A Study of Interactive Code Annotation for Access Control Vulnerabilities

Tyler Thomas, Bill Chu, Heather Lipford
Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, North Carolina 28223

Justin Smith, Emerson Murphy-Hill
Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27606

***Abstract*—While there are a variety of existing tools to help detect security vulnerabilities in code, they are seldom used by developers due to the time or security expertise required. We are investigating techniques integrated within the IDE to help developers detect and mitigate security vulnerabilities. In this paper, we examine using interactive annotation for access control vulnerabilities. We evaluated whether developers could indicate access control logic using interactive annotation and understand the vulnerabilities reported as a result. Our study indicates that developers can easily find and annotate access control logic but can struggle to use our tool to trace the cause of the vulnerability. Our results provide design guidance for improving the interaction and communication of such security tools with developers.**

I. Introduction

Software security vulnerabilities are a leading cause for many data breaches [1], resulting in billions of dollars of records stolen. Detecting and resolving security vulnerabilities in software, especially later in the development cycle, can be both time-consuming and expensive. Static analysis techniques can help developers detect vulnerabilities early in the development process — even before executing the code. There are many widely used research and commercial static analysis tools available [2]–[7]. However, these tools are underused [8] in part because of their high false positive rates [9], and the need for security expertise to write customized rules to reduce those false positives.

Our goal is to help developers address security concerns and reduce security vulnerabilities while they write code. We are examining techniques for helping developers detect and mitigate security issues within the Integrated Development Environment (IDE). We refer to these techniques as **interactive static analysis** [10]. We have previously prototyped and evaluated an interactive static analysis tool named ASIDE (Application Security in the IDE) for basic vulnerabilities such as SQL Injection and Cross Site Scripting. We demonstrated that providing warnings and explanations to developers alongside their code improves awareness of these security vulnerabilities and how to prevent them [11].

We are now expanding our approach to include **interactive annotation**, where developers are prompted to indicate security-critical components in the code, both to remind them to perform security actions and to document application-specific security information. This in turn allows a static analysis tool to reason more accurately about the code and detect more complex vulnerabilities. In our first prototype of this approach, we are examining access control decisions.

Access control vulnerabilities have been consistently ranked as among one of the top security vulnerabilities in applications [12]. These vulnerabilities result from program logic errors made by developers, resulting in missing or inconsistent access control checks for sensitive operations [13]. Our tool asks a developer to annotate the access control logic for sensitive database operations, prompting developers to review the access control code, and is then used for additional static analysis. We examined the performance of our approach on vulnerability detection for 6 open source projects, and found that we detected more vulnerabilities than existing automated approaches, with significantly less work for users than commercial tools require [13]. However, our approach depends on developers correctly performing the annotations, and understanding the resulting vulnerability warnings. Thus, we now examine this interaction.

We report on a user study of ASIDE's interactive annotation, with the following objectives: evaluate the usability of our interface, and developers' behaviors in annotating code; examine how developers identify and understand access control logic within code; and examine how developers interpret and understand vulnerability warnings that result from their annotations. Our results will help to improve the interface of our tool, as well as provide a deeper understanding of how such tools can communicate with developers regarding security vulnerabilities generally, and access control more specifically. Future commercial implementations of interactive static analysis and interactive annotation with greater usability will ultimately result in fewer security vulnerabilities.

II. Related Work

The use of code annotation has been explored in a variety of research in order to improve program efficiency (e.g. [14], [15]), document code, or make the code easier for people to understand (e.g. [10], [14]–[16]), or to identify errors or problems during development (e.g. [16]–[18]). Our work falls under the last goal. However, instead of using interactive annotation to identify errors, we are using it for annotating security-related decisions to aid in vulnerability detection.

Previous research has explored the use of alternative types of code annotation in vulnerability detection. Qui et al. propose an annotation toolkit, where developers make annotations in the form of API calls [18]. At runtime, these calls are then capable of detecting denial of service attacks and other resource abuses. There are also several commercial annotation languages for assisting static analysis tools in detecting bugs, including security vulnerabilities [19]. The security analysis tool FindBugs also allows the developer to textually annotate their code to improve the accuracy of the analysis [14]. FindBugs can detect SQL injection, cross site scripting, hard-coded database passwords, and the creation of a cookie from untrusted input.

Despite the variety of uses of code annotation, the majority of annotations utilize text. In other words, developers adding annotations do so by adding additional comments or code during development. This means that developers must learn an additional annotation language, and remember how and when to use it in order to complete the annotations. We believe this added burden will limit the use and effectiveness of any solutions relying on code annotation. Developers are unlikely to be motivated to put in significant upfront effort to learn a new language to assist with security.

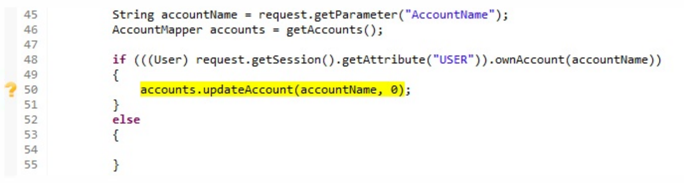
III. ASIDE

Application Security for the Integrated Development Environment (ASIDE) is a plug-in for the Eclipse PHP and Java Development Environments [10], and currently designed for Web applications. ASIDE generates requests for its users to associate annotations with security sensitive operations. In [10] we detail how ASIDE generates requests for, and detects access control vulnerabilities using these annotations. Annotation requests are indicated by a yellow highlight of the sensitive code, and a yellow question mark alongside the code (Figure 1). Clicking on the icon or code provides a menu where the developer can access ASIDE explanations, as well as choose to enter annotation mode.

In annotation mode, the developer highlights the statements performing access control for the sensitive operation. In doing so, the developer is reminded to add such checks, if they are not already implemented. An access control check is a Boolean condition, or a call to a function that throws an exception. ASIDE indicates the annotation with a green highlight and a small green diamond next to the code. The sensitive operation also turns green when an annotation is added, with the icon changing to a green check mark. Once the annotation is made, ASIDE leaves annotation mode and the developer returns to the task of coding. With the annotations provided by the developer, static analysis is used to detect vulnerabilities, which are then indicated with a red icon next to the sensitive operation. Our vulnerability detection algorithms and performance are detailed in [13]. ASIDE's current interface has been iteratively developed based on the lessons learned through our use with several open source projects as well as a small formative study [20]. We now report the results of a full user study on interactive annotation and their implications for ASIDE and similar security tools.

IV. Methodology

We recruited participants from advanced programming classes offered at the University of North Carolina at Charlotte and North Carolina State University. We chose to use students



```
String accountName = request.getParameter("AccountName");
AccountMapper accounts = getAccounts();

if (((User) request.getSession().getAttribute("USER")).ownAccount(accountName))
{
    accounts.updateAccount(accountName, 0);
}
else
{

}
```

Fig. 1. Annotation request in Gold Rush, shown with a yellow highlight and question mark icon.

from these classes since it is often difficult to obtain a large sample of professional developers for academic research. Additionally, since most of these students are either almost finished with an undergraduate degree or pursuing a graduate degree, we feel that data obtained from them may be similar to that obtained from entry level professional developers.

Participants from UNC Charlotte interacted with ASIDE running on a project called Gold Rush, an internally developed Java-based banking application (99 files) to teach web application security. Participants had just completed a class assignment involving this code, and were thus familiar with it. Participants from NC State interacted with ASIDE running on a project called iTrust, an open source medical information system (1,860 files). Participants had previously fixed real bugs and added additional features to iTrust over the course of a semester in a senior-level undergraduate software engineering course. We intentionally chose code which would be somewhat familiar to students to simulate the situation of a developer working within their own projects.

For each project, we created a set of scenarios for participants to examine. The code of both projects was modified slightly so that we could induce several additional access control vulnerabilities. To simplify the tasks and time involved, we ensured that each scenario only involved code in one file, and we only showed requests and vulnerabilities for our scenarios (and suppressed any others ASIDE would normally identify). These scenarios included:

- Requests for access control logic, requiring annotation. All scenarios contained one piece of access control logic. However, some scenarios were very simple while others varied in complexity and contained much more code between the start of the file and the sensitive operation.
- Request for access control logic, but where the logic was missing from the code.
- A false annotation request - where an access control check was unneeded. We provided one false annotation request to observe how users would react.
- Vulnerability warning as a result of missing or incorrect access control in the code.

Participants were first given a brief introduction to ASIDE, and allowed to interact with a trainer example before beginning the study. Participants were then shown the files with the requests and warnings, in the same order. For every annotation request participants were asked to "annotate the access control logic, if it exists, for this particular annotation request or warning." When participants made the annotation, they were asked why they chose a particular line or lines of code as the access control logic. They were also asked about the meaning of warnings, and how they would fix any issues they identified. When completed, they were then asked several questions about their perceptions and use of ASIDE. We recorded the audio and screen activity during the session. Data was analyzed by transcribing the audio and creating notes based on the screen recordings. The primary author performed open coding on the transcriptions and notes, to determine performance and look for common patterns and interesting responses.

V. Results

We had 28 participants – 13 (9 male, 4 female) were from UNC Charlotte and examined the Goldrush system, and 15 (14 male, 1 female) were from NC State and examined iTrust. 21 were undergraduate students in computing, while the rest were Master's students. Most participants had taken 1 to 3 security courses and around 5 courses that included programming as a major component. 6 participants reported that they had worked as a professional developer.

A. *Interactive Annotation*

Our 28 participants encountered a total of 125 annotation requests and 56 warnings. Users felt our interface was very intuitive and that annotations were very easy to make (n=26) as they were able to complete an annotation in all but 1 instance. By inspecting the screen recordings, we determined that participants annotated the right access control checks or correctly identified reasons why an annotation was not necessary in 141 of the 181 (78%) annotation requests and warnings. We investigated whether there were differences in accuracy between those with more or less programming experience, or more or less security experience, but did not find any significant differences. This is actually encouraging because it means that even participants with lower programming experience were still able to use ASIDE effectively, although we did anecdotally observe that the more experienced programmers spoke more confidently.

There were a variety of incorrect annotations, such as variable declarations, the function *request.getParameter()* since it is where sensitive data enters the program, or the "try" of an encapsulating try block. Two participants also annotated the sensitive operation instead of the access control checks. ASIDE could reduce such confusion by either indicating potentially valid annotations or checking annotations after they are made (e.g. code that actually contains a Boolean conditional statement). The study prototype did not perform verification.

In fact, three participants expressed confusion about whether or not ASIDE could verify their annotation. The green color and green check that appeared after an annotation was complete seemed to indicate a false sense that the annotation was being checked by the system and was found to be correct:

> "I'm gonna go ahead and annotate it, and we'll see if that makes it happy."

In order to detect vulnerabilities, our tool requires that users highlight a Boolean conditional statement or a function that throws an exception. Participants generally highlighted either entire lines or small blocks of code containing such statements, such as the entire if code block containing the conditional statement. Our current prototype can not yet handle this behavior with sufficient flexibility. This implies that our tool will either need to help users identify and highlight just the smaller snippet of access control logic, or will need to parse that logic out of the larger block of code that users identify.

Interestingly, several participants saw the requests for annotation as requests for "security checks" in a general sense and would search for validation code. When asked why they chose to annotate certain pieces of code, they would talk about how the code might be vulnerable to cross site scripting attacks or SQL injection and how the annotated code fixed that problem. Yet those issues would be corrected by validating untrusted input, rather than an access control check. Surprisingly, many of these participants actually still annotated the correct access control logic. Thus, these participants had some generic security knowledge, and while they were confused as to why, they did equate the access control logic they found in the code with somehow providing security.

B. *Interpretation of Warnings*

27 of 28 participants (96%) understood that red warnings were indicating a possible security vulnerability. It means that the tool was effectively able to communicate this information to participants. Five participants did not give confident answers as to whether or not a sensitive operation with a warning was actually vulnerable (a true positive). We saw an interesting pattern, where participants who expressed confusion discussed "how" vulnerable a piece of vulnerable code was, rather than simply whether the code really was or was not vulnerable. The following responses illustrate this mental model of vulnerabilities:

> "I think there is a degree of vulnerability to it."

The tool was less successful at helping participants understand why a warning occurred, and thus how to mitigate the vulnerability. Participants were generally good at identifying the cases where access control logic was missing in the code. However, they did not realize that their previous annotations were related to a warning, and that they should examine the annotations for the same database operations to identify mismatches that may be causing the vulnerability. One reason is that only four participants (14%) looked at the contextual help ASIDE provided regarding a vulnerability. Instead, our participants simply looked at the code to determine the validity and fixes for warnings.

C. *User Perceptions and Comments*

General user impressions were overwhelmingly positive, with only one participant responding negatively. When asked, no participants felt that the process was tedious. Multiple participants expressed concern that the task of interactive annotation could become tedious on large projects, but they also felt that more false positives and higher detection rates were preferred over fewer false positives and lower detection rates.

> "That made me stop and think. When I'm in a developer role, I'm always thinking about the functionality because that's what I get paid to deliver.It forced me

to stop and think about security even if just for a few minutes. It was good."

Additionally, all participants except one stated that they would use ASIDE in the real world. Six participants explicitly or implicitly recommended that ASIDE be further developed into a code review tool. One participant even suggested that it would be nice to be able to mark confusing annotation requests or warnings with a note for another developer to annotate or address later.

These user comments are particularly interesting in that they reflect our envisioned future work. In our previous work, we examined the possibility of automatically generated annotations and concluded that automatic annotation is insufficient on its own and a hybrid approach involving a human is required [13]. However, we did not communicate this information to any of the participants of this study.

VI. Discussion

Not surprisingly, graphically highlighting code in order to make an annotation was an easy task. This should ease adoption of any mechanism requiring annotations. One improvement to the interface could be finding an acceptable way to modify the icon for annotation requests. For some users, the color yellow implied caution or warning, instead of a request for information which we were trying to convey. This may have led to the mindset of certain operations (those in yellow) being "less" vulnerable than others (those in red). One participant suggested the color purple in place of yellow, although there is no standard color equated with information requests. Perhaps the icon itself can be modified in some way to represent a request for information instead of caution or warning.

Some participants annotated code which could never function as access control logic, such as variable declarations. Accuracy may be improved if ASIDE could provide feedback on what is potentially annotatable if possible, such as through highlighting, or autocompleting highlighting that has been started. Several of our participants also felt that the green icons were somehow indicating approval of their annotations. We need to investigate efficient algorithms to provide such feedback and validation before or during the act of annotation, which can help users who did not fully understand access control, and ensure that annotations are all valid. This would also help the tool to accurately perform vulnerability detection.

Even those who did not fully understand access control did a reasonable job of responding to requests and annotating the code. They also did a reasonable job of noticing when access control was missing. This is a positive and encouraging result, as it implies that many developers will be able to accomplish this task, and validates that our approach could be effective for a range of developers. Perhaps if annotations could be reviewed by a human with security knowledge, developers could increase their understanding of access control based on feedback from the human.

First, only four participants read the contextual help for a warning, so clearly that needs to be made more prominent and part of the interaction with ASIDE, and not just look like interface help. Second, our tool should more clearly convey the correct mental model of what a warning means, and what actions should follow. For example, one participant reported that the color red seemed to suggest an error. While vulnerabilities are potentially serious issues, they are very different from errors and will not stop program compilation. In addition, our warnings are about potential vulnerabilities. A human must still determine whether or not a true vulnerability exists based on a detailed examination of the code. Again, ideally our tool should help developers make this determination as much as they are able, and not simply imply that there is a problem.

Users also did not connect their previous annotations with subsequent warnings. A warning should imply that what is annotated may somehow be wrong - usually that the access control code that was annotated needs to be corrected - some logic is missing or wrong. So, users should examine previous annotations for all instances of that operation when there is a warning. Tracing the cause of a vulnerability would involve comparing the access control logic across those operations. Helping users to do that is a challenging task, and clearly where we need to focus on creative design solutions. The tool interface needs to make that more prominent, and easier to perform. The explanations regarding the warning that were shown in the side dialog of the menu were ignored. A modification to the interface that could visually show a link between the warning and relevant annotations within the code window itself may prove beneficial.

VII. Conclusion

We believe that developers can be provided with tools to better enable them to detect and mitigate security vulnerabilities, enhancing the security of their applications. Thus we are investigating how to communicate and interact with developers regarding security vulnerabilities so that such tools are usable and effective. As we have demonstrated, developers could benefit from such tools with greater awareness of the security implications of their code and potential vulnerabilities. Even those with lesser programming and security experience were able to indicate security-related decisions in the code, thus providing valuable information to drive more complex analysis or for use in later code review. Our participants were appreciative of the awareness of security that our tool provided. Providing annotations interactively through highlighting was intuitive, yet also requires more flexibility from the tool in allowing users to highlight larger chunks of code than are needed. Our results provide valuable feedback into our tool design, and in particular highlight the challenge of helping developers trace and fix complex vulnerabilities and their relationship to the annotated security discussions. This study can serve as a baseline for additional examination of interactive annotation interfaces in security tools, and we hope to use our results to inform the design of future interfaces for interactive annotation.

Acknowledgment

This work was partially supported by NSF grants 1129190, 1318854, and DOE award number P200A130088. We would also like to thank Jun Zhu and Mahmood Mohamadi for their work on the ASIDE implementation.

References

[1] [Online]. Available: http://www.eweek.com/security/software-vulnerabilities-lead-to-internal-security-problems-kaspersky.html

[2] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 931–940. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486915

[3] A. K. Tripathi and A. Gupta, "A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for java programs," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 23:1–23:4. [Online]. Available: http://doi.acm.org/10.1145/2601248.2601288

[4] "Findbugs," 2015. [Online]. Available: http://findbugs.sourceforge.net/

[5] "Fortify," 2015. [Online]. Available: http://www8.hp.com/us/en/software-solutions/staticcodeanalysissast/

[6] "Codepro," 2015. [Online]. Available: https://developers.google.com/javadevtools/codepro/doc/?hl=en

[7] "Pmd," 2015. [Online]. Available: https://pmd.github.io/

[8] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1109/MS.2008.130

[9] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[10] J. Zhu, J. Xie, H. R. Lipford, and B. Chu, "Supporting secure programming in web applications through interactive static analysis," *Journal of Advanced Research*, vol. 5, no. 4, pp. 449–462, 2014.

[11] 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*, ser. CHI '12. New York, NY, USA: ACM, 2012, pp. 2707–2716. [Online]. Available: http://doi.acm.org/10.1145/2207676.2208665

[12] "Top ten vulnerabilities," 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10

[13] J. Zhu, B. Chu, H. Lipford, and T. Thomas, "Mitigating access control vulnerabilities through interactive static analysis," in *ACM Symposium on Access Control Models and Technologies*. ACM, 2015.

[14] M. Berón, P. Henriques, M. J. Pereira, and R. Uzal, "Static and dynamic strategies to understand c programs by code annotation," 2007.

[15] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "Dyc: an expressive annotation-directed dynamic compiler for c," *Theoretical Computer Science*, vol. 248, no. 1, pp. 147–199, 2000.

[16] U. Dekel and J. D. Herbsleb, "Pushing relevant artifact annotations in collaborative software development," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*. ACM, 2008, pp. 1–4.

[17] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *FME 2001: Formal Methods for Increasing Software Productivity*. Springer, 2001, pp. 500–517.

[18] X. Qie, R. Pang, and L. Peterson, "Defensive programming: Using an annotation toolkit to build dos-resistant software," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 45–60, 2002.

[19] "Using sal annotations to reduce c/c++ code defects." [Online]. Available: http://msdn.microsoft.com/enus/library/ms182032.aspx

[20] H. Lipford, T. Thomas, B. Chu, and E. Murphy-Hill, "Interactive code annotation for security vulnerability detection," in *Proceedings of the 2014 ACM Workshop on Security Information Workers*, ser. SIW '14. New York, NY, USA: ACM, 2014, pp. 17–22. [Online]. Available: http://doi.acm.org/10.1145/2663887.2663901