored in the database.

```
if ( $username_input_from_user == $username_in_db Or $password_input_from_user ==
    $password_in_db ) {
query("INSERT INTO blog ...");  //SSO
}
```

Table 1 summarizes types of errors addressed by previous research. The ASIDE-PHP column represents my work, which will be discussed in detail later in the chapter.

Most auto detection approaches learn a correct access control model based on observed access control patterns. Throughout this chapter, I use the term access control check to

Table 2: Projects used in evaluation.

Project	LOC	Description
Mybloggie 2.1.3 [70]	8874	Blogging system
SCARF 1.0 [70]	1318	Conference discussion forum for papers
Wheatblog 1.1 [81]	4032	Blogging system
Bilboblog 0.2.1 [37]	2000	Blogging system
PhpStat 1.5 [37]	12,700	Application presenting IM statistics
Moodle [43]	625,000	Course management

refer to a fragment of code that implements an access control policy. An *access control pattern* is a tuple of *(access control check, SSO)* representing a code instance where the access control check is performed for the SSO. Inferring the correct access control model for a given SSO requires multiple access control patterns. Once the access control model for an SSO is learned, one can easily identify vulnerabilities as code instances where the correct access control check does not appear. Therefore in [81, 43, 70] detecting missing access control is a special case of inconsistent access control checks. A natural question then arises: do applications have sufficiently large numbers of access control patterns to support learning of an accurate access control model. I conducted a study of six open source projects to find out.

3.3 Empirical Evaluation of Auto Detection Techniques

I selected six open source PHP-based projects for this evaluation. They are summarized in Table 2. References next to each project indicate the papers in which a given project has been used as part of their evaluations.

I focused my evaluations on SSOs that are database operations. The techniques discussed here can be easily extended to other type of SSOs such as file operations. Besides Moodle, all other projects have easy to understand functions and a small number of database tables so it is fairly straightforward to identify security sensitive database operations through code reviews. I manually examined each SSO instance and identified access control checks in the

Table 3: Distribution of access control patterns in projects.

Project	Insert	Update	Delete	Select
Mybloggie 2.1.3	2,2,2,1,1	2,2,1,1,1	2,2,1,1	
SCARF 1.0	1,1,1,1,1,1,1	4,3,2,2,1,1,1	2,1,1,1,1	
Bilboblog 0.2.1	1	1	1	
Wheatblog 1.1	1,1,1,1,1,1	2,1,1,1,1,1,1	1,1,1,1,1	2,2,1
PhpStat 1.5	2,1,1,1,1,1	6,6,3,2,1,1	2,1,1	

source code. Table 3 summarizes my findings for projects MyBloggie, SCARF, Biboblog, Wheatbog, and PhpStat. Numbers in the table are access control patterns found. Columns headed with insert, update, delete, and select represent types of SSOs involved in these access control patterns. For example, for project SCARF, there are five access control patterns involving database delete operations. One pattern is repeated twice while the rest are not repeated. It is clear that in the majority of cases, an SSO either appear only once in an access control pattern or the number of patterns it appears in is very small.

Because of the large number of tables in Moodle (over 250) as well as Moodle's large code base, I identified SSOs as database operations involved in known access control vulnerabilities (details described in Section 5.1). I believe this gives an accurate estimated list of SSOs because Moodle is professionally maintained and has very good documentation of security fixes. I identified 31 SSOs in Moodle following this method and manually annotated access control logic associated with each of them. Figure 1 shows my results on the distribution of access control patterns involving these 31 SSOs. Each bar represents the number of repeated access control patterns. The median number of repetitions for an access control pattern is one.

Low repetitions of access control patterns limit the performance of auto detection approaches. At one extreme, consider the case that an access control pattern only appears once in the code. The auto detection approach will not be able to construct a correct access

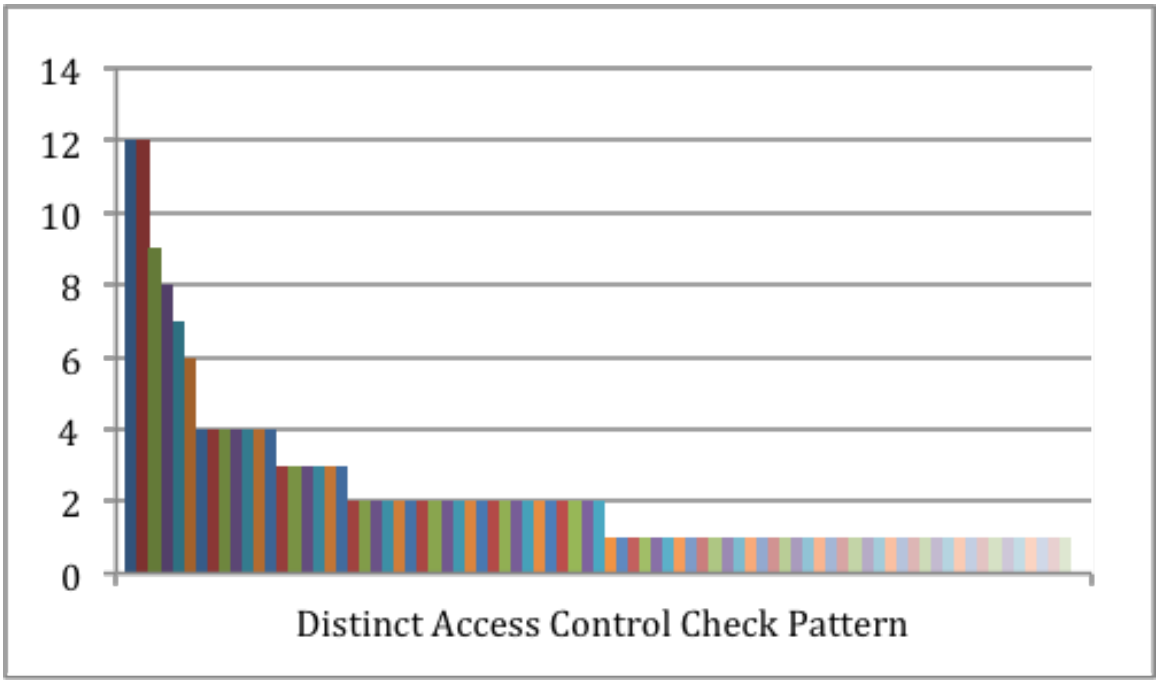

Figure 1: Distribution of access control patterns in Moodle.

control model for such a case. More generally, the confidence of the constructed access control model depends on the size of repeated access control patterns as well as ”noise” in code instances. Here ”noise” refers to branching logic that may look like access control but is in fact not related to access control (e.g. business logic to select what table to update depending on application context). If code instances contain much non-security related branching logic in addition to access control checks, a larger number of code pattern repetitions are needed to ”learn” how to separate the two kinds of branching logic. Threshold values were used by auto detection approaches to make these decisions. However, it is not clear how the combination of size of access control pattern repetition and branching complexity affect the accuracy of access control model extraction. It is also not clear how thresholds tuned for one project can be applied to another project.

Furthermore, auto detection techniques typically assume that there is only one unique

access control check for a given SSO. This is however not always the case. I will provide concrete examples where an application may have multiple access control checks for the same SSO in Section 5.3.

At the observed level of repetition of access control patterns, auto detection approaches will have large false negatives (details in Section 5.2). I am thus motivated to take a hybrid approach in which I seek developer input to identify access control models for SSOs. I argue that such an approach will be more effective provided that it does not cause too much distraction to the developers. I will describe the hybrid approach in next Chapter.

CHAPTER 4: INTERACTIVE STATIC ANALYSIS FOR ACCESS CONTROL VULNERABILITY DETECTION

As described in Chapter 3, I conducted a comparative study using six open source PHP applications to evaluate the limits of previous research works on automatic detection of access control vulnerabilities, which found that the implicit assumptions made by these previous research techniques have significant limitations. Detecting access control vulnerabilities automatically is hard because it is difficult to infer intention of the developer based on source code alone. I propose a hybrid approach called *interactive static analysis*. In this chapter, I will describe the interactive static analysis framework in details, and use concrete examples and my prototype implementation to illustrate it. I will describe the evaluation of the framework and prototype in Chapter 5.

4.1 Interactive Static Analysis Example

I refer to my prototype implementation as ASIDE-PHP (Application Security plug-in for the Integrated Development Environment for PHP). In this section I use an example from the open source project Moodle to illustrate key concepts of interactive static analysis and how it might work in practice. A Moodle chat room is created by a teacher for a specific course. Only logged-in users with required chat capability should send messages to the chat room. Listing 3 and Listing 4 show two code snippets from two different Moodle files that I use for this example. Listing 4 has an access control vulnerability, which I will explain below.

Listing 3: Code snippet from /mod/chat/gui_basic/index.php of Moodle

```
require_capability('mod/chat:chat', $context);
$DB->insert_record('chat_messages', $newmessage);
$DB->insert_record('chat_messages_current', $newmessage);
```

Listing 4: Code snippet from /mod/chat/gui_sockets/index.php of Moodle

```
if(isguestuser()){
print_error('noguests','chat');
}
//chat_login_user() calls database insertion to tables
   chat_messages, chat_messages_current
if (!$chat_sid = chat_login_user($chat->id, 'sockets',
$groupid, $course)) {
    print_error('cantlogin');
}
```

Identifying SSOs is an important step in a secure software development lifecycle (SSDLC) [46] because knowing which data elements have security implications is critical for threat modeling [80]. A Software Security Group (SSG) within organizations often coordinates the SSDLC activities with development teams. In such an environment, security sensitive operations (e.g. operations SELECT, INSERT, UPDATE, DELETE on specific database tables) will be identified. This is the primary information required as input to interactive static analysis. My example involves two Moodle database SSOs: INSERT to

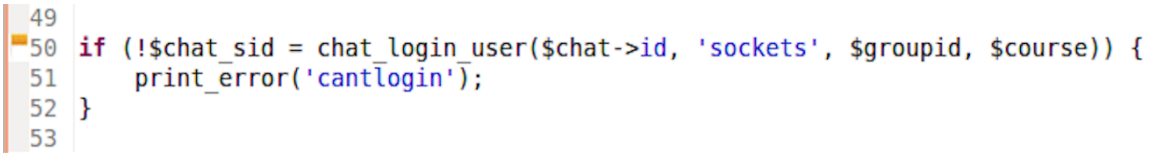

Figure 2: An annotation request is generated (yellow marker).

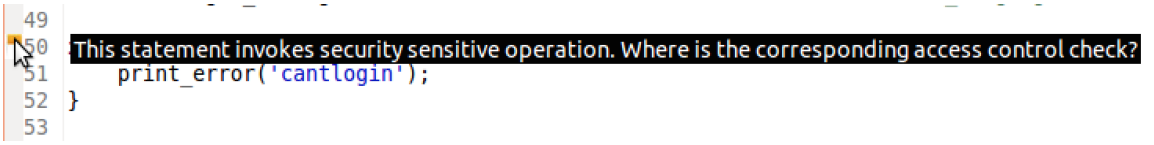

Figure 3: Developer hovers over the yellow marker to get a quick tip.

chat_messages and *chat_message_current*.

ASIDE-PHP continuously analyzes the source code in the background as the developers write code in Eclipse. When an SSO is detected, a yellow notification is placed alongside such as in line 50 in Figure 5 because it involves two Moodle database SSOs: INSERT to *chat_messages* and *chat_message_current*. The developer can hover over the notification icon to get a quick tip about what the notification icon means, as shown in Figure 3. Developer could click on the notification to interact with ASIDE-PHP, as shown in Figure fig:asidephpReadmore, a list of options is shown to developer, developer could click on the first option, which will open a web page within Eclipse providing more explanations about the annotation request and related information. When developer chooses the second option ”make annotation” as shown in Figure 5, an explanation box will be shown besides the option to illustrate what the option means and what the developer should do. In this example, ASIDE-PHP is asking for the developer to highlight access control logic for inserting data into tables *chat_messages* and *chat_messages_current*. The function called on line 50 in Figures 5 invokes those database insertion operations.

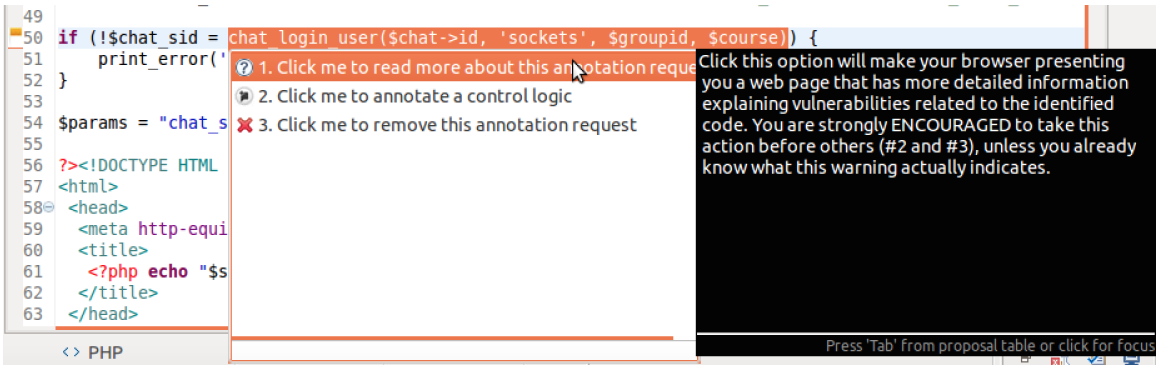

Figure 4: Developer could click the readmore option, which will open a webpage within Eclipse providing more explanations about the annotation request and related knowledge.

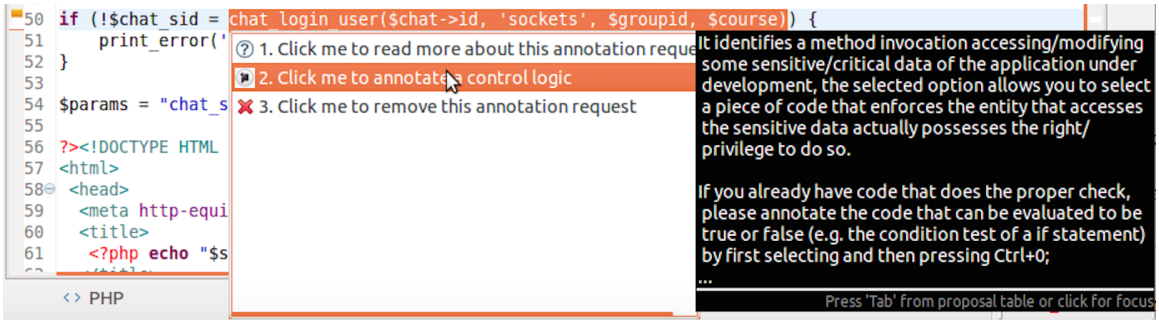

Figure 5: Explanation for interactive annotation.

The developer selects the code snippet that he thinks is access control check as shown in Figure 6, and performs the annotation by highlighting the code containing the intended access control logic as shown in Figure 7 (lines 27 and 29 in green). This process is referred to as interactive annotation because no additional language syntax is required and it is integrated into the IDE. By providing the annotation, the developer is reminded to write code for access control checks, and will have an opportunity to reflect on the checks as they are being annotated. This reflection provides an additional opportunity for developers to self-review their code and notice mistakes they or others may have made. One nice effect of interaction annotation is that it can quickly identify cases where the developer has inadvertently left out the access control logic for a SSO entirely (the missing error type

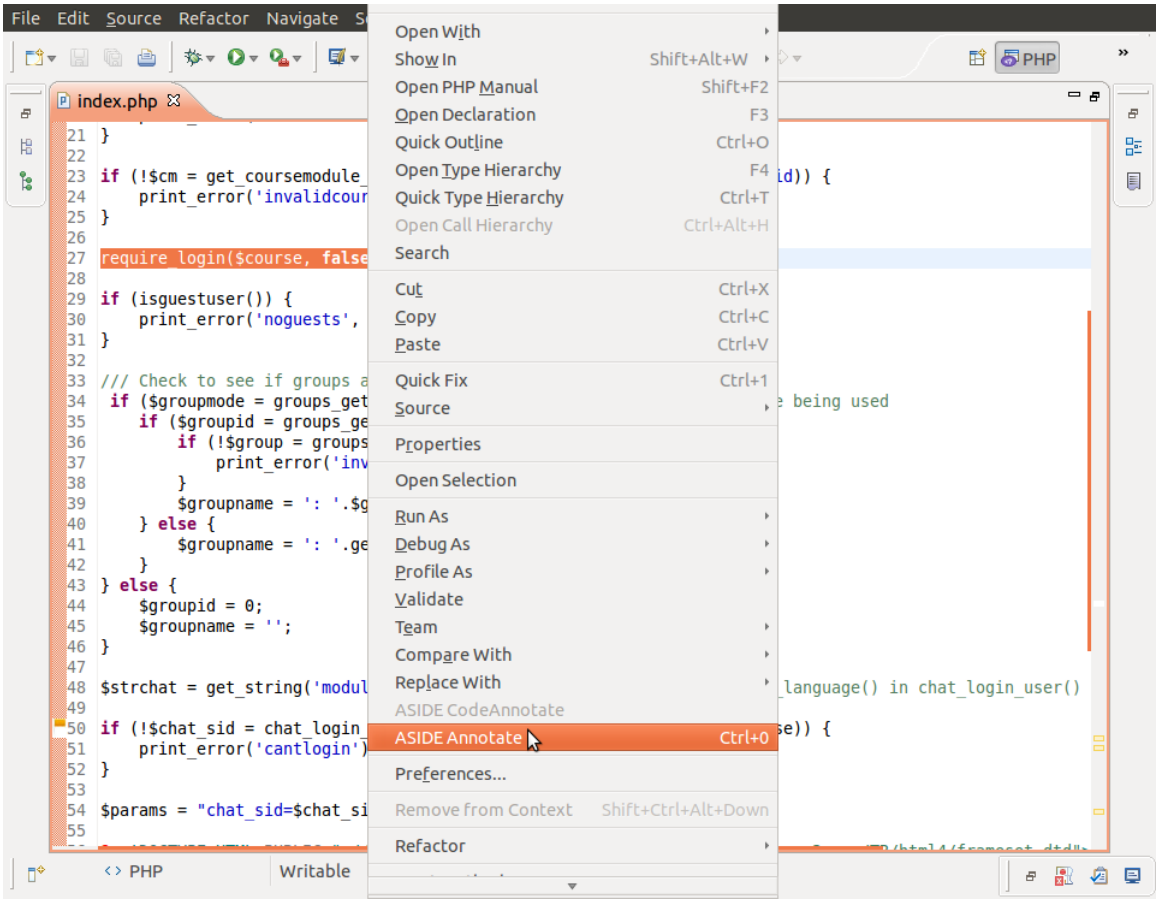

Figure 6: Developer selects a code snippet and annotates it as a control check.

discussed in Section 3.2).

ASIDE-PHP uses static analysis techniques to detect vulnerabilities. In this example, ASIDE-PHP noticed that both code snippets from Listing 3 and Listing 4 perform the same SSOs: inserting records to tables *chat_messages* and *chat_messages_current*, but the two access control checks are different. This is an example of using duplication of access control checks to find inconsistent access control checks for the same SSO. ASIDE-PHP alerts the developer that there is a likely vulnerability, as shown by the red mark besides line 50 in Figure 7. This example is indeed a known Moodle vulnerability, CVE-2013-2242, where an unauthorized user can insert messages to a chat room. The vulnerable code in

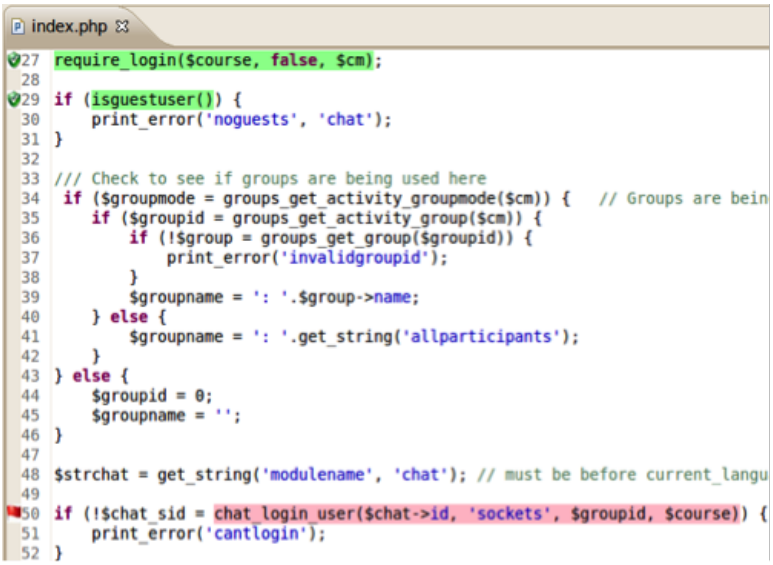

Figure 7: Access control checks (above green highlight), and a vulnerability warning (lower red highlight).

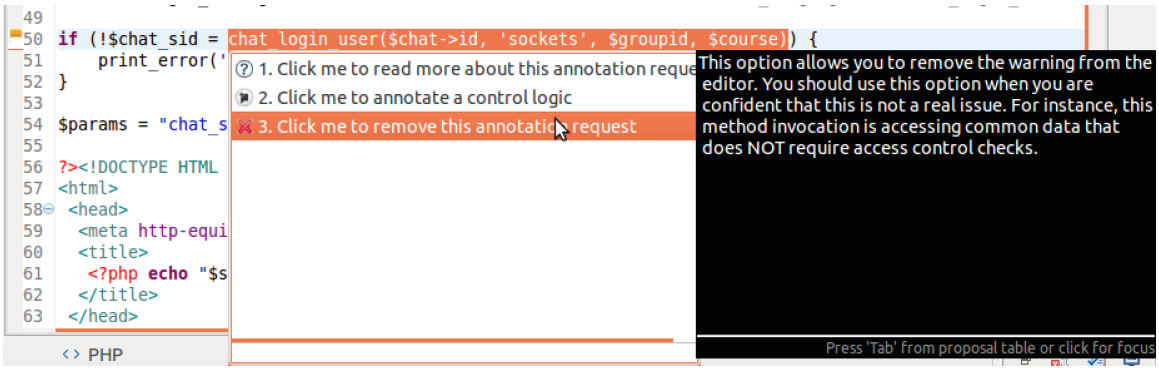

Figure 8: Developer could click this option to remove the annotation request if developer thinks this line of code does not require access control check.

Listing 4 only checks to make sure a user is not a guest user but did not ensure the user has the authorization for a particular chat room.

Besides, if developer thinks the line of code with the yellow icon as shown in Figure 2 does not require access control check, he could choose the third option in the options list to remove the annotation request as shown in Figure 8, and then the yellow icon will disappear as shown in Figure 9.

```
50 if (!$chat_sid = chat_login_user($chat->id, 'sockets', $groupid, $course)) {
51     print_error('cantlogin');
52 }
```

Figure 9: After developer chose to remove the annotation request, the annotation request marker disappears.

This example demonstrates key components of the interactive static analysis approach. First, vulnerability detection and mitigation is aimed at the developer, with the goal of providing assistance to help developers with this task. The notifications also provide developers in-situ reminders of the security implications of their code. Static analysis utilizes application specific access control knowledge requested from the developer, to aid in the detection of vulnerabilities. Together, this approach could potentially reduce the cost of vulnerability detection through earlier detection and reduced demand on software security experts.

4.2 Overview of Prototype Implementation ASIDE-PHP

I implemented an interactive static analysis prototype as a plug-in for Eclipse IDE for PHP, called Application Security in IDE for PHP (ASIDE-PHP), which constantly scans source code in the workspace. The plug-in is based on Eclipse framework, which provides the infrastructure facilitating the transformation of source code into Abstract Syntax Tree (AST), AST manipulation libraries as well as interface and interaction libraries. Figure 10 provides an overview of major components of ASIDE-PHP, which takes as input: (a) Abstract Syntax Trees (ASTs) of the application generated by Eclipse PDT [19], and (b) a set of security sensitive operations (SSOs) specified by the Software Security Group (SSG). ASIDE-PHP then uses these to generate annotation requests to the developer. With the annotations provided by the developer, static analysis is used to detect vulnerabilities.

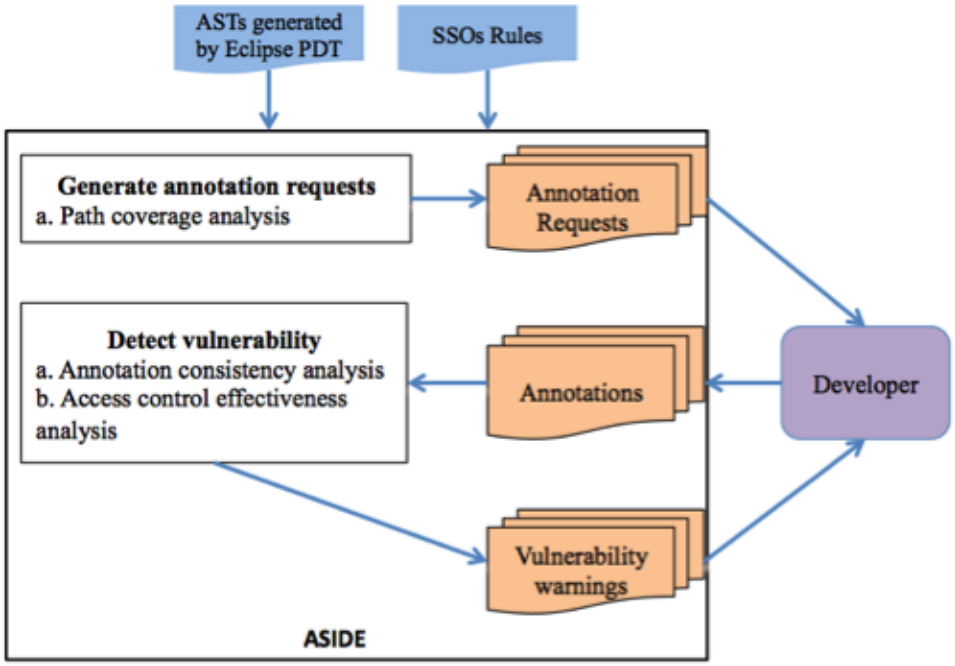

Figure 10: ASIDE-PHP architecture.

Related auto detection approaches have avoided studying access control related to SELECT operations for database tables [81, 70], because they may lead to overwhelming false positives. Applications often read sensitive information from tables without the need of performing access control checks. For example, a password may be read from a database table as part of the login process without access control checks. I overcome this limitation by considering sensitive SELECT operations only when sensitive data retrieved from the SELECT operation flows to a page-displaying API, for example echo() or print() in PHP. Frequently used page-displaying APIs for major web application development platforms are well known. They are often represented as default sinks by commercial static analysis tools to detect cross-site scripting vulnerabilities. If, instead of being displayed, a sensitive information read impacts other SSOs, access control checks would be performed for these SSOs.

ASIDE-PHP then performs path coverage analysis to generate annotation requests, which are displayed to a developer as shown in Figure 2, details are described in Section 4.5. The developer could interact with these requests and make annotations through ASIDE-PHP, as shown in Figure 5. Based on new annotations, ASIDE-PHP performs static analysis to detect vulnerabilities and display warnings to developers. Static analysis can be performed in a background thread, and should not block the developer's interaction with the IDE. While the current implementation is not yet optimized for efficiency, I believe these techniques can be efficiently implemented because they fit nicely with multi-core computer architecture.

4.3 ASTs Generated by Eclipse PDT

Based on Eclipse PHP Development Tools (PDT) [19], the plug-in ASIDE-PHP could obtain information about the projects opened in the Eclipse workspace. Relevant information used by ASIDE-PHP include: source packages in a project as illustrated in Figure 11, and what is the code in a specific line of a chosen source file. Eclipse PDT provides APIs to access and manipulate a source file as an AST. Code elements, such as function declaration, statement, function call, Boolean expression, are represented as an AST node in the AST, for example Figure 12 shows an AST illustration of a function declaration, all code elements in a function declaration could be accessed based on Eclipse PDT APIs. Each time an annotation is made on a code snippet by the developer, the annotation will be kept with that code snippet as auxiliary information, so it is identifiable.

4.4 SSOs Rules

A set of security sensitive operations (SSOs) could be specified by the Software Security Group (SSG) in the format of XML in ASIDE-PHP. The SSOs rules include two

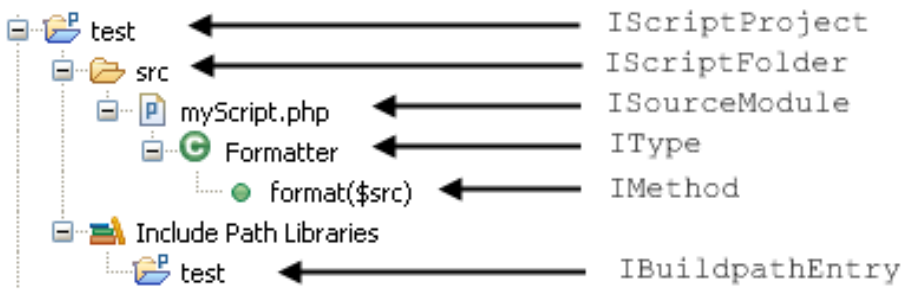

Figure 11: Illustration of project structure in Eclipse PDT [24].

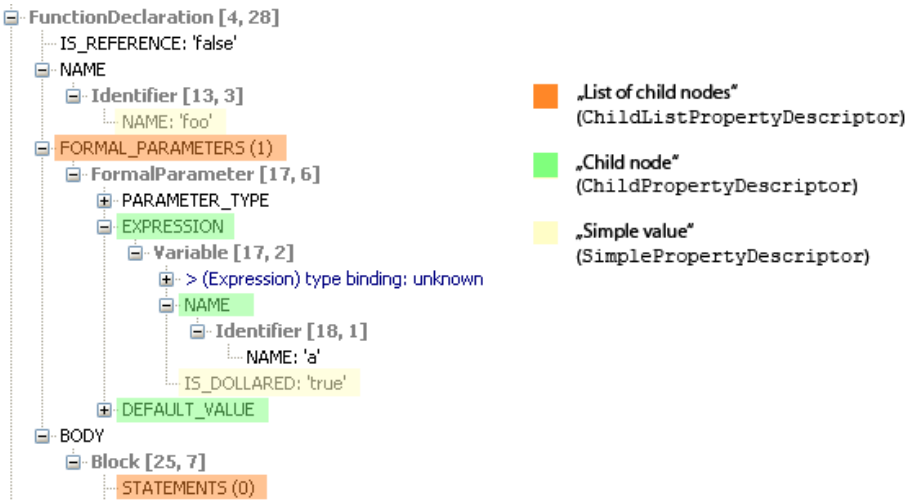

Figure 12: AST illustration of a function declaration in Eclipse PDT [24].

parts: abstract SSOs rules and concrete database operation APIs. Abstract SSOs rules define the security sensitive database operations in a conceptual and programming-language-independent way. For example, ”insert operation on table account” is an abstract SSO rule, security specialist needs to write a rule shown below. Every <sso></sso>pair represents an abstract SSO rule. The tag operationType specifies the database operation type, which could be insert, delete, update or select, and the tag tableName specifies the name of the table on which the database operation runs against.

<sso>

```
<operationType>insert</operationType>
<tableName>account</tableName>
</sso>
```

Concrete database operation APIs define the database operation APIs existing in specific programming languages. For example, *mysql_query* is a concrete database API in PHP programming language. There are two types of database operation APIs, parameterized database operation APIs and unparameterized database operation APIs. For example, in project Moodle [71], it has database operation API *insert_record*, which is a parameterized database operation API. Its usage is shown below, table name 'account' is explicitly specified in its first parameter as a constant.

```
$DB->insert_record('account', $account);
```

To specify the parameterized database operation API *insert_record*, security specialist needs to write a rule shown below. The tag id denotes the name of the API; the tag type indicates the database operation type of the API, for parameterized database operation API, allowed type include select, delete, update, and insert, in this example type is database "insert" operation; the tag tableParam specifies which parameter of the API represents table name, 0 represents the first parameter; the tag language represents the programming language to which the API belongs. In the example, *insert_record* is an API in PHP.

```
<DBOperation id="insert_record" type="insert", language="PHP
    ">
```

```
<tableParam>0</tableParam>
</DBOperation>
```

For unparameterized database operation APIs, which refer to cases where the API takes a SQL string as its parameter and the SQL string contains information about table name and operation type. For example, below is a unparameterized database SQL query example, the operation type is ”update” and table name is ”account”.

```
mysql_query(”update␣account␣set␣password␣=␣’$pwd’␣where␣
    userid␣=␣’$the_userid’”, $db);
```

To specify the unparameterized database operation API *mysql_query*, security specialist needs to write a SSO rule shown below. The tag id denotes the name of the API; type ”all” indicates that this API could be database SELECT, DELETE, UPDATE, or INSERT, its operation type is specified in the SQL string which stays as a parameter of the API; the tag sqlStringParam specifies which parameter of the API represents the SQL string, 0 represents the first parameter; the tag language represents the programming language to which the API belongs, in the example, *mysql_query* is an API in PHP.

```
<DBOperation id=”mysql_query” type=”all”, language=”PHP”>
<sqlStringParam>0</sqlStringParam>
</DBOperation>
```

Listing 5 shows a complete example of SSOs rules, which includes four abstract SSOs rules as shown in the four pairs of <sso></sso>, one unparameterized database operation

API as shown between the pair of <unparameterizedDBOperations></unparameterizedDBOperations>, and four parameterized database operation APIs shown between the pair of <parameterizedDBOperations></unparameterizedDBOperations>.

Listing 5: Example SSOs Rules

```
<ssos>
<unparameterizedDBOperations>
<DBOperation id="mysql_query" type="all", language="PHP">
<sqlStringParam>0</sqlStringParam>
</DBOperation>
</unparameterizedDBOperations>
<parameterizedDBOperations>
<DBOperation id="insert_record" type="insert", language="PHP
    ">
<tableParam>0</tableParam>
</DBOperation>
<DBOperation id="delete_record" type="delete", language="PHP
    ">
<tableParam>0</tableParam>
</DBOperation>
<DBOperation id="update_record" type="update", language="PHP
    ">
<tableParam>0</tableParam>
```

```
</DBOperation>
<DBOperation id="select_record" type="select", language="PHP
    ">
<tableParam>0</tableParam>
</DBOperation>
</parameterizedDBOperations>
<sso>
<operationType>insert</operationType>
<tableName>account</tableName>
</sso>
<sso>
<operationType>delete</operationType>
<tableName>account</tableName>
</sso>
<sso>
<operationType>update</operationType>
<tableName>account</tableName>
</sso>
<sso>
<operationType>select</operationType>
<tableName>account</tableName>
</sso>
```

</ssos>

4.5 Interactive Annotation

Web applications differ from other applications in that they do not have a unique program entry point, or ”main program”. One can identify program entry points for web applications. For example, I distinguish two types of files with .PHP extensions. First, a file may contain only class and/or function definitions. Second, a file may contain executable code outside of function and class definitions. The second type of file is referred to as a program entry as a URL can invoke it.

ASIDE-PHP requests interactive annotation at the program entry level because it most likely corresponds to an application use case. Access control policies are often defined for use cases. Placement of an annotation request at this level makes it easier for a developer to identify intended access control code. ASIDE-PHP expects annotations for access control logic to consist of statements that could divert the execution path away from performing the SSO. For example, in Listing 3, method invocation *require_capability(”capability_name”)* throws an exception if the capability indicated by *capability_name* is not satisfied. More specifically, annotated code should satisfy several requirements: (1) it must be on the execution path from the program entry point to the SSO; (2) it must consist of either (2.a) a set of Boolean expressions in a branch/conditional statement that lead to altering the execution path leading to the SSO involved, or (2.b) method invocations that could either throw exceptions or terminate the execution.

ASIDE-PHP enumerates all the execution paths from every program entry point to every SSO by computing program call graphs. This process is referred to as path coverage anal-

ysis because it ensures that for every execution path thus identified, there must be at least one annotation for an access control check ”covering” this path. ASIDE-PHP generates an annotation request for the developer for each path without an annotation. For example, in Figure 2, there exists an execution path from the program entry point index.php to the SSO INSERT to *chat_messages*, via function call *chat_login_user($chat->id, 'sockets', $groupid, $course)* on line 50. ASIDE-PHP thus generates an annotation request for line 50, shown as a yellow notification beside the code line.

4.6 Vulnerability Detection

Vulnerability detection in ASIDE-PHP is based on three mechanisms. First, the interactive annotation process encourages developers to self-review the access control logic and detect errors. This is particularly effective to help discover cases of missing access control checks. The second mechanism is based on the observation that the same SSOs normally require the same access control checks. The third mechanism is based on the observation that the annotated control checks should be (access) control effective, meaning they rely on trusted data for control decisions, rather than untrusted data or uninitialized variables.

4.6.1 Annotation Consistency Analysis

Differences in access control checks for the same SSOs often indicate a mistake. I adapted a method proposed by Son et al. [82] to determine whether two annotations match. I briefly summarize the steps of this process below and explain it through an example.

1. Obtain the calling context of an annotated check, using program slicing [91], containing all statements to which the annotated check has data dependency. Listings 6, 7, 8 shows slices of code with annotations for control checks represented in italics.

2. Given the annotated check and its calling context, extract the access control template (ACT) of the annotated check, an abstract representation of the access control check along with its data dependencies for determining matches.

3. Given two ACTs, ACT_a and ACT_b, for each statement in ACT_a, check if there exists only one statement in ACT_b that matches it, and vice versa. The order of statements does not matter, because I consider their data dependency as part of the matching. Two statements match iff (a) their AST structures and operators are isomorphic, and (b) their data dependencies also match.

Consider the examples in Listings 6, 7, 8. The code in Listing 6 matches code in Listing 8 because the annotated access control checks refer to the same function call *has_capability()*. The first parameters in both instances match as both refer to the same constant. The second parameters are variables but both can be traced to the same constant "admincontext". On the other hand, Listing 7 and Listing 8 do not match because the second parameters are traced to different values.

Listing 6: This access control check matches c but not b

```
$context = "admincontext";
 //annotated check
if (has_capability('moodle/blog:create', $context)) {
$DB->insert_record('blog'); //SSO
}
```

Listing 7: This check does not match with either a or c

```
$context = "studentcontext";
//annotated check
if (has_capability('moodle/blog:create', $context)) {
$DB->insert_record('blog'); //SSO
}
```

Listing 8: This access control check matches a but not b

```
$newcontext = "admincontext";
//annotated check
if (has_capability('moodle/blog:create', $newcontext)) {
$DB->insert_record('blog'); //SSO
}
```

4.6.2 Access Control Effectiveness Analysis

Access control decisions must be made based on trusted data such as a constant or data validated by the application (e.g. values in a web session). I use an example from SCARF to motivate my discussions. In Listing 9, the developer may annotate the Boolean expression *WHERE session_id='$id'* as a check for the SSO DELETE FROM sessions. ASIDE-PHP analyzes the annotated expression *session_id='$id'*, and identifies it is constraining the *session_id* field with a variable *$id*. ASIDE-PHP performs data flow analysis for *$id* and finds it is from a common untrusted data source *$_GET*, and therefore ASIDE-PHP regards the annotated access control check to have no access control effectiveness, and reports it as

an access control vulnerability.

Listing 9: editsession.php in SCARF

```
$id=(int)$_GET['session_id']; //untrusted data
query("DELETE FROM sessions WHERE session_id='$id'"); //SSO
```

Given an annotated access control check, ASIDE-PHP performs taint propagation data flow analysis. I use default taint sources in commercial static analysis tools (e.g. *$_GET*). ASIDE-PHP generates a vulnerability warning if any variable used in an annotated access control check can be traced to a taint source.

CHAPTER 5: EVALUATION OF INTERACTIVE STATIC ANALYSIS FOR ACCESS CONTROL VULNERABILITY DETECTION

I evaluated ASIDE-PHP's effectiveness using six open source PHP applications. Five of the six applications, Mybloggie, SCARF, Bilboblog, Wheatblog, and PhpStat, were used by auto detection research. They are summarized in Table 4. References next to each project indicate the papers in which a given project has been used as part of their evaluations. I thus use them to compare ASIDE-PHP's performance. In addition I chose Moodle to find out how ASIDE-PHP performs in a large-scale complex application. Moodle is an open source e-learning platform with thirteen stable releases and over 73 million users across 273 countries [23]. Moodle has over 625,000 LOC and 2,000 PHP files.

I evaluated ASIDE-PHP against these projects with the following research questions: (a) How would ASIDE-PHP impact the developer's work flow? (b) How effective is ASIDE-PHP at mitigating known vulnerabilities? (c) What is the number of false positives? (d) How does ASIDE-PHP's ability to detect vulnerabilities compare with related work? (e) What are the limitations of ASIDE-PHP in detecting access control vulnerabilities?

5.1 Evaluation Setup

Projects Mybloggie, SCARF, Bilboblog, Wheatblog, and PhpStat have relatively simple functions so I performed manual code review to identify security sensitive operations as shown in Table 5. I could not perform a thorough code review of Moodle because of its size. Moodle has 199 documented vulnerabilities for all versions. I selected Moodle version

Table 4: Projects used in evaluation.

Project	LOC	Description
Mybloggie 2.1.3 [70]	8874	Blogging system
SCARF 1.0 [70]	1318	Conference discussion forum for papers
Wheatblog 1.1 [81]	4032	Blogging system
Bilboblog 0.2.1 [37]	2000	Blogging system
PhpStat 1.5 [37]	12,700	Application presenting IM statistics
Moodle [43]	625,000	Course management

Table 5: Security sensitive operations identified in projects Mybloggie, SCARF, Bilboblog, Wheatblog, and PhpStat.

Project	SSOs
Mybloggie 2.1.3	Insert: POST_TBL, COMMENT_TBL, CAT_TBL, USER_TBL Update: POST_TBL, COMMENT_TBL, CAT_TBL, USER_TBL Delete: POST_TBL, COMMENT_TBL, CAT_TBL, USER_TBL
SCARF 1.0	Insert: comments, papers, files, authors, sessions, users, options Update: comments, papers, files, sessions, users, options Delete: comments, files, authors, sessions, users
Bilboblog 0.2.1	Insert: article Update: article Delete: article
Wheatblog 1.1	Insert: tblComments, tblCategories, tblLinks, tblPosts, tblUsers Update: tblPosts, tblUsers, tblLinks, tblCategories, tblComments, wbtbl_settings Delete: tblPosts, tblUsers, tblLinks Select: tblUsers, wbtbl_settings
PhpStat 1.5	Insert: stat_file, stat_alias, stat_user, stat_time, stat_wordcount Update: stat_file, stat_user, stat_alias Delete: stat_file, stat_alias, stat_user

2.1.0 released in 2011 because it contains the largest number (19) of known access control vulnerabilities in any stable release. Based on bug fix logs, I determined that 13 of them were connected with access control vulnerabilities for database tables: CVE-2013-2242, 2012-4408, 2012-3392, 2012-0797, 2012-2356, 2012-2367, 2012-3397, 2012-2354, 2012-2355, 2012-2358, 2012-3391, 2012-2359, and 2012-5473. Three vulnerabilities, CVE-2011-4293, 2012-4407 and 2012-3390, were access control vulnerabilities related to file access. I was not able to determine the source of failed access control in the remaining three cases: CVE-2012-0798, 2011-4309, and 2011-4303. Since my prototype implementation is targeting sensitive operations on database tables, I focused my efforts on examining the 13 CVEs connected with database tables. My approach can be extended to cover many file accesses as well. Read and write operations on sensitive files could be considered as

SSOs. It is straightforward to specify file paths of the sensitive files and the file read and write APIs. For example, ”$sensitiveFile = fopen(”customers”, ”w”); fwrite($sensitiveFile, $txt);” writes $txt into the file specified by the sensitiveFilePath, so we can specify an SSO: fwrite into file customers. This approach will work as long as file names can be determined via data flow analysis.

Thirty-one database table operations are connected with the 13 selected CVEs, as shown in Table 6. Because these 13 CVEs are classified as access control vulnerabilities, I assume that the 31 database table operations are SSOs. I ran ASIDE-PHP against the Moodle 2.1.0 code base with the 31 identified SSOs. I simulated a developer by making interactive annotations of access control checks responding to ASIDE-PHP requests. This effort spanned 70 files with an average of 223 lines of code per file. I used the following types of clues in identifying access control checks: (a) Developer comments, (b) Function and variable names, (c) Reported security fixes to access control vulnerabilities, and (d) Context and flow of the program.

5.2 Vulnerability Detection Results

Table 7 summarizes vulnerabilities discovered by ASIDE-PHP across all six projects. The column ”known vulnerability” includes reports from other approaches discussed in Chapter 3 and the 13 known access control vulnerabilities from Moodle. In the table, [M] represents missing checks, [I] represents ”inconsistent”, [UT] represents ”untrusted data”, and [L] stands for ”other logic error”.

Except for 7 logic errors in Moodle, ASIDE-PHP was able to discover all 26 known vulnerabilities. I will discuss these 7 logic errors in a later section. ASIDE-PHP discovered 20 zero day access control vulnerabilities. Listing 10 shows a zero-day example where a

Table 6: Known Moodle vulnerabilities and associated security sensitive operations.

Brief Description	**SSOs**
Unauthorized chat room access (CVE-2013-2242)	Insert: chat_messages, chat_messages_current, chat_users
Unauthorized reset (CVE-2012-4408)	Delete: event, role_capabilities, role_assignments, groupings, comments, grade_outcomes, grade_settings, course_completion_crit_compl, groups_members, groupings_groups, assignment Update: event
Unauthorized forum deletion (CVE-2012-3392)	Delete: forum_subscriptions
Deleted user can maintain access (CVE-2012-0797)	Insert: external_tokens
Unauthorized access to question-bank (CVE-2012-2356)	Insert: question Update: question
Unauthorized addition of calendar event (CVE-2012-2367)	Insert: event Update: event
Unauthorized access to group information (CVE-2012-3397)	Select: groups
Unauthorized reading of messages (CVE-2012-2354)	Select: message, message_read
Unauthorized addition of quiz questions (CVE-2012-2355)	Insert: quiz_question_instances
Unauthorized database modifications (CVE-2012-2358)	Insert: data_records, data_content Update: data_records
Unauthorized read of forums (CVE-2012-3391)	Select: forum
Privilege escalation (CVE-2012-2359)	Insert: role_capabilities Update: role_capabilities
Unauthorized information access (CVE-2012-5473)	Select: user

Table 7: Vulnerability detection results.

Project	**Known Vul.**	**Known Vul. by ASIDE-PHP**	**0-day Vul. by ASIDE-PHP**
Moodle 2.1.0	6[I], 7[L]	6[I]	1[I]
Mybloggie 2.1.3	3[M], 3[UT]	3[M], 3[UT]	15[M]
SCARF 1.0	1[M], 10[UT]	1[M], 10[UT]	
Bilboblog 0.2.1	1[UT]	1[UT]	
Wheatblog 1.1	1[M]	1[M]	
PhpStat 1.5	1[M]	1[M]	4[M]

developer created a comment ”Added security”, from which I can infer that the developer intended to add a control check for the SSO, but wrote it incorrectly. With ASIDE-PHP, this issue will be detected right at the point of annotation. Once the developer annotates the expression *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* as access control. ASIDE-PHP will identify that this is an invalid check because function echo() does not generate an exception. This check will not alter the execution path leading to the SSO. ASIDE-PHP will notify the developer to write a valid check so that this vulnerability could be mitigated.

Listing 10: Vulnerable code in del.php in Mybloggie 2.1.3

```
// Added security
if(!isset($_SESSION['username']) &&
        !isset($_SESSION['passwd'])) {
      echo"<metahttp-equiv=\"Refresh\" content=\"2;url=".
self_url()."/login.php\" />";
}
sql_query("DELETE FROM " .POST_TBL."
WHERE post_id='$post_id'"); //SSO
```

Listing 11 illustrates another zero-day finding confirmed and patched by the Moodle development team (CVE-2014-0122). The SSOs involved are INSERT into database tables *chat_messages*, and *chat_messages_current*, the same operations as involved in the example in Listing 3. The vulnerable code is in file "chat_ajax.php". It was discovered by comparing this code instance with the code instance shown in Listing 3.

Listing 11: Vulnerable code in chat_ajax.php in Moodle 2.1.0

```
if (!$chatuser = $DB->get_record('chat_users', array('sid'=>$chat_sid))) {
throw new moodle_exception('notlogged', 'chat');
}
if (!isloggedin()) {
    throw new moodle_exception('notlogged', 'chat');
}
$DB->insert_record('chat_messages', $message);
```

```
$DB->insert_record('chat_messages_current', $message);
```

Moodle requires that only logged in users with access to a particular chat room can insert chat messages. Moodle maintains a temporary table of users currently in chat rooms. The vulnerable code checks the temporary table as a proxy for the fact that a given user has the appropriate chat authorization. It did not account for the case where a user's chat privilege may be revoked after he entered the chat room. Besides the two zero-day examples I just discussed here in this section, the other 18 zero-day findings have been listed out in appendix A.

In addition, using the same technique, ASIDE-PHP found two other zero day Cross-site Request Forgery vulnerabilities in Moodle 2.1.0 affecting the most recent release of Moodle. They have been confirmed and patched by the Moodle development team (CVE-2014-0010, CVE-2014-0126).

5.3 False Positives for Vulnerability Detection

False positive warnings arise when ASIDE incorrectly alerts the developer of a potential vulnerability. We found false positives resulting from vulnerability detection based on annotation inconsistency. Table 8 summarizes the false positives for warnings generated by annotation inconsistency. For example, in Moodle 2.1.0, 19 warnings were generated because there exist 19 sets of inconsistencies where different access control checks were annotated for the same SSO. Among them, 12 are true positives leading to the discovery of 7 vulnerabilities, including a zero-day. Note that ASIDE may generate multiple warnings per vulnerability. This is because multiple sensitive operations are often grouped together. It is possible to have more than one form of valid access control check for the same SSO

Table 8: False positives for warnings.

Project	**Total warnings generated by inconsistency**	**False positives for warnings**	**Why false positives?**
Moodle 2.1.0	19	7	Group difference, Context difference
Mybloggie 2.1.3	1	0	N/A
SCARF 1.0	0	0	N/A
Bilboblog 0.2.1	0	0	N/A
Wheatblog 1.1	1	1	Context difference
PhpStat 1.5	1	1	Context difference

and it is impossible to detect semantic equivalency in general. An SSO may appear to have multiple forms of associated access control checks for several reasons. First, SSOs often occur in groups. For example in Moodle, insertions into the following tables often occur together: *chat_messages*, *chat_messages_current*, *chat_users*. Two such groups may require different access control authorities even when they share some common SSOs. Therefore, from the perspective of a given SSO, it has different access control checks when it belongs to different groups.

Second, a given access control policy may have semantically equivalent but syntactically different implementations. The third cause is due to application context difference as illustrated in Listing 12 and Listing 13. In this example both function calls *role_change_permission()* in Listing 12 and *reset_role_capabilities()* in Listing 13 insert information into the security sensitive table *role_capabilities*. Thus, they both have the same SSO. However, changing role permission requires the ”review role” whereas resetting role capabilities requires the ”manage role” as indicated in the code. I found similar situations in Wheatblog and PhpStat where the same SSO may use a different access control check depending on either an administrator context or a user context.

Listing 12: Code snippet from /moodle210/admin/roles/permissions.php

require_login($course, false, $cm);

Table 9: Total number of annotations requested.

Project	Total annotations requests	Annotations per file
Moodle 2.1.0	186	(0.093, 2.67)
Mybloggie 2.1.3	21	0.32
SCARF 1.0	27	1.42
Bilboblog 0.2.1	2	0.08
Wheatblog 1.1	24	0.49
PhpStat 1.5	30	2

```
require_capability('moodle/role:review', $context);
role_change_permission($roleid, $context, $capability->name,
    CAP_PREVENT);
```

Listing 13: Code snippet from /moodle210/admin/roles/manage.php

```
require_login();
require_capability('moodle/role:manage', $systemcontext);
reset_role_capabilities($roleid);
```

5.4 Impact on Developers

Developer adoption is critical for the success of interactive static analysis. One way to measure developer usability is to look at interactive annotation requests per file because it diverts a developer's attention from writing code. In an IDE environment a file is always the focus of the developer's attention. Table 9 summarizes annotations requests for all six projects.

Results from Moodle only have an estimated range because I estimated the set of SSOs using a process described in Section 5.1. A total of 186 annotation requested were generated based on identified SSOs for Moodle. Average annotation request per file is estimated between 0.093 and 2.67. Many Moodle files will not contain any SSOs at all. The lower

Table 10: False positives for annotation requests.

Project	Total annotations requests	False requests	Why false positives for requests?
Moodle 2.1.0	186	20	install
Mybloggie 2.1.3	21	3	install
SCARF 1.0	27	3	install, registration, password recovery
Bilboblog 0.2.1	2	0	N/A
Wheatblog 1.1	24	3	registration, login
PhpStat 1.5	30	0	N/A

number is based on total files in Moodle while the higher number is based on files containing at least one SSO. For the other five projects I was able to get a more precise estimate of the number of annotation requests because I have identified all the SSOs through code review. The data suggest that a developer will receive no more than 2 annotation requests per file.

A false positive in annotation requests (aka. False requests), arises when there is no access control check needed. An example for such a situation is code for system installation, during which security sensitive database tables are initialized. Such a situation is different from online system maintenance functions, which require appropriate authorization checks. System installation often is run in a special setting before the system is brought online. In such a mode explicit access control checks in the source code are not necessary. Such false positives are generated because it is difficult for ASIDE-PHP to differentiate online system maintenance functions and offline system initialization functions as both of them are contained in files with .PHP extensions. Excluding the small group of PHP files performing installation functions from the interactive static analysis process can easily eliminate these false requests. Another example of false requests involves user registration, login and password recovery functions where sensitive tables are accessed but no access control checks are needed. Table 10 summarizes my false positive findings.

Overall, ASIDE-PHP generated 290 requests for annotation for all six projects. I iden-

tified 29 of these requests as unnecessary requests. 23 out of the 29 false requests belong to code performing system installation. One can eliminate these requests quite easily by excluding the group of PHP files performing installation from the interactive static analysis process. For example in Moodle, six of the 70 files I studied are dedicated to installation. Five out of the 29 false requests occurs in login, registration and password recovery.

Automatic Annotation could be used to reduce the number of annotation requests at a risk of somewhat reduced capacity on vulnerability detection. This may be a viable choice if minimizing developer interruption is a high priority.

In my evaluation of the six applications, I had two observations. First, given an access control check for an SSO, if the exact same code pattern is found associated with the same SSO in a different file, then there is a very high probability that this code pattern is the access control check in that second file. I thus could with high confidence make an automatic annotation.

Our second observation is that multiple SSOs in one file often share access control checks. For example, in Listing 14, SSO_1 and SSO_2 shares the same check *require_admin()*. Thus once a developer finishes making an access control check annotation for SSO_1, it is likely to be the access control check for all other SSOs on the execution path (e.g. SSO_2 in Listing 14), and thus ASIDE-PHP automatically annotates *require_admin()* as the check for SSO_2 in Listing 14 as well.

Listing 14: Example code of applying auto-annotation

```
require_admin(); //control check
if (isset($_GET['delete'])) {
```

Table 11: Annotations needed with or without auto-annotation.

Project	Annotations requests	Annotation requests per file	Annotation requests with auto annotation	Annotation requests per file with auto annotation
Moodle 2.1.0	186	0.093,2.67	48	0.024,0.69
Mybloggie 2.1.3	21	0.32	15	0.23
SCARF 1.0	27	1.42	11	0.58
Bilboblog 0.2.1	2	0.08	2	0.08
Wheatblog 1.1	24	0.49	17	0.37
PhpStat 1.5	30	2	4	0.27

```
query("DELETE FROM comments WHERE
comment_id='$comment_id'"); //SSO_1
} else if (isset($_GET['approve'])) {
query("UPDATE comments SET approved='1' WHERE comment_id='
    $comment_id'"); //SSO_2
}
```

Table 11 shows the results of reductions in annotation requests if automatic annotation heuristics were used. The average number of requests per file can be reduced to well below one. However, with auto annotation, I may not detect one of the known vulnerabilities discussed in Section 5.2, shown in Listing 4. I found that vulnerability by comparing two duplicates of access control checks for the same SSO. One of the duplicates is a proper subset of the other, the correct access control check. If the developer annotated the vulnerable check without noticing his mistake, the duplicated one would be incorrectly annotated automatically, thus missing the opportunity of finding this vulnerability. However, if the correct access control check had been annotated first, I would still be able to find the vulnerability.

5.5 Comparison with Auto Detection

ASIDE-PHP is able to detect all 26 vulnerabilities found by related work auto detection approaches. In addition ASIDE-PHP found 20 zero-day vulnerabilities that were missed by related work. This supports my claim that auto detection has significant limitations in detecting access control vulnerabilities because there are insufficient access control patterns for them to learn access control models for SSOs. ASIDE-PHP performs better because it relies on developer input to identify access control models.

None of the related work provided false negative analysis. My analysis suggests that they are likely to have high false negatives. For example, in Mybloggie and Wheatblog I found more zero-days than vulnerabilities reported by related work that used the same applications for evaluation. Moodle provides an additional source for false negative analysis as it has well documented vulnerability findings. Seven of the 13 Moodle vulnerabilities could not be discovered by any of the methods discussed in this chapter. I examined all 7 cases in detail. Each case can be attributed to subtle logic errors. In each case, there are no other access control patterns one can compare to. It is not surprising that logic errors are unavoidable in a large complex application. Thus, there is unlikely to ever be an automated method to identify all access control vulnerabilities. The best industry practice to discover logic errors is still through manual code reviews [44].

ASIDE-PHP can offer useful assistance to manual code reviews by saving annotations in the IDE. First, through interactive annotation, it encourages developers to self-review the code. They may be able to detect and quickly correct access control vulnerabilities even in cases where the access control code pattern is not repeated. Second, interactive annotation

can reduce the cost of manual code review by capturing developer's design rational and helping a reviewer to highlight access control concerns during code review. It can help reviewers zoom in on critical parts of the code quickly. It can also be incorporated into a team development environment where peer review of critical access control logic can be tracked to reduce human errors.

In addition to new projects, interactive static analysis can be applied effectively to projects with a large legacy code base as well. Developers will be requested to provide interactive annotations either upon addition of new code, or modifications of existing code. Such annotations can be used to trigger automatic annotation and detect vulnerabilities in the legacy code.

Interactive static analysis is not without challenges, including: (a) how to motivate developer adoption, and (b) how to deal with annotation errors, which may be unavoidable. Developer adoption should be addressed first by designing a tool that is easy to use and provides the least possible disruption to developer's normal workflow. For example, in Section 5.4 I illustrated that unnecessary developer interruption could be minimized. There is a strong incentive for organizations to adopt a low cost and effective secure programming tool, and influence their developers to use it.

While developer input is the most reliable source of application knowledge, human errors are unavoidable. Incorrect annotations may lead to false negatives. My prototype implementation does not contain any mechanism to account for human errors. A number of strategies may be used to detect human errors, including:

Identify omissions. I observed that access control checks for different SSOs often have shared parts. So if a term (e.g. a Boolean expression or function call) which has been part

of an annotation for an access control check for one SSO and is not part of an annotation for another SSO even though it is along an execution path from a program entry point, this may be pointed out as a possible annotation omission.

Peer review. An annotation may be reviewed by a second developer working on related pieces of code.

It is important to remember that vulnerabilities found by any tool ultimately are subject to the review of a developer. Final actions depend on the developer, the same people interactive analysis relies on for providing interactive annotations.

5.6 Comparison with Commercial Static Analysis Tools

Writing custom rules is the approach adopted by most commercial static analysis tools, such as Fortify SCA [25]. I take the code snippet shown in Listing 15 as an example and write a custom rule, in Fortify SCA, for its access control requirements. The security sensitive operation *$DB->insert_record()* requires the access control check *require_permission()* before its execution. To write an access control custom rule for it, one first needs to model this requirement as a finite state machine, as shown in Figure 13. Each box represents a state; each arrow represents a transition between two states triggered by certain code patterns. In this example, the program must be in the checkState state before the security sensitive operation can be carried out securely. BrokenAccess represents the state when a broken access control vulnerability occurs, upon which Fortify SCA will report a warning. With this model in mind, one can write a custom rule as illustrated in Listing 16 where one first defines all the states shown in Figure 13 and then defines the transitions between the states.

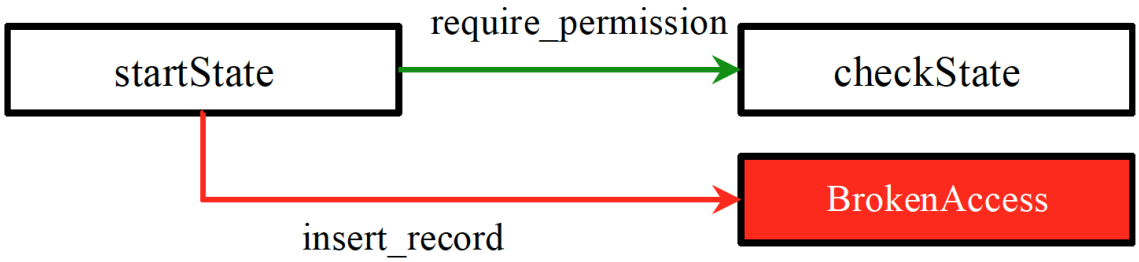

Figure 13: Finite state transition machine for the control flow.

Listing 15: Example code snippet with SSO and control check

```
require_permission();
$DB->insert_record();
```

This example illustrates the disadvantage of writing custom rules, in that it requires learning of new concepts and specification languages. Such training is unlikely to be widely available for average developers. This type of analysis is typically performed by software security experts collaborating with developers who would provide the required application specific knowledge. Such an arrangement contributes to the high cost of writing custom rules.

Listing 16: A custom control flow rule for Fortify SCA

```
<ControlflowRule formatVersion="3.2" language="PHP">
<VulnCategory>Broken Access Control</VulnCategory>
<!-- Function definitions for require_permission,
    insert_record are omitted -->
<Definition><![CDATA[
state startState(start);
```

```
state checkState;
state sensitiveState(error); <!-- BrokenAccess -->
startState -> checkState {require_permission()}
startState -> sensitiveState {insert_record()}]]>
</Definition>
</ControlflowRule>
```

CHAPTER 6: PRIVILEGE ESCALATION DETECTION USING APPLICATION SENSORS

Current intrusion detection systems (IDS) are primarily based on network traffic sensors. Network-based IDS are not aware of the application context and are unable to detect stealthy attacks on high value application targets. For example, a web-based application using https for transport may be subject to privilege escalation attacks. Network sensors cannot effectively detect such attacks or subsequent leakage of information. Privilege escalation attacks, however, can be detected by application sensors reporting events of failed access control checks. Most successful attacks do not succeed on the first try. Attackers typically go through a process of reconnaissance and exploration. Thus failed attempts provide valuable clues to detect on-going attacks and enable preemptive actions (e.g., revoked compromised credentials).

Intrusion detection based on application events has been used before in customized situations. For example, account lockout due to repeated failed login attempts is a type of application-based intrusion detection and prevention. IDS based on application sensors is different from Web Application Firewalls (WAF) [76] in that WAF treats the application as a black box, and thus will not be effective at detecting privilege escalation attacks. Application sensors are embedded within the application itself and can provide application context information. Watson et. al. [89] proposed to use application sensors to detect attacks through malicious input. They developed a framework where application sensors report

malicious input events to a centralized monitoring system where events can be correlated to detect attacks. However, no research has addressed how to automatically instrument applications with sensors for intrusion detection.

Relying on developers to manually insert application sensors is unlikely to lead to wide adoption. First, inserting sensors is extra effort that is not part of the application specification and that will add to the work load of the developers, both in terms of programming and testing to make sure the added sensor does not change application functions. Second, as part of normal application maintenance, application sensors must be updated and tested to make sure they are in sync with relevant application logic. I believe that developers should be focused on developing applications (and practice secure programming!). Instrumenting application sensors should be (a) done in an automated way, (b) without changing the intended functions of the application, and (c) having minimal performance impact.

For the rest of the chapter I describe an approach that can automatically instrument a web-based application to detect privilege escalation attacks. This is accomplished in two steps. First I describe an approach to identify candidate web pages to insert application sensors to detect failed attempts to access sensitive information. I evaluate this approach based on two open source applications. Second, I present a model, based on analysis of six open source applications, on how to automatically insert application sensors without impacting application logic.

6.1 Related Works

In this section, I will mainly survey related work in intrusion detection.

NIST [29] defines intrusion as an attempt to compromise the confidentiality, integrity, and availability, or bypass the security protection mechanism for a computer system or

network. Intrusion detection system (IDS) [29] is defined as the system monitoring and analyzing the events happening on the computer or network to detect signs of intrusions. IDSs detect intrusion mainly based on three methodologies: signature-based detection, anomaly-based (behavior-based) detection, and stateful protocol analysis (specification-based). The focus of the related work in this chapter is not to survey and compare literatures based on their detection methodologies, but to survey literature on IDSs based on where the IDSs are deployed and the types of events or activities toward which the detection methodologies are applied.

Based on where they are deployed and what events or activities they monitor and capture, IDS could be categorized into four types, host-based IDS, network-based IDS, wireless-based IDS, and mixed IDS. A host-based IDS monitors and captures suspicious host activities and host characteristics for severs running services, and hosts containing sensitive information [62]. A network-based IDS monitors and captures network traffic at specific network segments via sensors and analyzes them to detect suspicious incidents [62]. A wireless-based IDS is similar to network-based IDS, the only difference is that it monitors and captures wireless network traffic, such as wireless sensor network traffic, etc. Mixed IDS is a type of IDS that adopts multiple different types of IDS to achieve more accurate detection, for example, adopting network-based IDS and host-based IDS together.

There has been extensive research on host-based IDS and network-based IDS. Research works [28, 38, 39, 56, 72, 84, 59] are host-based IDSs, which monitor and capture host characteristics such as usage of disk and memory. Research works [28, 38, 39, 56, 72, 84, 59, 27, 35, 42, 45, 52, 53, 55, 61, 64, 69, 74, 88, 96, 78] are network-based IDSs, which monitor and capture network traffic, such as network packages and sequence of commands.

Although the four types of IDSs are useful in detecting some common intrusions, none of them are effective in detecting application specific attacks, because they are unable to captures the application context information. To achieve application layer intrusion detection, there have been very few research efforts. One way of application layer intrusion detection is through a Web application firewall (WAF) [76]. WAF acts as a filter and applies preconfigured rules to an HTTP conversation. It is a generic attack detection mechanism, it is able to detect common actions of a known attack sequence such as basic SQL injection or cross-site scripting attacks. However, because it has no insights about application specific traffic, it is unable to detect many application specific attacks, and thus it is not a sufficient prevention approach for critical high value applications, such as financial applications.

As another way of application layer intrusion detection, Watson et. al. [89] proposes to use application sensors to detect attacks. Their tool puts sensors inside the application and obtains application context information so that it is able to detect application specific attacks. OWASP AppSensor [90] provides a reference implementation of sensors to detect intrusions based on inputs, which requires developers to manually put those sensors code in places that need sensors. Requiring developers to manually write sensor code would prevent wide adoption. Little research has been performed toward how to instrument those application sensors. In this chapter, I discuss using application sensors to detect privilege escalation attacks, and focus on automatically instrumenting application sensors. I provide an approach on automatically placing application sensors via interactive static analysis.

6.2 Privilege Escalation Detection Based on Application Sensors

Privilege escalation is a common attack against high value applications such as financial management, human resource, and electronic medical records by exploiting application

vulnerabilities. Previous research efforts have been focused on secure coding [51, 98, 93, 34] to reduce vulnerabilities. However it is impossible to eliminate software vulnerabilities as no software of any complexity can be expected to be bug free. My focus is to place sensors inside an application to report failed attempts to gain access to protected resources. Such events are good indicators for a privilege escalation attack. In the physical world, this is analogous to a security guard identifying a suspicious individual attempting to gain access to a high security building. Application-based intrusion prevention has a richer range of responses than a typical network counterpart, e.g. account lockout and process termination, to stop the attacker before he succeeds in finding an exploitable vulnerability. This is because the application sensors can provide detailed application context, such as the login credential of a potential attacker.

However not all access failures reliably signal attacks. For the rest of discussion, I assume the application is web-based. In a web-based application a typical user navigates application functions by following links and menu items. For example an application may require users to login before writing comments. However, a comment link is always available on web pages even if the user has not logged in. Clicking on the comment link will lead to an access control failure but it is not indicative of an attack.

Forced browsing is defined as an action where a subject visits a web page without following available links and menu items. Forced browsing is often associated with privilege escalation attacks [85], because the attacker is actively looking for an access control vulnerability. My approach is to place application sensors to detect forced browsing events to detect privilege escalation attacks.

It should be pointed out that there are scenarios where an innocent user may exhibit

forced browsing behavior. The most common case is a user refreshing a page after session expiration. False alarms are unavoidable in IDS systems and can be minimized by considering context information. For example malicious forced browsing differs from innocent cases in that an attacker often attempts forced browsing on multiple pages over a short period of time. The focus of this chapter is automatic placement of application sensors. I leave improving accuracy of intrusion detection for future work.

The focus of this research is to insert application sensors into web-based applications to detect forced browsing events. I assume the application is developed in an environment where secure software development is a priority in that serious efforts are made to minimize software vulnerabilities. By relying on interactive static analysis proposed in Chapter 3, a reliable set of access control checks are captured through interactive code annotation. I use an example from the open source course management application Moodle [71] to illustrate key concepts. A Moodle chat room is created by a teacher for a specific course. Only logged-in users with required chat capability should send messages to the chat room. Listing 17 shows a code snippet from Moodle. The example involves two Moodle database SSOs: INSERT to *chat_messages* and *chat_message_current*.

Listing 17: Example code snippets from Moodle. Access control checks are shown in italics

```
require_login($course, false, $cm);
if (isguestuser()) {
print_error('noguests','chat');
}
```

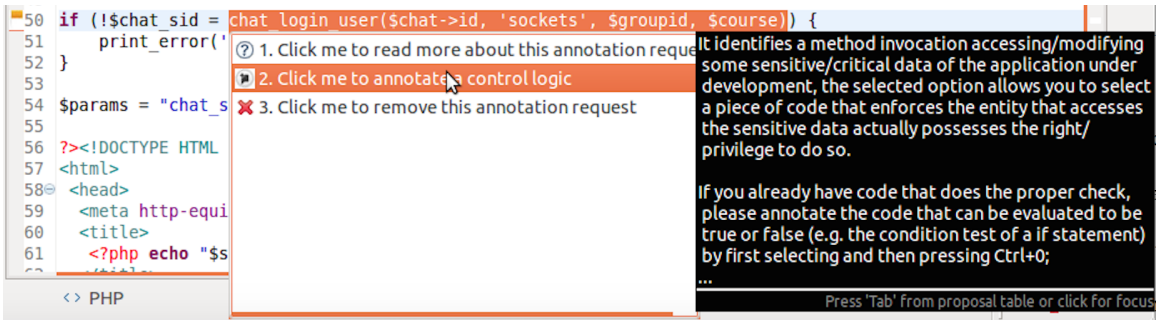

Figure 14: Explanation for interactive annotation.

```
if (!$chat_sid = chat_login_user($chat->id, 'sockets',
$groupid, $course)) {
print_error('cantlogin');
}
```

The developer is prompted at the line of code where SSOs are called, as shown on line 50 in Figure 14. Following the instruction, the developer performs the annotation by highlighting the code containing the intended access control logic as shown in Figure 15 (lines 27 and 29 in green). Through this process, a reliable set of access control checks are captured via point-and-click interaction, which is easy for developers to use.

6.2.1 Determining Candidate Webpages to Place Application Sensors

Figure 16 shows part of a sitemap for Wheatblog, an open source PHP application. A sitemap shows possible navigation paths by following links inside web pages. It is a directed graph where each node is a web page. A link from page C to D is a conditional link if this link is displayed on page C pointing to D only if certain access control checks are satisfied. They are shown as dashed links in Figure 16. Conditional links are often found in an index page where links pointing to pages accessing sensitive information are

index.php

```
27 require_login($course, false, $cm);
28
29 if (isguestuser()) {
30     print_error('noguests', 'chat');
31 }
32
33 /// Check to see if groups are being used here
34  if ($groupmode = groups_get_activity_groupmode($cm)) {   // Groups are bein
35     if ($groupid = groups_get_activity_group($cm)) {
36         if (!$group = groups_get_group($groupid)) {
37             print_error('invalidgroupid');
38         }
39         $groupname = ': '.$group->name;
40     } else {
41         $groupname = ': '.get_string('allparticipants');
42     }
43 } else {
44     $groupid = 0;
45     $groupname = '';
46 }
47
48 $strchat = get_string('modulename', 'chat'); // must be before current_langu
49
50 if (!$chat_sid = chat_login_user($chat->id, 'sockets', $groupid, $course)) {
51     print_error('cantlogin');
52 }
```

Figure 15: Developer annotated access control checks, highlighted in green.

displayed after the user has been authorized. In other words web applications often ”hide” links to pages accessing sensitive information until the access control credential of the user is checked first. Because web-applications are multi-entrant, access control checks must be repeated in every page accessing sensitive information to prevent forced browsing attacks. An unconditional link from page A to page B means there is a web link displayed in page A referring to page B without the need to satisfy access control checks. They are shown as solid links in Figure 16.

Web applications are built with intended execution paths. Intended starting pages (typically files named index.php in various directories), are assumed to be provided as input to constructing the sitemap [85]. For example, in Figure 16, index.php and admin/settings.php

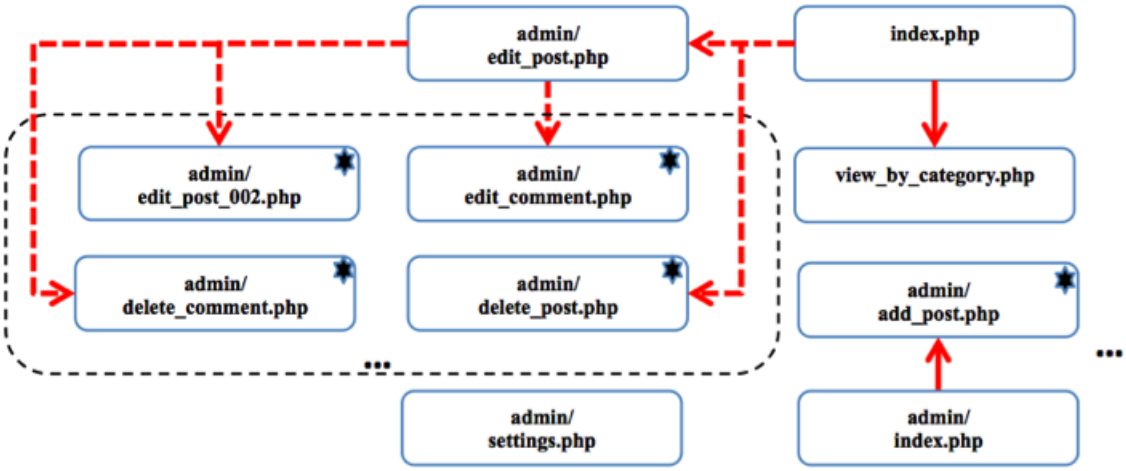

Figure 16: Part of the sitemap of Wheatblog (some pages are omitted as ellipsis due to page limit).

are entry pages. The dashed box in Figure 16 indicates a set of pages that are candidates for placing application sensors for forced browsing. This is because each page has at least one SSO and it cannot be reached from an entry point via a path consisting of unconditioned links. That is to say, links to pages in the dashed box are protected by access control logic. I will describe how to identify conditional links in the next subsection. Therefore access control failure reports by application sensors in these pages are good indications of forced browsing.

The whole process of identifying candidate pages for sensor placement consists of the following steps. The first step is to construct a rough site map by extracting links from the source files. I call it a "rough" sitemap because the sitemap at this stage does not distinguish conditional vs. unconditional links. Second, identify conditional links that are displayed based on some access control conditions. Third, identify candidate pages to place sensors. Input to this process includes: (a) a set of SSOs, (b) a set of access control checks for each SSO, (c) all web page source files (.php files), and (d) a set of entry web pages.

6.2.1.1 Rough Sitemap Construction

A PHP file refers to a file with .php extension. A PHP file may contain function definitions as well as PHP executable program. Some PHP files contain function definitions only. Such a file is not a web page and its role is to define shared functions. A PHP web page is a PHP file with a PHP executable program that is not part of any function definition. Execution of a PHP web page starts with the first statement in the file that is not part of a function definition.

The most complex part of building the sitemap is extracting links out of web pages because finding the link target of an HTML output string is a non-trivial task. Based on PHP library function calls for output, such as 'echo' and 'print', I could locate the HTML output strings. For example, Listing 18 and Listing 19 show example code snippets displaying a web link. For both examples, I use regular expressions to match and locate the HTML anchor, form, frame tags, etc. Listing 18 represents a static link, pointing to editsession.php. Listing 19 represents a dynamic link, because $lang is a string variable. Identifying its target involves resolving string variable $lang, which is a complicate task. Sun et. al. have looked into this problem and have come up with an approach that can detect many types of dynamic links [85]. I leverage the link extraction algorithm proposed by [85] to obtain the link targets of all the links in the web page, details of the link extraction algorithm are described in [85]. The rest of the algorithm follows a fairly standard graph building process. It should be noted that the sitemap is represented as a multigraph as it is possible to have parallel links in one page pointing to another page.

Listing 18: Static link example from SCARF.

```
print "<a href='editsession.php'>(edit)</a>";
```

Listing 19: Dynamic link example.

```
print "<a href=".$lang."."php>Anchor</a>";
```

I use an example adapted from Wheatblog shown in Figure 16 to illustrate the algorithm. The first step of the sitemap construction is to put all known entry pages into a worklist queue. Then I pick a node from the worklist queue, suppose it is index.php. If the node has not been processed before, I use the approach by Sun et. al. [85] to extract all links from a PHP webpage. This process will return tuples (index.php, admin/edit_post.php) and (index.php, admin/delete_post.php). For page nodes with no links identified on it, tuples (nodes, _) will be returned, _ represents null. Tuples representing links to web pages outside the application are filtered. After the filtering step, these tuples will be added to the sitemap. Nodes admin/edit_post.php and admin/delete_post.php will be added to the worklist. I remove index.php from the worklist and add it to the visited set to prevent processing it again. This process continues until the worklist is empty.

6.2.1.2 Identify Conditional Links

As mentioned in the interactive static analysis in chapter 3, access control checks could be either Boolean expressions or function calls that could throw exceptions or terminate the execution. I consider the following types of Boolean expressions:

(a) Boolean expression in if-branch. E.g. if(expr), elseif(expr).

(b) Boolean expression in switch-branch. E.g. switch(expr) case expr1: ... ; the Boolean expression is expr == expr1.

Listing 20 shows an example code snippet in which Boolean expression *$_SESSION['loggedin'] != null* is the access control check. Listing 21 shows an example in which a function call *require_loggedin()* is the access control check, and *require_loggedin()* is a function that may cause the execution to terminate. Some access control checks could be part of the WHERE clause in SQL statements. I did not include these cases in my proof of concept evaluation. Monshizadeh et al. [70] has developed an approach to work with access control code in SQL statements and that can be integrated with my approach.

Listing 20: Example code of Boolean expression as access control check shown in italic

```
if ($_SESSION['loggedin'] != null){
query("INSERT␣INTO␣comments␣");
}
```

Listing 21: Example of function call as access control check shown in italic

```
require_loggedin(); //may cause execution to terminate
query("INSERT␣INTO␣comments␣");
```

A PHP execution sequence can be changed by either particular PHP library function calls that terminate the execution of current program, such as *exit()* or *die()*, or by throwing of exceptions. I collectively refer to these function calls as function calls with abnormal return code. When considering exceptions, all subclasses of the PHP class Exception must be considered. For example, in Listing 22 function *print_error()* abnormally returns the execution by throwing a moodle exception, a developer-defined subclass of Exception.

Listing 22: print_error() is abnormal return code in Moodle.

```
function print_error($errorcode, $module = 'error', $link =
    '', $a = null, $debuginfo = null) {
throw new moodle_exception($errorcode, $module, $link, $a,
    $debuginfo);
}
```

Extract Security Critical Variables from access control checks

Ultimately access control checks rely on the values of the program variables involved in these checks to make the determination. For example in Listing 23 *$_SESSION['privilege'] == 'admin'* is an Boolean expression access control check. It relies on the value of the variable *$_SESSION['privilege']* to determine whether it returns TRUE or FALSE, which then determines whether the SSO will be executed.

Listing 23: Example of Boolean expression as access control check

```
if ($_SESSION['privilege'] == 'admin')  //annotated check
query("DELETE FROM sensitive_table ");  //SSO
```

A variable is regarded as involved in an access control check if the variable impacts the control flow leading to the execution of an SSO. Listing 24 shows an example of several annotated access control checks for one SSO, two of them are Boolean expressions, *$_SESSION['user'] == null* and *$reviewable*, and one is function call *require_permission()* with abnormal return code. Listing 25 shows the definition of *require_permission()*. So by

analyzing the inter-procedural data and control flow related to the three checks, variables involved in them include: global variable *$reviewable* which is declared outside a function and can be accessed outside a function or accessed within a function with global keyword; super-global variable *$_SESSION['permission_level']* and *$_SESSION['user']* are always available in all scopes and can be accessed without using the keyword global; local variable *$permission_level* which is declared within a function and can be only accessed within the function.

Not all variables involved in access control checks serve security functions. Some of them may relate to application flow, for example, passing value from one variable to another variable. *$permission_level* is just passing the value from *$_SESSION['permission_level']* so that the value of *$_SESSION['permission_level']* could be compared with null or 0, shown as *$permission_level == null*, *$permission_level >0*. The variables hold the authorization or authentication state throughout the program are *$_SESSION['permission_level']*, *$_SESSION['user']* and *$reviewable*, which are regarded as access control related. Based on my studies of six open source PHP projects, it is observed that in PHP web applications authentication or authorization state information is usually held by untainted global or super-global variables, and access control decisions rely on values held by those global and super global variables. In the examples shown in Listings 23, 24, 25, SESSION state variables *$_SESSION['privilege']*, *$_SESSION['permission_level']*, and *$_SESSION['user']* are trusted super-global variables; *$reviewable* which obtains its data from a type of trusted data source, database, is a trusted global variable.

Listing 24: Example of multiple access control checks for one SSO

```
if($_SESSION['user'] == null)  //annotated check
exit();
require_permission(); //annotated check
$reviewable = mysql_query("SELECT reviewable FROM ".
    REVIEWS_TABLE.");
if($reviewable){ //annotated check
query("INSERT INTO REVIEWS_TABLE." ); //SSO
}
```

Listing 25: Definition of require_permission().

```
function require_permission(){
$permission_level = $_SESSION['permission_level']; //data
    from super-global variable
if($permission_level == null)
exit();
if($permission_level > 0)
return;
else
exit();
}
```

I define a security critical variable as follows:

(1) It is directly or indirectly referenced by an access control check through inter-procedural

data flow chain.

(2) It is either a global or super-global variable.

(3) It is a trusted variable, that is not tainted by data from an untrusted data source such as user input (e.g. super global variables *$_POST* and *$_GET* represents user input, and they are not trusted variable).

For example, function call *require_admin()* is an identified access control check for SSOs in web application SACRF. Listing 26 shows the definition of it, and as a related function *is_admin()* is shown in Listing 27. There exists a reference chain: *require_admin() ->is_admin() ->$_SESSION['privilege']*, and thus, the security critical variable extracted from *require_admin()* is *$_SESSION['privilege']* because it indirectly references the variable *$_SESSION['privilege']*, which is a super-global variable and is regarded as trusted data and not tainted. There are other identified access control checks for SSOs in SCARF, I simplify the case for the sake of illustration, and suppose the set of identified access control checks for SSOs in SCARF, the set of security critical variable is {*$_SESSION['privilege']*};

Listing 26: require_admin() declaration in functions.php in SCARF.

```
function require_admin(){
if(!is_admin())
        die("You don't have access to view it");
}
```

Listing 27: is_admin() declaration in functions.php in SCARF.

```
function is_admin() {
```

```
        if ($_SESSION['privilege'] == 'admin') return TRUE;
        else return FALSE;
}
```

Listing 28 gives another example to illustrate Boolean expressions as the identified access control check for SSOs and the taint propagation involved in the process to determine whether the variable is trusted or tainted. The identified access control check is Boolean expression *$user_level == 'Admin' && $action =='deleteAll'*. There exist data flow reference chains from the check, *$user_level == 'Admin' && $action ==deleteAll' ->$user_level ->$_SESSION['user_level']* and *$user_level == 'Admin' && $action =='deleteAll' ->$action ->$_GET['action']*. So initially, the set of variables obtained is *$user_level, $action, $_SESSION['user_level'], $_GET['action']*}. Then in order to obtain the security critical variables, first, I remove the variables that are not global or super-global variables, so variables *$user_level*, *$action* are removed; second, I further remove global variables that capture untrusted data, because *$_GET['action']* is data from user input and it is widely regarded as a tainted source, *$_GET['action']* is removed. Then the obtained set of security critical variables is {*$_SESSION['user_level']*};

Listing 28: Example of Boolean expression as access control check

```
$action = $_GET['action'];
$user_level = $_SESSION['user_level']; //data from global
    variables
if ($user_level == 'Admin' && $action =='deleteAll') //annotated check
```

```
query("DELETE FROM sensitive_table "); //SSO
```

Determining conditional links

For each link in the sitemap, I determine whether it is displayed conditioned upon some access control checks. I make this determination based on the observation that access control logic used to determine displaying of the link share security critical variables, such as global session variables, with access control logic for SSOs. Since the access control checks for SSOs have already been identified via interactive code annotation, my approach is to analyze the source code to look for whether conditions placed upon displaying links share the same security critical variables as access control checks for SSOs.

Given all the links in an application and the identified access control checks for SSOs, steps to identify conditional links are summarized in the following,

1. Extract a set of security critical variables from the set of identified access control checks for SSOs, referred to as SCV_*ssos*.

2. For each link *l* given,

   (a) First, identify all potential access control checks for displaying of the link (detailed in next paragraph), referred to as a set of potential control checks for the link.

   (b) Second, extract a set of security critical variables from the set of potential control checks for the link, referred to as SCV_link_*l*.

   (c) Third, compare SCV_link_*l* with the set of security critical variables for SSOs SCV_*ssos*, if the two sets share one or more elements (variables), then the link

is regarded as a conditional link.

Identify all potential access control checks for displaying of a link

Potential access control checks for displaying of a link include, (1) Boolean expressions being the condition for conditional branches that determine whether the code for displaying of the link will be executed, and (2) function calls that contain one or more such conditional branches. Inter-procedural control dependence analysis is performed to find conditional branches that determine whether the code for displaying of the link will be executed. Boolean expressions being the condition of such conditional branches are added into the set of potential access control checks; in addition, function calls that contain one or more such conditional branches are also added into the set. Listing 29 shows an example from application SCARF, the if-branch *if(is_admin())* determines whether the link displaying code will be executed, so Boolean expression *is_admin()* is added into the set of potential access control checks.

Listing 29: Example in showsessions.php in SCARF.

```
if (is_admin()) {
print "␣<a␣href='editsession.php'>(edit)</a>";
}
```

Suppose the set of potential access control checks for displaying of the link 'editsession.php' is $\{$*is_admin()*$\}$, the definition of *is_admin()* is shown in Listing 27. Then by following step 2.(b) of the process, the set of security critical variables for displaying of the link 'editsession.php', SCV_*editsession.php* is $\{$*$_SESSION['privilege']*$\}$. Then

SCV_link_*editsession.php* shares an element (variable) *$_SESSION['privilege']* with the set of control checks for SSOs SCV_*ssos* which is {*$_SESSION['privilege']*}. And thus *is_admin()* is considered as a valid check, and the link 'editsession.php' shown in Listing 29 is considered as a conditional link.

6.2.1.3 Identify Candidate Pages

Based on the rough sitemap constructed in Section 6.2.1.1 and the set of conditional links identified in Section 6.2.1.2, a candidate web page for inserting application sensors is a web page satisfying the following conditions: (1) the page has at least one SSO, and (2) there does not exist any navigation paths from any entry page to it with only unconditional links.

The process of identifying candidate pages is straightforward, based on the set of conditional links, I tag all the links on the rough sitemap as either conditional or unconditional, and transform the sitemap into a weighted graph. Then the problem of identifying a candidate page is equivalent to the problem of identifying pages with SSOs to which the shortest paths from any entries are weighted larger than a threshold. In Figure 16, all page nodes with a star on its top right are pages with SSOs; six pages in the dashed box are identified as candidate pages.

6.2.2 Evaluation Results

I performed a proof-of-concept evaluation using two PHP open source projects Wheatblog and SCARF that were used in previous research [85, 83, 81, 37, 43, 70]. Wheatblog is a blogging application with over 4000 lines of code. SCARF is a conference paper discussion forum with over 1300 lines of code. I applied all fixes to all known vulnerabilities in these projects. I seek to answer the following questions: (a) what are the likely false pos-

Figure 17: Admin/add_link.php is a candidate page.

itive scenarios for inserted application sensors: i.e. a forced browsing event is incorrectly detected; and (b) false negatives: a forced browsing event is not identified by the approach outlined in Section 6.2.1.

Table 12 and Table 13 summarize my results for Wheatblog and SCARF respectively. Each row represents a page, e.g. the web page in row 3, admin/add_post.php, has one SSO; page 17 has an unconditional link pointing to it, pages 10-14 and 18 have conditional links pointing to it. Pages identified as a candidate for insertion of application sensors are shown in italics. Figure 17 shows a part of the constructed sitemap. Dashed links represent conditional links, solid links for unconditional links, and dashed box represents pages with SSO. From the figure, one can see that admin/add_link.php is a candidate page, because one cannot navigate to it using unconditional links.

Page admin/add_post.php, on row 3, on the other hand is not a candidate page because it is pointed to by entry pages admin/index.php. This means an unauthorized user could access admin/add_post.php by clicking a visible link on page admin/index.php.

Among all the 14 pages with SSOs, 10 pages are considered as candidate pages for putting sensors by the algorithm described in Section 6.2.1. The results for SCARF are similarly reported below, with 4 candidate pages out of 14. All candidate pages re-quire administrative privileges and are ”hidden” behind administrative login pages.

The accuracy of the algorithm described in Section 6.2.1 can be evaluated based on its false positives and false negatives. A false positive is when I mistakenly identify a page

Table 12: Wheatblog results (Candidate pages in italic).

No.	PHP executable file	Num of SSOs in it	Pages having unconditional links to it	Pages having conditional links to it
1	*admin/add_category.php*	1		12
2	*admin/add_link.php*	1		13
3	admin/add_post.php	1	17	10-14, 18
4	*admin/delete_category.php*	1		12
5	*admin/delete_comment.php*	2		10
6	*admin/delete_link.php*	1		13
7	*admin/delete_post.php*	1		15,19
8	*admin/edit_categories.php*	1		12
9	*admin/edit_comment.php*	1		10
10	admin/edit_post.php	0		5,15,19
11	*admin/edit_post_002.php*	1		10
12	admin/manage_categories	0	17	4,8,10-14,18
13	admin/manage_links.php	0	17	2,6,10-14,18
14	*admin/manage_links_002.php*	1		13
15	admin/manage_posts.php	0	17	7,9-14,18
16	admin/manage_users.php	3	17	10-14,18
17	admin/index.php	0	Entry page	Entry page
18	includes/header.php	0	3,5,10,11,17,19-25,27-29	12-16,26,30
19	index.php	0	Entry page	Entry page
20	add_comment.php	2	21	
21	view_by_permalink.php	0	19,20,22-24	
22	view_by_category.php	0	18,19,21,23,24,27	15
23	view_by_archive.php	0	25	26
24	view_by_title.php	0	18	
25	archive.php	0		26
26	update_archive.php	0	Entry page	Entry page
27	list_category.php	0	Entry page	Entry page
28	registration.php	0	18	
29	view_links.php	0	Entry page	Entry page
30	admin/settings.php	1	Entry page	Entry page

as only reachable by conditioned links (i.e. one must first be authorized before accessing the page). Such a situation could occur, for example, due to inaccuracy of my approach to identify conditioned links. I examined all candidate pages in Tables 12 and 13 and did not find any such case.

False negative refers to situations where a forced browsing event would be missed. I examined every page in Wheatblog and SCARF and, based on the intended application function, determine whether a forced browsing attack could occur on the page. I found two cases in SCARF, as illustrated in Figure 18. In both cases, the condition used to check whether a link should be displayed has no relation with code written for access control for the SSO. These are indicated by a marked link in Figure 18. Page 10, index.php, checks

Table 13: SCARF results (Candidate pages in italic).

No.	PHP executable file	Num of SSOs in it	Pages having unconditional links to it	Pages having conditional links to it
1	*editpaper.php*	8		7-9
2	*addsession.php*	1		7
3	*editsession.php*	6		9
4	useroptions.php	3	7	5
5	comments.php	3	8	7
6	*generaloptions.php*	3		7
7	header.php	0	1-6,8-14	
8	showpaper.php	0	9	1,3
9	showsessions.php	0	3,12	
10	index.php	0	Entry page	Entry page
11	fogot.php	0	12	
12	login.php	0	5,7	
13	install.php	0	10	
14	register.php	0	7,10,12	

to make sure an administrative account has been created before displaying links to pages 7 and 9. These are false negatives only for an uninitialized system. As soon as the system is initialized, they are not false negatives.

6.2.3 Automatic Sensor Insertion

Access control checks for SSOs in six open source PHP applications are summarized in Table 14. For example, SCARF has 24 instances where SSOs are invoked. For each SSO invocation I identified the corresponding access control code. All 24 access control checks are function calls with abnormal return code. In contrast all 21 access control checks in Wheatblog involve conditional statements. Based on analysis of these empirical results, I create a model for automatic insertion of application sensors as described below.

Various information could be captured by sensors by calling APIs or retrieving global variables. For example, in PHP, a session id could be obtained by calling session_id(), client host and IP address could be obtained by retrieving from global variable $_SERVER. The sensor could also provide a serialized object for session content along with time and date stamps. For the rest of the chapter, I focus on how to insert an application sensor into

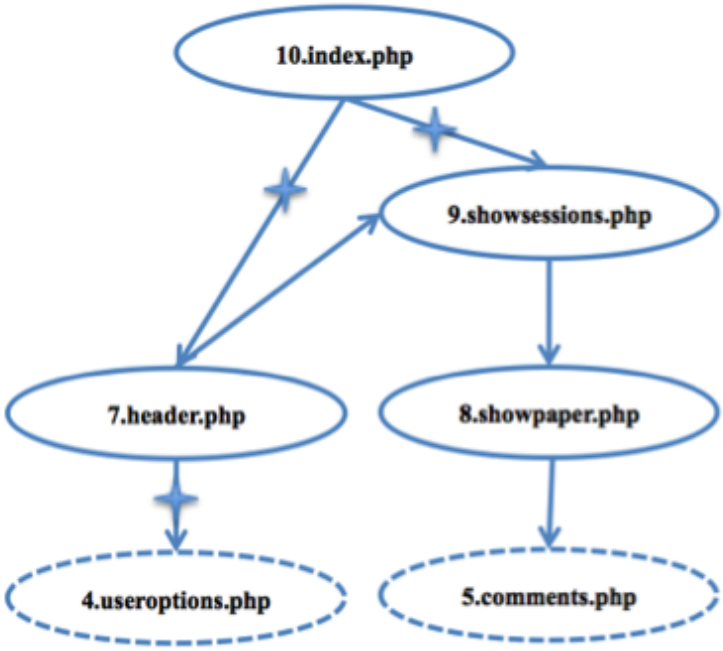

Figure 18: Two pages with SSOs (dashed circle) but not considered candidate in SCARF.

Table 14: Summary about access control checks made in open source projects.

Project	LOC	Description	Num of access checks made	Access control check is conditional statement	Access control check is function call with abnormal return code
SCARF	1,318	Conference discussion forum for papers	24	0	24
Wheatblog	4,032	Blogging system	21	21	0
FreeWebShop	8.613	Online store	56	56	0
Mybloggie	8,874	Blogging system	18	18	0
PhpStat 1.5	12,700	Application presenting IM statistics	30	30	0
Moodle2.1.0	625,000	Course management	166	64	102

a piece of access control check code to report the event of the access control failure. This must be in done in such a way that does not impact the normal application flow. For this purpose, I use the function call sensor() to denote such an application sensor.

Since application sensors are used to capture failure events of access control checks, they should be placed on all execution paths other than the path leading to execution of the SSO. As described in previous section, access control checks could be Boolean expressions or function calls with abnormal return code. Among Boolean expressions, there are Boolean

expressions in if-branch, or switch-branch. I did not observe any access control checks involving iteration logic. I first describe sensor insertion for Boolean expression access control checks, then for function call access control checks.

6.2.3.1 Sensor Insertion for Boolean Expression Access Control Checks

I illustrate the sensor insertion through examples. Figures 19, 20, 21 show code examples for different cases when the SSO stays in if-branch, if-elseif branch, and switch branch. The left side of each figure is the code before sensor insertion, in which access control check is in italics, the right side is the code after sensor insertion, in which the newly inserted code is shown in bold. In all these cases, sensors are put on the paths other than the execution paths leading to the SSO. For example, in Figure 20, three different cases exist for where the SSO stays within the if-elseif-branch, each row in the figure represents a case, case 1 means access control is granted if Boolean expression condition_a evaluates to true, and after sensor insertion, sensors have been put into all branches other than the execution paths leading to the SSO, and during the process an explicit else branch is created to enable the insertion.

6.2.3.2 Sensor Insertion for Function Call Access Control Checks

I illustrate the sensor insertion for this case through an example adapted from SCARF. Figure 22 shows the code example before sensor insertion, in which the access control check is in italics and abnormal return code is marked with comments; Figure 22 shows the code examples after sensor insertion, in which the newly inserted code is shown in bold. Since access control check require_login() dominates the path flowing to the SSO, access control failure is represented by the abnormal return of the access control function, require_login(). This means application sensors should be inserted in the require_login()

1	`if (condition){`
	`  SSO;`
	`}`
	
	(1) Before sensor insertion
		`if (condition){`
		`  SSO;`
		`} else{`
		`  sensor();`
		`}`
		
		(2) After sensor insertion

2	`if (condition){`
	
	`} else{`
	`  SSO;`
	`}`
	
	(1) Before sensor insertion
		`if (condition){`
		`  sensor();`
		`} else{`
		`  SSO;`
		`}`
		
		(2) After sensor insertion

Figure 19: Sensor insertion for If-branch.

right before the abortion of normal execution path, as shown in Figure 22. There may exist multiple abnormal returns, each needs to be instrumented with a sensor. There might be further function calls in the definition of function require_login(), whose definition contains abnormal return code and needs to be analyzed. But the general analysis process is the same.

1	`if (condition_a){`
	`  SSO;`
	`} elseif (condition_b){`
	
	`}`
	
	(1) Before sensor insertion
		`if (condition_a){`
		`  SSO;`
		`} elseif (condition_b) {`
		`  sensor();`
		`} else{`
		`  sensor();`
		`}`
		
		(2) After sensor insertion

2	`if (condition_a){`
	
	`} elseif (condition_b) {`
	`  SSO;`
	`}`
	
	(1) Before sensor insertion
		`if (condition_a){`
		`  sensor();`
		`} elseif (condition_b){`
		`  SSO;`
		`} else{`
		`  sensor();`
		`}`
		
		(2) After sensor insertion

3	`if (condition_a){`
	
	`} elseif (condition_b){`
	
	`} else{`
	`  SSO;`
	`}`
	
	(1) Before sensor insertion
		`if (condition_a){`
		`  sensor();`
		`} elseif (condition_b){`
		`  sensor();`
		`} else{`
		`  SSO;`
		`}`
		
		(2) After sensor insertion

Figure 20: Sensor insertion for If-Elseif-branch.

1	switch (*expr*) {
	  case label1:
	  SSO;
	  break;
	  case label2:
	  break;
	}
	
	(1) Before sensor insertion
		switch (*expr*) {
		  case label1:
		  SSO;
		  break;
		  case label2:
		  sensor();
		  break;
		  default:
		  sensor();
		  break;
		}
		
		(2) After sensor insertion

2	switch (*expr*) {
	  case label1:
	  break;
	  case label2:
	  break;
	  default:
	  SSO;
	  break;
	}
	
	(1) Before sensor insertion
		switch (*expr*) {
		  case label1:
		  sensor();
		  break;
		  case label2:
		  sensor();
		  break;
		  default:
		  SSO;
		  break;
		}
		
		(2) After sensor insertion

Figure 21: Sensor insertion for Switch-branch.

```
require_loggedin();
query("INSERT INTO comment
…); //SSO
```

(1) Access control check and SSO

```
function require_loggedin()
{
if (getEmail() === FALSE) {
die ("You must be logged
in"); //abnormal return
}
}
```

(2) Definition of require_loggedin()

Before sensor insertion

```
require_loggedin();
query("INSERT INTO comment
…); //SSO
```

(1) Access control check and SSO

```
function require_loggedin()
{
if (getEmail() === FALSE) {
sensor();
die ("You must be logged
in"); //abnormal return
}
}
```

(2) Definition of require_loggedin()

After sensor insertion

Figure 22: Sensor insertion for function call with abnormal return code.

CHAPTER 7: CONCLUSION AND FUTURE WORK

Software vulnerabilities have become increasingly pervasive targets for attackers and result in severe data and financial loss to organizations and individuals. One leading source of software vulnerabilities is the insecure code written by developers. Although many of the vulnerabilities could be addressed through secure programming practices and there have already been a collection of secure programming practices well documented, developers continue to make same mistakes. With the rapidly growing complexity of software, security bugs are difficult to avoid.

7.1 Summary of Contributions

This dissertation presents a generalized framework, interactive static analysis, a developer-oriented hybrid model for vulnerability detection and mitigation, which integrates static analysis into the Integrated Development Environment (IDE) as a plug-in, facilitating two-way interaction between static analysis and developer. Developers are not required to have any knowledge of static analysis, nor are they security experts. The goal of this approach is to assist developers in detecting and mitigating vulnerabilities during code construction phrase and solicit application specific knowledge from the developer to customize static analysis. In particular, I focus on using this approach for detecting access control vulnerabilities. This approach also enables automatic placement of application sensor for application based intrusion detection.

Focusing on access control vulnerability detection, this dissertation found some unrea-

sonable implicit assumptions of previous research techniques to automatically detect access control vulnerabilities, which significantly limit the performance of the auto detection techniques, such as leading to large false negatives. This is demonstrated through studying six open source PHP web applications. It argues that a hybrid approach, such as interactive static analysis, is a much more reasonable approach for detecting access control vulnerabilities.

This dissertation presents an interactive static analysis prototype for access control vulnerability detection as a plug-in in Eclipse PHP IDE [19], called ASIDE-PHP (Application Security plug-in for the Integrated Development Environment for PHP). It describes an extensive evaluation of the prototype with six open source PHP web applications including a large project named Moodle [71]. In addition to finding all access control vulnerabilities detected by previous work, it found 20 zero-day access control vulnerabilities.

Based on the interactive static analysis framework, this dissertation proposes an approach for automatic placement of application sensors to enable application-based intrusion detection systems. This work focuses on using application sensors to detect events of failed access control to detect privilege escalation attacks. A proof of concept analysis of two open source projects has been conducted to evaluate the effectiveness of the approach. In addition, it illustrates a model for automatically inserting application sensors into applications to detect access control events, based on an extensive case study involving six open source PHP projects.

7.2 Future Work

This dissertation has laid a foundation for future research in this direction. I outline some open areas for further research.

7.2.1 Interactive Static Analysis

First, because developer adoption is a critical success factor for interactive static analysis, one must design an appropriate user interface and interaction for developers. To this end, the design must be intuitive, fit seamlessly into developer workflow, and minimize unnecessary distractions. More research is also needed towards automatic annotation mechanisms to minimize unnecessary interactive annotations.

Second, it would be interesting to explore more hybrid heuristics, involving developer input as well as program analysis, to further improve the performance of detecting access control vulnerabilities. For example, the notion of auto annotation can be expanded by incorporating more program analysis heuristics to reduce the number of annotations requested to developers.

Third, it will be very interesting to applying interactive static analysis to discover vulnerabilities other than access control vulnerabilities, such as insecure API usage.

7.2.2 Placing Application Sensors for Application Based Intrusion Detection and Prevention

More empirical evaluations of applications of different types and complexity are needed to validate the results presented here. Future directions include looking at different ways for sensor implementation, evaluating the correctness of inserted sensors and its impact on performance, expanding forced browsing detection to include cases where access controls are enforced as part of the SQL statement, and examining strategies for accurate intrusion detection using application sensors.

REFERENCES

[1] Cve-2012-2354. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2354.

[2] Cve-2012-2355. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2355.

[3] Cve-2012-2358. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2367.

[4] Cve-2012-2358. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2358.

[5] Cve-2012-3391. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3391.

[6] Cve-2012-3397. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3397.

[7] Cve-2012-5473. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5473.

[8] Findbugs. http://findbugs.sourceforge.net/.

[9] Fix changes for cve-2012-2354. http://git.moodle.org/gw?p=moodle.git;a=commit;h=48e03792ca8faa2d781f9ef74606f3b3f0d3baec.

[10] Fix changes for cve-2012-2355. http://git.moodle.org/gw?p=moodle.git;a=commitdiff;h=76cf77e4d3e3ca30a2a983aa71bc5a2090133668.

[11] Fix changes for cve-2012-2358. http://git.moodle.org/gw?p=moodle.git;a=commitdiff;h=e8027a40cd16f50402a6c13f3bfd3128639a797c.

[12] Fix changes for cve-2012-2367. http://git.moodle.org/gw?p=moodle.git;a=commitdiff;h=3c0ea9b2841a3af83b3726d981d6f12d5b73bd4c.

[13] Fix changes for cve-2012-3391. http://git.moodle.org/gw?p=moodle.git;a=commitdiff;h=35124c3c4fd4692a54359fe32d3c29e2590dbb83.

[14] Fix changes for cve-2012-3397. http://git.moodle.org/gw?p=moodle.git;a=commitdiff;h=a098f340fe2864d91ac056fc90250564883e62a9.

[15] Fix changes for cve-2012-5473. http://git.moodle.org/gw?p=moodle.git;a=commitdiff;h=76fb0443b6f84e25d7ea983829e78b5556f2fcdf.

[16] Grammatech. http://www.grammatech.com/.

[17] Klocwork. http://www.klocwork..com/.

[18] Veracode static aanlyzer. http://www.veracode.com/products/static.

[19] Eclipse php ide. http://projects.eclipse.org/projects/tools.pdt, 2012.

[20] Owasp secure coding practices quick reference guide. https://http://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf, 2013.

[21] Top ten vulnerabilities. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013.

[22] Whitehat security website security statistics report. https://http://www.whitehatsec.com/assets/WPstatsReport_052013.pdf, 2013.

[23] Moodle statistics. http://moodle.org/stats/, 8 2014.

[24] Eclipse pdt ast model illustration. http://www.eclipse.org/pdt/articles/ast/PHP_AST.html, 2015.

[25] Hp fortify static code analyzer. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812-.UUnHuVthvUY, 3 2015.

[26] Pmd. http://pmd.sourceforge.net/, 2015.

[27] O. Alomari and Z. A. Othman. Bees algorithm for feature selection in network anomaly detection. *Journal of Applied Sciences Research*, 8(3):1748–1756, 2012.

[28] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Technical report, 2000.

[29] R. Bace and P. Mell. Nist special publication on intrusion detection systems. Technical report, DTIC Document, 2001.

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

[31] M. Bishop and B. Orvis. A clinic to teach good programming practices. In *Proceedings of the 10th Colloquium for Information Systems Security Education*, pages 168–1174, 2006.

[32] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 607–618. ACM, 2010.

[33] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 575–586. ACM, 2011.

[34] B. Chess and J. West. *Secure programming with static analysis*. Pearson Education, 2007.

[35] Y. Y. Chung and N. Wahid. A hybrid network intrusion detection system using simplified swarm optimization (sso). *Applied Soft Computing*, 12(9):3014–3022, 2012.

[36] S. Corporation. Symantec global internet security threat report. *White Paper, Symantec Enterprise Security*, 1, 2013.

[37] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium*, pages 267–282, 2009.

[38] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[39] H. Debar, M. Dacier, and A. Wespi. A revised taxonomy for intrusion-detection systems. In *Annales des télécommunications*, volume 55, pages 361–378. Springer, 2000.

[40] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the ear: discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 251–262. ACM, 2011.

[41] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, pages 143–160, 2010.

[42] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1):18–28, 2009.

[43] F. Gauthier, T. Lavoie, and E. Merlo. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 209–218. ACM, 2013.

[44] P. Guo. Refining students' coding and reviewing skills. *Communications of the ACM*, 57(9):10–11, 2014.

[45] S.-J. Horng, M.-Y. Su, Y.-H. Chen, T.-W. Kao, R.-J. Chen, J.-L. Lai, and C. D. Perkasa. A novel intrusion detection system based on hierarchical clustering and support vector machines. *Expert systems with Applications*, 38(1):306–313, 2011.

[46] M. Howard and S. Lipner. *The security development lifecycle*. O'Reilly Media, Incorporated, 2009.

[47] C. inc. Coverity static analysis verification engine. `http://www.coverity.com/products/coverity-save.html`.

[48] O. inc. Secure coding guidelines for the java programming language. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`, 2013.

[49] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[50] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[51] H. L. T. T. Jun Zhu, Bill Chu. Mitigating access control vulnerabilities through interactive static analysis. In *ACM Symposium on Access Control Models and Technologies*. ACM, 2015.

[52] P. Kabiri and A. A. Ghorbani. Research on intrusion detection and response: A survey. *IJ Network Security*, 1(2):84–102, 2005.

[53] A. Kartit, A. Saidi, F. Bezzazi, M. El Marraki, and A. Radi. A new approach to intrusion detection system. *Journal of theoretical and applied information technology*, 36(2):284–289, 2012.

[54] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1):41–84, 2005.

[55] C. Kolias, G. Kambourakis, and M. Maragoudakis. Swarm intelligence in intrusion detection: A survey. *computers & security*, 30(8):625–642, 2011.

[56] C. Kruegel and T. Toth. A survey on intrusion detection systems. In *TU Vienna, Austria*. Citeseer, 2000.

[57] J. A. Kupsch and B. P. Miller. Manual vs. automated vulnerability assessment: A case study. In *First International Workshop on Managing Insider Security Threats (MIST)*, pages 83–97, 2009.

[58] M. S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 3–12. ACM, 2008.

[59] A. Lazarevic, V. Kumar, and J. Srivastava. Managing cyber threats: issues, approaches, and challenges. *Chapter: A survey of Intrusion Detection techniques. Boston: Kluwer Academic Publishers, doi*, 10:b104908, 2005.

[60] D. LeBlanc and M. Howard. *Writing secure code*. Pearson Education, 2002.

[61] Y. Li, J. Xia, S. Zhang, J. Yan, X. Ai, and K. Dai. An efficient intrusion detection system based on support vector machines and gradually feature removal method. *Expert Systems with Applications*, 39(1):424–430, 2012.

[62] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

[63] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, pages 18–18, 2005.

[64] J. Mar, I.-F. Hsiao, Y. C. Yeh, C.-C. Kuo, and S.-R. Wu. Intelligent intrusion detection and robust null defense for wireless networks. *International Journal of Innovative Computing Information and Control*, 8(5A):3341–3359, 2012.

[65] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *ACM SIGPLAN Notices*, volume 40, pages 365–383. ACM, 2005.

[66] G. McGraw. Talk on static analysis tools. `http://www.informit.com/articles/article.aspx?p=1680863`.

[67] G. McGraw. Software security. *Security & Privacy, IEEE*, 2(2):80–83, 2004.

[68] Microsoft. Microsoft sal annotations. `http://msdn.microsoft.com/en-us/library/ms235402.aspx`, 2011.

[69] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications*, 36(1):42–57, 2013.

[70] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 690–701. ACM, 2014.

[71] Moodle.org. Moodle. `https://moodle.org/`.

[72] A. Murali and M. Rao. A survey on intrusion detection approaches. In *Information and Communication Technologies, 2005. ICICT 2005. First International Conference on*, pages 233–240. IEEE, 2005.

[73] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[74] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[75] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *ACM SIGOPS Operating Systems Review*, 36(SI):45–60, 2002.

[76] M. J. Ranum. Thinking about firewalls. In *Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II)*, volume 8. Citeseer, 1993.

[77] J. Reason. *Human error*. Cambridge university press, 1990.

[78] F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *Systems and Networks Communications, 2008. ICSNC'08. 3rd International Conference on*, pages 23–26. IEEE, 2008.

[79] R. C. Seacord. *The CERT C secure coding standard*. Pearson Education, 2008.

[80] A. Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[81] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. *ACM SIGPLAN Notices*, 46(10):1069–1084, 2011.

[82] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[83] S. Son and V. Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, page 8. ACM, 2011.

[84] P. Stavroulakis and M. Stamp. *Handbook of information and communication security*. Springer Science & Business Media, 2010.

[85] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2011.

[86] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[87] K. R. Van Wyk. *Secure coding: principles and practices*. " O'Reilly Media, Inc.", 2003.

[88] S.-S. Wang, K.-Q. Yan, S.-C. Wang, and C.-W. Liu. An integrated intrusion detection system for cluster-based wireless sensor networks. *Expert Systems with Applications*, 38(12):15234–15243, 2011.

[89] C. Watson, M. Coates, J. Melton, and D. Groves. Creating attack-aware software applications with real-time defenses. *Crosstalk-September/October*, 2011.

[90] C. Watson, D. G. J. Melton, J. A.-Z. R. B. Michael, C. C. M. J. Reynolds, and M. Coates. Appsensor guide. 2008.

[91] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[92] J. Xie, B. Chu, and H. R. Lipford. Idea: interactive support for secure software development. In *Engineering Secure Software and Systems*, pages 248–255. Springer, 2011.

[93] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton. Aside: Ide support for web application security. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 267–276. ACM, 2011.

[94] J. Xie, H. Lipford, and B.-T. Chu. Evaluating interactive support for secure programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2707–2716. ACM, 2012.

[95] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 161–164. IEEE, 2011.

[96] M. Xie, S. Han, B. Tian, and S. Parvin. Anomaly detection in wireless sensor networks: A survey. *Journal of Network and Computer Applications*, 34(4):1302–1325, 2011.

[97] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, volume 6, pages 179–192, 2006.

[98] J. Zhu, J. Xie, H. R. Lipford, and B. Chu. Supporting secure programming in web applications through interactive static analysis. *Journal of Advanced Research*, 5(4):449–462, 2014.

APPENDIX A: ZERO-DAY ACCESS CONTROL VULNERABILITIES DETECTED IN THE EVALUATION

As mentioned in the evaluation section in Chapter 5, I detected 20 zero-day access control vulnerabilities in the evaluated projects. Since two of them have already been discussed in Chapter 5, I will describe the 18 remaining ones in this appendix. A brief overview is shown in Table 15. I will describe the findings in each project separately in the following sections.

Table 15: Vulnerability detection results to be described in this appendix.

Project	**0-day Vul. by ASIDE-PHP**
Mybloggie 2.1.3	14[Missing check]
PhpStat 1.5	4[Missing check]

In this section, for each zero-day vulnerability, I list out its vulnerable code and give a brief explanation of the affected SSO (operation on database table).

Fourteen missing check vulnerabilities in Mybloggie 2.1.3

1. Vulnerable code about INSERT INTO USER_TBL in adduser.php of Mybloggie, there exists no access control checks for it.

```
$result = $db->sql_query("INSERT INTO ".USER_TBL." SET
    user='$user', password='$password', level='$level'");
    //SSO
```

2. Vulnerable code about INSERT INTO CAT_TBL in addcat.php of Mybloggie, there exists no access control checks for it.

```
$result = $db->sql_query("INSERT INTO ".CAT_TBL." SET
    cat_desc='$cat_desc'"); //SSO
```

3. Vulnerable code about INSERT INTO POST_TBL in add.php of Mybloggie, there exists no access control checks for it.

```
$query = mysql_query("INSERT INTO ".POST_TBL." SET
    user_id='$user_id', subject='$subject', message='
    $message', timestamp='$timestamp', cat_id='$cat_id'")
    ; //SSO
```

4. Vulnerable code about UPDATE POST_TBL in edit.php of Mybloggie, there exists no access control checks for it.

```
if ($edit_date) {
$result = $db->sql_query("UPDATE ".POST_TBL." SET
    subject='$subject', message='$message', timestamp='
    $timestamp', cat_id='$cat_id' WHERE post_id='$post_id
    '"); //SSO
}
else {
$result = $db->sql_query ("UPDATE ".POST_TBL." SET
    subject='$subject', message='$message', cat_id='
    $cat_id' WHERE post_id='$post_id'"); //SSO
}
```

5. Vulnerable code about UPDATE COMMENT_TBL in view.php of Mybloggie, there exists no access control checks for it.

```
$result = $db->sql_query("UPDATE " . COMMENT_TBL . " SET
    comment_subject='$commentsubject', comments='
    $commenttext', poster = '$commentname', email='
    $commentemail' , home='$commenthome' Where comment_id
    ='$comment_id' "); //SSO
```

6. Vulnerable code about INSERT INTO COMMENT_TBL in trackback.php, there exists no access control checks for it.

```
$result = $db->sql_query("INSERT INTO " . COMMENT_TBL . "
    SET post_id='$tb_id', comment_subject='$title',
    comments='$excerpt', com_tstamp='$timestamp', poster
    = '$blog_name', home='$url', comment_type='trackback'
    "); //SSO
```

7. Vulnerable code about UPDATE COMMENT_TBL in view.php of Mybloggie, there exists no access control checks for it.

```
$result = $db->sql_query("UPDATE " . COMMENT_TBL . " SET
    comment_subject='$commentsubject', comments='
    $commenttext', poster = '$commentname', email='
    $commentemail' , home='$commenthome' Where comment_id
    ='$comment_id' "); //SSO
```

8. Vulnerable code about UPDATE CAT_TBL in editcart.php of Mybloggie, *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* is an invalid check because function header() is used to send raw HTTP header and does not generate an exception or terminate the program execution, so it is considered as a missing check vulnerability.

```
// Added security
if (!isset($_SESSION['username']) && !isset($_SESSION['
    passwd'])) {
  header("Location: ./login.php");
}
$result = $db->sql_query("UPDATE ".CAT_TBL." SET
    cat_desc='$cat_desc' where cat_id='$cat_id'"); //SSO
```

9. Vulnerable code about UPDATE USER_TBL in edituser.php of Mybloggie, *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* is an invalid check because function header() does not generate an exception or terminate the program execution, so it is considered as a missing check vulnerability.

```
// Added security
if (!isset($_SESSION['username']) && !isset($_SESSION['
    passwd'])) {
header("Location: ./login.php");
}
```

```
$result = $db->sql_query("UPDATE ".USER_TBL." SET user='
    $user', password='$password', level='$level' where id
    ='$id'"); //SSO
```

10. Vulnerable code about DELETE FROM POST_TBL in del.php of Mybloggie, *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* is an invalid check because function echo() is used to output string(s) and does not generate an exception or terminate the program execution, so it is considered as a missing check vulnerability.

```
// Added security
if (!isset($_SESSION['username']) && !isset($_SESSION['
    passwd'])) {
echo "<meta http-equiv=\"Refresh\" content=\"2;url=".
    self_url()."/login.php\" />";
}
$result = $db->sql_query("DELETE FROM ".POST_TBL." WHERE
    post_id='$post_id'"); //SSO
```

11. Vulnerable code about DELETE FROM CAT_TBL in delcat.php of Mybloggie, *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* is an invalid check because function header() does not generate an exception or terminate the program execution, so it is considered as a missing check vulnerability.

```
// Added security
if (!isset($_SESSION['username']) && !isset($_SESSION['
    passwd'])) {
header("Location: ../login.php");
}
$result = $db->sql_query("DELETE FROM ".CAT_TBL." WHERE
    cat_id=$cat_id"); //SSO
```

12. Vulnerable code about DELETE FROM COMMENT_TBL in delcomment.php of Mybloggie, *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* is an invalid check because function header() does not generate an exception or terminate the program execution, so it is considered as a missing check vulnerability.

```
if (!isset($_SESSION['username']) && !isset($_SESSION['
    passwd'])) {
header("Location: ../login.php");
}
$result = $db->sql_query("DELETE FROM ".COMMENT_TBL." 
    WHERE comment_id=$comment_id"); //SSO
```

13. Vulnerable code about DELETE FROM USER_TBL in deluser.php of Mybloggie, *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* is an invalid check because function header() does not generate an exception or terminate the program execution, so it is considered as a missing check vulnerability.

```
// Added security
if (!isset($_SESSION['username']) && !isset($_SESSION['
    passwd'])) {
header("Location: ../login.php");
}
$result = $db->sql_query("DELETE FROM ".USER_TBL." WHERE
    id=$id"); //SSO
```

14. Vulnerable code about DELETE FROM POST_TBL in deluser.php of Mybloggie, *!isset($_SESSION['username']) && !isset($_SESSION['passwd']* is an invalid check because function header() does not generate an exception or terminate the program execution, so it is considered as a missing check vulnerability.

```
// Added security
if (!isset($_SESSION['username']) && !isset($_SESSION['
    passwd'])) {
header("Location: ../login.php");
}
$result = $db->sql_query("DELETE FROM ".POST_TBL." WHERE
    user_id=$id"); //SSO
```

Four missing check vulnerabilities in PhpStat 1.5

1. Vulnerable code about INSERT INTO stat_user in refresh.php of PhpStat, there exists no access control checks for it.

```
$dump=start(2); //SSO, because it calls INSERT INTO
    stat_user
```

2. Vulnerable code about UPDATE stat_file in refresh.php of PhpStat, there exists no access control checks for it.

```
$dump=start(2);  //SSO, because it calls UPDATE
    stat_file
```

3. Vulnerable code about UPDATE stat_user in refresh.php of PhpStat, there exists no access control checks for it.

```
$dump=start(2); //SSO, because it calls UPDATE stat_user
```

4. Vulnerable code about UPDATE stat_user in global_stat.php of PhpStat, there exists no access control checks for it.

```
$res=top_10users();  //SSO, because it calls UPDATE
    stat_user
```

APPENDIX B: SEVEN UNSOLVABLE KNOWN ACCESS CONTROL VULNERABILITIES IN MOODLE 2.1.0 DUE TO LOGIC FLAWS

This appendix describes the seven known access control vulnerabilities in Moodle 2.1.0, which result from logic flaws and are not solvable with ASIDE-PHP of this dissertation and auto detection techniques in previous research works. Table 16 lists out these vulnerabilities and their associated security sensitive operations. I describe them one by one in following paragraphs.

Table 16: Seven unsolvable known access control vulnerabilities in Moodle 2.1.0 and associated security sensitive operations.

No.	**Brief Description**	**SSOs**
1	Unauthorized addition of calendar event (CVE-2012-2367)	Insert: event Update: event
2	Unauthorized access to group information (CVE-2012-3397)	Select: groups
3	Unauthorized reading of messages (CVE-2012-2354)	Select: message, message_read
4	Unauthorized addition of quiz questions (CVE-2012-2355)	Insert: quiz_question_instances
5	Unauthorized database modifications (CVE-2012-2358)	Insert: data_records, data_content Update: data_records
6	Unauthorized read of forums (CVE-2012-3391)	Select: forum
7	Unauthorized information access (CVE-2012-5473)	Select: user

1. Unauthorized addition of calendar event (CVE-2012-2367)

This vulnerability allows authenticated users to bypass *moodle/calendar:manageownentries* capability requirement and add a calendar entry via a New Entry action [3]. Figure 23 shows the fix changes for the flaw, which occurs in the function *calendar_get_allowed_type()*. The code in green with a plus marker on its left side represents code that is added to fix the flaw, please note that all figures appears in this appendix follow this present style. In the figure, one line of code *$allowed->user=has_capability('moodle/calendar:manageownentries',*

```
diff --git a/calendar/lib.php b/calendar/lib.php
index 7280f14..fc5573a 100644 (file)
--- a/calendar/lib.php
+++ b/calendar/lib.php
@@ -1740,6 +1740,7 @@ function calendar_get_allowed_types(&$allowed, $course = null) {
         }
         if ($course->id != SITEID) {
             $coursecontext = get_context_instance(CONTEXT_COURSE, $course->id);
+            $allowed->user = has_capability('moodle/calendar:manageownentries', $coursecontext);

             if (has_capability('moodle/calendar:manageentries', $coursecontext)) {
                 $allowed->courses = array($course->id => 1);
```

Figure 23: Fix changes for CVE-2012-2367 [12].

$coursecontext) is added to fix the flaw, which means the vulnerable code before the fix does not have this line of code, and not having this line of the code is the flaw. Figure 24 shows the full vulnerable code of the function *calendar_get_allowed_type()* before the fix. Through analyzing where the function *calendar_get_allowed_type()* is called, I identified the call chain *calendar_get_allowed_type()* <- *calendar_user_can_add_event()*, and *calendar_user_can_add_event()* is an access control function, it determines whether user can add events to calendar. So even if *calendar_user_can_add_event()* has been identified as the access control check through interactive annotation or automatic inference techniques, because the flaw is within the definition of the access control function itself, in this case it is a flaw within the definition of function *calendar_get_allowed_type()* that is called within the definition of access control function *calendar_user_can_add_event()*, the flaw is unique and there exist nothing to compare with, and therefore there exist no signal or clue to trigger a security flaw warning. And thus it is not solvable by ASIDE-PHP of this dissertation and auto detection techniques in previous research works.

2. Unauthorized access to group information (CVE-2012-3397)

In this vulnerability, lib/modinfolib.php does not check a group-membership requirement when determining whether an activity is available or hidden [6]. Fix changes are

```
1724  * Get calendar's allowed types
1725  *
1726  * @param stdClass $allowed list of allowed edit for event  type
1727  * @param stdClass|int $course object of a course or course id
1728  */
1729 function calendar_get_allowed_types(&$allowed, $course = null) {
1730     global $USER, $CFG, $DB;
1731     $allowed = new stdClass();
1732     $allowed->user = has_capability('moodle/calendar:manageownentries', get_system_context());
1733     $allowed->groups = false; // This may change just below
1734     $allowed->courses = false; // This may change just below
1735     $allowed->site = has_capability('moodle/calendar:manageentries', get_context_instance(CONTEXT_COURSE, SITEID));
1736
1737     if (!empty($course)) {
1738         if (!is_object($course)) {
1739             $course = $DB->get_record('course', array('id' => $course), '*', MUST_EXIST);
1740         }
1741         if ($course->id != SITEID) {
1742             $coursecontext = get_context_instance(CONTEXT_COURSE, $course->id);
1743
1744             if (has_capability('moodle/calendar:manageentries', $coursecontext)) {
1745                 $allowed->courses = array($course->id => 1);
1746
1747                 if ($course->groupmode != NOGROUPS || !$course->groupmodeforce) {
1748                     $allowed->groups = groups_get_all_groups($course->id);
1749                 }
1750             } else if (has_capability('moodle/calendar:managegroupentries', $coursecontext)) {
1751                 if($course->groupmode != NOGROUPS || !$course->groupmodeforce) {
1752                     $allowed->groups = groups_get_all_groups($course->id);
1753                 }
1754             }
1755         }
1756     }
1757 }
```

Figure 24: Code of get_allowed_type() before fix [12].

shown in Figure 25, the code in green with a plus marker on its left side represents code that is added to fix the flaw, the code in red with a minus marker on its left side represents code that is removed to fix the flaw, please note that all figures appears in this appendix follow this present style. The line of code *$this->showavailability = 0* in the function definition of *update_user_visible()* in class *cm_info* is missing before the fix, because *$showavailability = 1* represents activity is shown to students as greyed out with information about when it will be available, while *$showavailability = 0* means activity is hidden completely, the flaw of not having *$this->showavailability = 0* cause the activity is not completely hidden from user. The flaw is a unique code instance, and there exist nothing to compare with, and therefore there exist no signal or clue to trigger a security flaw warning.

3. Unauthorized reading of messages (CVE-2012-2354)

This vulnerability allows authenticated users to bypass the *moodle/site:readallmessages* capability check to read arbitrary messages [1]. Figure 26 shows the fix changes. The flaw exists at the access control check. Boolean expression *$user1->id != $USER->id &&*

```
diff --git a/lib/modinfolib.php b/lib/modinfolib.php
index bcbb7ec..b6a6e98 100644 (file)
--- a/lib/modinfolib.php
+++ b/lib/modinfolib.php
@@ -1078,18 +1078,24 @@ class cm_info extends stdClass {
         $modcontext = get_context_instance(CONTEXT_MODULE, $this->id);
         $userid = $this->modinfo->get_user_id();
         $this->uservisible = true;
+        // Check visibility/availability conditions.
         if ((!$this->visible or !$this->available) and
                 !has_capability('moodle/course:viewhiddenactivities', $modcontext, $userid)) {
             // If the activity is hidden or unavailable, and you don't have viewhiddenactivities,
-            // set it so that user can't see or access it
+            // set it so that user can't see or access it.
             $this->uservisible = false;
-        } else if (!empty($CFG->enablegroupmembersonly) and !empty($this->groupmembersonly)
+        }
+        // Check group membership. The grouping option makes the activity
+        // completely invisible as it does not apply to the user at all.
+        if (!empty($CFG->enablegroupmembersonly) and !empty($this->groupmembersonly)
                 and !has_capability('moodle/site:accessallgroups', $modcontext, $userid)) {
             // If the activity has 'group members only' and you don't have accessallgroups...
             $groups = $this->modinfo->get_groups($this->groupingid);
             if (empty($groups)) {
                 // ...and you don't belong to a group, then set it so you can't see/access it
                 $this->uservisible = false;
+                // Ensure activity is completely hidden from user.
+                $this->showavailability = 0;
             }
         }
     }
```

Figure 25: Fix changes for CVE-2012-3397 [14].

```
diff --git a/message/index.php b/message/index.php
index 03995db..85c114e 100644 (file)
--- a/message/index.php
+++ b/message/index.php
@@ -115,8 +115,14 @@ if (!empty($user2id)) {
 }
 unset($user2id);

-//the current user isnt involved in this discussion at all
-if ($user1->id != $USER->id && (!empty($user2) && $user2->id != $USER->id) && !has_capability('moodle/site:readallmessages', $context)) {
+// Is the user involved in the conversation?
+// Do they have the ability to read other user's conversations?
+// There will always be a $user1
+// but $user2 may be null. For example, if viewing $user1's recent conversations
+if ($user1->id != $USER->id
+    && (empty($user2) || $user2->id != $USER->id)
+    && !has_capability('moodle/site:readallmessages', $context)){
+
     print_error('accessdenied','admin');
 }
```

Figure 26: Fix changes for CVE-2012-2354 [9].

(!empty($user2) && $user2->id != $USER->id) && !has_capability('moodle/site:readallmessages', $context) is an access control check, but it has a small flaw in the expression, the part *!empty($user2) && $user2->id != $USER->id* is wrong. It is a very small unique code flaw, and there exist nothing to compare with, and therefore there exist no signal or clue to trigger a security flaw warning.

4. Unauthorized addition of quiz questions (CVE-2012-2355)

This vulnerability allows remote authenticated users to bypass *question:useall* capability requirements and add arbitrary questions to a quiz [2]. In Figure 27, the line of code

```
diff --git a/mod/quiz/addrandom.php b/mod/quiz/addrandom.php
index c7f832c..32a1f2c 100644 (file)
--- a/mod/quiz/addrandom.php
+++ b/mod/quiz/addrandom.php
@@ -44,8 +44,12 @@ $scrollpos = optional_param('scrollpos', 0, PARAM_INT);
 if (!$course = $DB->get_record('course', array('id' => $quiz->course))) {
     print_error('invalidcourseid');
 }
-//you need mod/quiz:manage in addition to question capabilities to access this page.
+// You need mod/quiz:manage in addition to question capabilities to access this page.
+// You also need the moodle/question:useall capability somewhere.
 require_capability('mod/quiz:manage', $contexts->lowest());
+if (!$contexts->having_cap('moodle/question:useall')) {
+    print_error('nopermissions', '', '', 'use');
+}
```

Figure 27: Fix changes for CVE-2012-2355 [10].

require_capability('mod/quiz:manage', $contexts->lowest()) is an access control check for INSERT to table *quiz_question_instances*, but because there is no other occurrence INSERT to table *quiz_question_instances* and thus no consistency analysis could be performed, and therefore there is no clue to know one additional access control check *$contexts->having_cap('moodle/question:useall')* is required, and it is an unsolvable case.

5. Unauthorized database modifications (CVE-2012-2358)

This vulnerability allows remote authenticated users to bypass an activity's read-only state and modify the database by leveraging the student role and editing database activity entries that already exist [4]. As shown in Figure 28, to fix the flaw, it adds a declaration of a new function *data_in_readonly_period*, and used the function in multiple places, one used in the declaration of *data_user_can_add_entry()*, *data_user_can_add_entry()* is an access control related function. So the flaw lies in the access control related function *data_user_can_add_entry()*, it ignores the checking for the read-only period state. It is a unique code flaw, and there exist nothing to compare with, and therefore there exist no signal or clue to trigger a security flaw warning.

6. Unauthorized read of forums (CVE-2012-3391)

```
+    // Check whether this activity is read-only at present
+    $readonly = data_in_readonly_period($data);
+
     foreach ($records as $record) {   // Might be just one for the single template

     // Replacing tags
@@ -1264,7 +1267,7 @@ function data_print_template($template, $records, $data, $search='', $page=0, $r
     // Replacing special tags (##Edit##, ##Delete##, ##More##)
         $patterns[]='##edit##';
         $patterns[]='##delete##';
-        if (has_capability('mod/data:manageentries', $context) or data_isowner($record->id)) {
+        if (has_capability('mod/data:manageentries', $context) || (!$readonly && data_isowner($record->id))) {
             $replacement[] = '<a href="'.$CFG->wwwroot.'/mod/data/edit.php?d='
                              .$data->id.'&amp;rid='.$record->id.'&amp;sesskey='.sesskey().'"><img src="'.$OUTPUT->pix_url
             $replacement[] = '<a href="'.$CFG->wwwroot.'/mod/data/view.php?d='
@@ -2078,11 +2081,8 @@ function data_user_can_add_entry($data, $currentgroup, $groupmode, $context = nu

     } else if (data_atmaxentries($data)) {
         return false;
-    }
-
-    //if in the view only time window
-    $now = time();
-    if ($now>$data->timeviewfrom && $now<$data->timeviewto) {
+    } else if (data_in_readonly_period($data)) {
+        // Check whether we're in a read-only period
         return false;
     }

@@ -2102,6 +2102,21 @@ function data_user_can_add_entry($data, $currentgroup, $groupmode, $context = nu
     }
 }

+/**
+ * Check whether the specified database activity is currently in a read-only period
+ *
+ * @param object $data
+ * @return bool returns true if the time fields in $data indicate a read-only period; false otherwise
+ */
+function data_in_readonly_period($data) {
+    $now = time();
+    if (!$data->timeviewfrom && !$data->timeviewto) {
+        return false;
+    } else if (($data->timeviewfrom && $now < $data->timeviewfrom) || ($data->timeviewto && $now > $data->timeviewto)) {
+        return false;
+    }
+    return true;
+}
```

Figure 28: Fix changes for CVE-2012-2358 [11].

This vulnerability allows remote authenticated users to bypass intended access restrictions by leveraging the student role and reading the RSS feed for a forum [5]. The fix changes are shown in Figure 29, which looks complicated. So to put it simple, I used A, B, C, D to represent Boolean expressions which might be access control related, and illustrate the fix changes in Table 17. So the basic structure of the program before the fix is shown on the left, the basic structure of the program after the fix is shown one the right. So it is a flaw in the access control related expressions. It is a unique code flaw, and there exist nothing to compare with, and therefore there exist no signal or clue to trigger a security flaw warning.

7. Unauthorized information access (CVE-2012-5473)

This vulnerability allows remote authenticated users to read activity entries of a different group's users via an advanced search [7]. As shown in Figure 30, multiple parts of the code

```
    $formatoptions = new stdClass();
    $items = array();
    foreach ($recs as $rec) {
            $item = new stdClass();
            $user = new stdClass();
-           if ($isdiscussion && !empty($rec->discussionname)) {
-               $item->title = format_string($rec->discussionname);
-           } else if (!empty($rec->postsubject)) {
-               $item->title = format_string($rec->postsubject);
+
+           if ($isdiscussion && !forum_user_can_see_discussion($forum, $rec->discussionid, $context)) {
+               // This is a discussion which the user has no permission to view
+               $item->title = get_string('forumsubjecthidden', 'forum');
+               $message = get_string('forumbodyhidden', 'forum');
+               $item->author = get_string('forumauthorhidden', 'forum');
+           } else if (!$isdiscussion && !forum_user_can_see_post($forum, $rec->discussionid, $rec->postid, $USER, $cm)) {
+               // This is a post which the user has no permission to view
+               $item->title = get_string('forumsubjecthidden', 'forum');
+               $message = get_string('forumbodyhidden', 'forum');
+               $item->author = get_string('forumauthorhidden', 'forum');
            } else {
-               //we should have an item title by now but if we dont somehow then substitute something somewhat meaningful
-               $item->title = format_string($forum->name.' '.userdate($rec->postcreated,get_string('strftimedatetimeshort', 'langconfig')));
+               // The user must have permission to view
+               if ($isdiscussion && !empty($rec->discussionname)) {
+                   $item->title = format_string($rec->discussionname);
+               } else if (!empty($rec->postsubject)) {
+                   $item->title = format_string($rec->postsubject);
+               } else {
+                   //we should have an item title by now but if we dont somehow then substitute something somewhat meaningful
+                   $item->title = format_string($forum->name.' '.userdate($rec->postcreated,get_string('strftimedatetimeshort', 'langconfig')));
+               }
+               $user->firstname = $rec->userfirstname;
+               $user->lastname = $rec->userlastname;
+               $item->author = fullname($user);
+               $message = file_rewrite_pluginfile_urls($rec->postmessage, 'pluginfile.php', $context->id,
+                       'mod_forum', 'post', $rec->postid);
+               $formatoptions->trusted = $rec->posttrust;
            }
```

Figure 29: Fix changes for CVE-2012-3391 [13].

Table 17: Illustration of fix changes for CVE-2012-3391.

	if(C){
	}else if(D){
if(A){	
	}else{
}else if(B){	if(A){
}else{	}else if(B){
}	}else{
	}
	}
(1) Structure before the fix	(2) Structure after the fix

get fixed, for example, for the first part fixed, it added the checking for whether a student is part of the group and whether separate group is enabled, these checks are added for the first time, so before the fix, there exists nothing else to compare with, and therefore there exists no signal or clue to know that these checks should exist, so it is unique code flaws, and thus it is not solvable.

```
diff --git a/mod/data/view.php b/mod/data/view.php
index abedca4..691cac0 100644 (file)
--- a/mod/data/view.php
+++ b/mod/data/view.php
@@ -321,6 +321,13 @@
     groups_print_activity_menu($cm, $returnurl);
     $currentgroup = groups_get_activity_group($cm);
     $groupmode = groups_get_activity_groupmode($cm);
+    // If a student is not part of a group and seperate groups is enabled, we don't
+    // want them seeing all records.
+    if ($currentgroup == 0 && $groupmode == 1 && !has_capability('mod/data:manageentries', $context)) {
+        $canviewallrecords = false;
+    } else {
+        $canviewallrecords = true;
+    }

     // detect entries not approved yet and show hint instead of not found error
     if ($record and $data->approval and !$record->approved and $record->userid != $USER->id and !has_capability('mod
@@ -465,7 +472,13 @@ if ($showactivity) {
             $groupselect = " AND (r.groupid = :currentgroup OR r.groupid = 0)";
             $params['currentgroup'] = $currentgroup;
         } else {
-            $groupselect = ' ';
+            if ($canviewallrecords) {
+                $groupselect = ' ';
+            } else {
+                // If separate groups are enabled and the user isn't in a group or
+                // a teacher, manager, admin etc, then just show them entries for 'All participants'.
+                $groupselect = " AND r.groupid = 0";
+            }
         }

         // Init some variables to be used by advanced search
@@ -590,10 +603,19 @@ if ($showactivity) {
         $sqlmax     = "SELECT $count FROM $tables $where $groupselect $approveselect"; // number of all recoirds use
         $allparams  = array_merge($params, $advparams);

-        $recordids = data_get_all_recordids($data->id);
+        // Provide initial sql statements and parameters to reduce the number of total records.
+        $selectdata = $groupselect . $approveselect;
+        $initialparams = array();
+        if ($currentgroup) {
+            $initialparams['currentgroup'] = $params['currentgroup'];
+        }
+        if (!$approvecap && $data->approval && isloggedin()) {
+            $initialparams['myid1'] = $params['myid1'];
+        }
+
+        $recordids = data_get_all_recordids($data->id, $selectdata, $initialparams);
         $newrecordids = data_get_advance_search_ids($recordids, $search_array, $data->id);
         $totalcount = count($newrecordids);
-        $selectdata = $groupselect . $approveselect;

         if (!empty($advanced)) {
             $advancedsearchsql = data_get_advanced_search_sql($sort, $data, $newrecordids, $selectdata, $sortorder);
```

Figure 30: Fix changes for CVE-2012-5473 [15].