What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool

Tyler W Thomas, Heather Lipford, and Bill Chu
Department of Software and Information Systems
University of North Carolina at Charlotte
9201 University City Blvd
Charlotte, North Carolina

Justin Smith and Emerson Murphy-Hill
Department of Computer Science
North Carolina State University
890 Oval Drive
Raleigh, North Carolina

ABSTRACT

Security vulnerabilities are often accidentally introduced as developers implement code. While there are a variety of existing tools to help detect security vulnerabilities, 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 previous work, we examined the questions developers ask when investigating security vulnerabilities with static analysis tools. With those questions as a lens, we now investigate our proposed approach of interactive static analysis. We evaluated the interactions and perceptions of professional developers as they interacted with warnings produced by our tool. Our results provide evidence that our approach effectively communicates security vulnerability information to software developers and provides design guidance for such tools.

1. INTRODUCTION

Security problems are a large and growing concern today [9]. At the heart of this problem are software security vulnerabilities [6], bugs in code that can lead to security attacks. Detecting and resolving these types of problems, especially later in the development process, is costly and time consuming. Static analysis techniques can help developers detect vulnerabilities early in the development process — even before executing the code. However, static analysis tools are underused [2] in part because of their high false positive rates [4], 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** [15]. 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 both these security vulnerabilities and how to prevent them [13]. We expanded 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. We performed an evaluation of this technique with students and found that they responded positively to the interface and could make the correct annotations with good accuracy [11].

Our previous studies emphasized the need for clear and contextual communication with developers regarding security vulnerabilities and their mitigations. In this paper, we extend our previous work by focusing on that communication and how it addresses the questions that developers have regarding security vulnerabilities. We also performed this evaluation with professional developers instead of advanced students. Our results demonstrate that clearly linking vulnerability information to the code enables developers without extensive security expertise to understand and mitigate such vulnerabilities.

2. RELATED WORK

Much research has been conducted to determine the effectiveness of various tools which aim to detect and remove vulnerabilities. Static analysis tools are the most commonly used technique, comprising roughly half of the entire market share for security tools [3]. Static analysis techniques detect vulnerabilities in non-running or "static" source code by looking for predefined patterns. However, due to high false positive rates, static analysis tools are seldom used [2] by regular developers.

Attempts have been made to improve the false positive rate and developer usage of these tools. For example, Sadowski et al. designed a tool called Tricorder for use within Google, combining code reviews with static analysis. To keep false positives low, Sadowski et al. allowed developers to write their own static analyzers and implemented them on a project

Symposium on Usable Privacy and Security (SOUPS) 2016, June 22–24, 2016, Denver, Colorado.

Table 1: Groups and Categories of Questions Developers Asked, from [10]

Group	Category
Vulnerabilities, Attacks, and Fixes	Preventing and Understanding Potential Attacks
	Understanding Alternative Fixes and Approaches
	Assessing the Application of the Fix
	Relationship Between Vulnerabilities
Code and the Application	Locating Information
	Control Code and Call Information
	Data Storage and Flow
	Code Background and Functionality
	Application Context and Usage
	End User Interaction
Individuals	Developer Planning and Self Reflection
	Understanding Concepts
	Confirming Expectations
Problem Solving Support	Resources and Documentation
	Understanding and Interacting with Tools
	Vulnerability Severity and Rank
	Notification Text

level, removing any analyzers with false positive rates that were too high. As a result, developers widely used the tool [8]. Livshits and Lam also designed a tool with emphasis on reducing the amount of false positives. They created static analyzers based on input from users. They also found their tool to be effective, which they attributed to the low false positive rate [7]. Lastly, Jovanovic evaluated a more specialized static analysis tool called Pixy. Pixy was designed purely to tackle cross site scripting vulnerabilities in one language, PHP. Jonavovic ultimately concluded that it was an effective tool because it had a very low false positive rate and was also effective in detecting zero day vulnerabilities [5].

We took a different approach, exploring how to raise the knowledge and practice of software developers in secure programming. We proposed interactive static analysis, where developers interact with a tool during the development process to detect and mitigate a range of security vulnerabilities [12]. The following section provides details on our current prototype.

All of the above research has focused on the effectiveness and perception of the tool under exploration. However, in our previous work we more deeply explored the kinds of questions developers ask when using static analysis tools to detect security vulnerabilities [10]. This study examined developers as they interacted with a commercial static analysis tool, FindBugs. The qualitative, exploratory study identified 17 categories of questions regarding understanding vulnerabilities and their fixes, understanding the code context such as the control and data flow, and problem solving and reasoning support. These categories, summarized in Table 1, highlight the range of issues that developers face when using tools to address security vulnerabilities. The questions now provide us with a lens for analyzing the current evaluation. We examine how interactive static analysis helped address the vulnerability questions raised by our participants.

3. ASIDE

Application Security in the IDE (ASIDE) is a prototype interactive static analysis tool, where developers are provided with assistance in detecting and mitigating security vulnerabilities as they write code [11, 13]. The goal is to raise developer awareness and behavior of secure programming in their code, without requiring a security background. The current prototype is an Eclipse plugin for both Java and PHP. Static analysis algorithms are run on the code under development to detect a variety of potential issues. Warning icons are then placed in the left margin of the code editing window (see Figure 1), with interactive options for understanding and mitigating vulnerabilities.

```
51      logger.trace("Entering doGet()");
52      String accountName = request.getParameter("AccountName");
53
54      int newBalance = Integer.valueOf(request.getParameter("NewBalance"));
55      if (request.getSession().getAttribute("USER") == null) {
56         logger.warn("User not authenticated");
57         response.sendRedirect(request.getContextPath() + "/login.jsp");
58      } else if (((User) request.getSession().getAttribute("USER"))
```

Figure 1: Two ASIDE vulnerability warnings

ASIDE currently works on two types of potential vulnerabilities. Type I are vulnerabilities that are independent of the code context. These are vulnerabilities traditionally found by static analysis tools. We currently support input validation, output encoding and SQL injection vulnerabilities. ASIDE provides short descriptions alongside the warning, and a "Read More" option for detailed and contextualized descriptions of the potential vulnerability, ways to mitigate, and example code. In addition, we have added "quick fixes" for automatically generating sanitization code for input validation and output encoding warnings using ESAPI's open source libraries [1]. Figure 2 shows a screenshot of the warning notification and quick fix options for an input validation vulnerability, while Figure 3 shows the high level "Read More" page for a SQL Injection vulnerability. Based on feedback from previous evaluations of ASIDE, we have made all notifications and communications as contextualized as possible, including lines of code or variable names where appropriate. For this study the text was hard-coded to ensure

accuracy, and we will fully implement such notifications in the future based on study feedback.

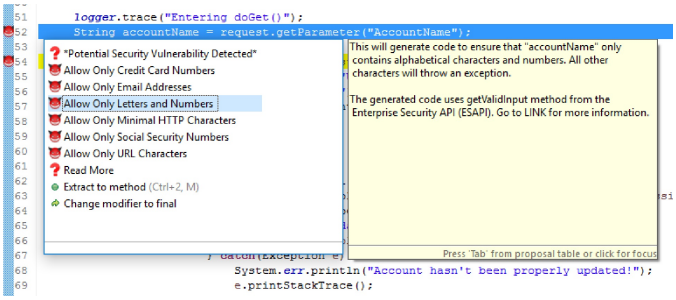

Figure 2: ASIDE menu, showing possible quick fixes for an input validation warning

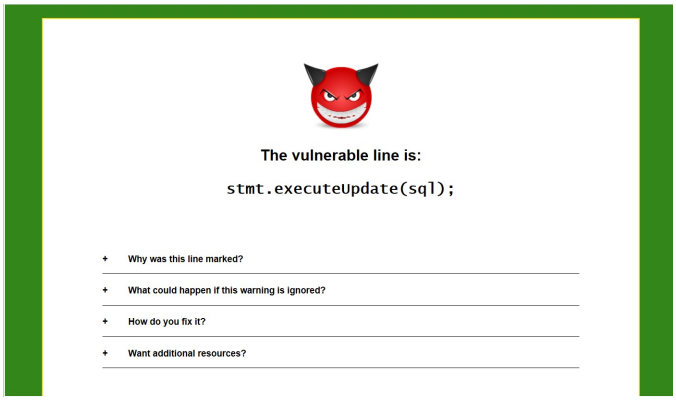

Figure 3: Read more contextualized help page for SQL injection warning

Type II vulnerabilities require additional application-specific knowledge in order to detect potential issues. In these cases, developers are first requested to provide that knowledge, in the form of an annotation performed by highlighting code. We currently support access control vulnerabilities. When ASIDE detects security-sensitive database operations, developers are requested to annotate the corresponding access control logic. This request is indicated by a yellow highlight of the sensitive code, and a yellow question mark alongside the code. We chose a question mark to convey that the tool is requesting information, not indicating anything wrong with the code. Thus, in the example in Figure 4, the developer was asked to indicate where access control checks are located for the function *updateAccount* that accesses sensitive database tables. 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 would then highlight the statements performing access control for the sensitive operation, highlighted in green in Figure 5. In doing so, the developer is reminded to add such checks, if they are not already implemented. ASIDE indicates the annotation with a green highlight and a small green diamond next to the code.

ASIDE then performs additional static analysis to detect potential access control vulnerabilities, signifying them with the same red warning icon as Type I vulnerabilities.

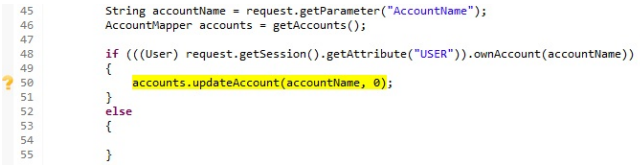

Figure 4: Annotation request for a security-sensitive operation, shown with a yellow highlight and question mark icon.

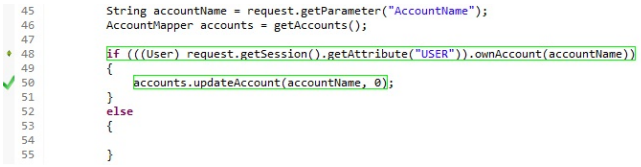

Figure 5: An annotation and annotation request that have been completed.

We have previously evaluated ASIDE's security performance, demonstrating its abilities to detect all currently supported classes of vulnerability [14, 12]. In addition, we have performed several user studies with our evolving versions of ASIDE [11, 13, 15]. Previous studies have mainly utilized students, and demonstrated that advanced students can understand ASIDE and appreciate the awareness of security vulnerabilities that ASIDE provides. However, they are also wary of implementing ASIDE's recommended fixes due to fears of breaking their functional code. We have continued to improve the communication and interaction with ASIDE based on these evaluations. In this study, we focus on professional software developers, and specifically whether and how ASIDE's design can effectively communicate vulnerability information.

4. METHODOLOGY

As a follow up study to our previous work, we designed an interview and observation-based study to examine software developer reactions and perceptions to interactive static analysis. Specifically, this included the communication of vulnerability information provided by our tool ASIDE. We recruited professional software developers using a snowball sampling technique. We recruited from a variety of sources, including developers we knew as well as local software development groups. We then asked participants to recommend additional participants. To compensate developers for their time, we provided a $25 gift card after the study.

We recruited a total of thirteen participants with professional development experience. Eight of our participants hold a job title of software engineer, or java or software developer. Other job titles include systems engineer, software tester, and security engineer. Participants had an average of 7.73 years (SD 7.17) of professional programming experience. Three of our participants were female; ten were male.

We collected our demographic data at the end of the study to remove any possibility of the survey biasing the participant's performance. As a part of this survey, we also asked participants to rate their programming knowledge and security skills on a scale of one to ten. Participants responded with an average of 7.92 (SD 1.38) for programming skills and

5.31 (SD 2.84) for security knowledge. The high ranking for programming skills was not surprising, considering all of our participants had professional software development experience. What is perhaps more surprising is that the security knowledge rating was moderate instead of poor. This suggests that our participants did have some form of security training or education, or, at the very least, had encountered security sensitive situations while coding.

Participation in the study was done remotely. Participants would call the evaluator on the phone, and login to our computer through the use of remote desktop software. Participants then interacted with our tool, running on our computer, through this software. Participants interacted with ASIDE running on a project called Gold Rush, an internally developed Java-based banking application (99 files) used to teach web application security. Participants were first given a brief introduction to ASIDE. They were then shown and allowed to interact with a trainer example, which consisted of one input validation warning. The purpose of the example was to answer any questions they had before beginning, and to account for Eclipse interface quirks which could not be removed or fixed in our current prototype.

Participants were then given 6 tasks - to view and respond to four vulnerability warnings (2 input validation, 1 output encoding, and 1 SQL injection) and 2 annotation requests produced by ASIDE. The second input validation task was more complex. In this instance, unsanitized input entered the program, but was immediately parsed to a double, meaning malicious code would not be able to enter the system. However, this would result in an uncaught exception being thrown. In order to standardize the study, ASIDE was altered to only show these 6 warnings, and suppress any additional warnings. All participants saw the warnings in the same order.

After showing each participant each warning, we asked the participant to interact with ASIDE and tell us how they would respond. After investigating the vulnerability and choosing a "quick fix" if one was available, we then interviewed participants regarding their understanding of the vulnerability and their reaction to ASIDE. After visiting all of the warnings, if they had not visited the "Read More" pages, we then prompted them to do so. Once they had observed all of the warnings, we showed them a very brief example of an annotation request and an annotation. Then, the participant was asked to respond to two different annotation requests, and provide their comments. For the first annotation request, access control logic was in the code and available to be highlighted. The second request was a false positive - access control logic was not necessary, and was thus not present in the code.

The phone call was audio recorded, and screen recording software was used to capture the activities of the participant on screen. The questions from the previous study on developer questions (See Table 1) were used for the first round of coding. Specifically, the primary author coded all responses and behaviors based on the question that it best related to. Below, we summarize our results based on this question categorization. The primary author also did an additional round of open coding to determine performance and look for any additional common patterns and interesting responses.

5. RESULTS

Our goal was to understand how interactive static analysis helps developers in answering their questions regarding security vulnerabilities. Our results will also help inform the evolving design of our own tool, ASIDE.

Developers had no major challenges with understanding and interacting with ASIDE. They were also quite accurate in their choices for addressing the warnings they saw. All thirteen developers chose the correct quickfix for the first input validation warning. Ten developers either correctly identified the second input validation warning as a false positive or provided a correct solution for sanitizing the input and handling the exception. Eleven out of thirteen developers chose the correct solution for the output encoding warning. All thirteen developers also correctly discussed how they would implement the solution to the SQL injection warning. Twelve out of thirteen developers successfully completed the first access control annotation task. However, only eight out of thirteen responded to the annotation false positive correctly. Others highlighted the incorrect access control logic, corresponding to a later sensitive operation.

As with our previous evaluations, participants responded positively towards using the tool and the additional awareness it provides:

"I'm not a security expert. It offers a different perspective, so I kind of felt like one while I was using it, just because it helps me recognize those things. It's also training too. I think it would be really valuable for developers to be using, especially if it's detecting vulnerabilities as they write them." -p4

We categorized the remaining behavior and perceptions of developers based on the relevant set of questions from Table 1 and concerns they were addressing at the time. We present only the most relevant question *categories* we found below, in order of how participants stepped through their tasks.

5.1 Notification Text

Questions in this category address how the notification related to the code. The first interaction developers have with ASIDE is clicking on the warning icon to view the notification text. Developers appeared to have little difficulty understanding the source of the warning - what in the code was triggering ASIDE. Integrating the warnings within the code, and further contextualizing the text of the explanations, reduced developer questions. Developers also commented positively about the contextualized text. Their performance in using the ASIDE tool to choose the right solution also indicates that the text was effective. Developers did often reread the text from the warnings to try to fully determine what was meant by the wording. They sometimes suggested a standard template response with one section that explains why the line was flagged and another for explaining the consequences if the issue is left alone. However, they indicated little difficulty in understanding the purpose of the warnings:

"The nice thing about these tools is it brings it to your attention. It could be thousands of lines of code here. If I looked at that one line of code for a long time I'd probably see it, but that's the nice thing about the tool. It gets you focused to that spot." -p8

5.2 Preventing and understanding potential attacks

Questions in this category involve understanding why there is a vulnerability and what kinds of attacks could occur as a result. Once our participants understood the warning, they then sought to determine the cause and whether it was instead a false positive. Developers did not seem to question why the warning was triggered, but instead focused on whether the issue was actually exploitable. In 21 instances, 8 developers explicitly or implicitly indicated that whether or not code was vulnerable depended on whether or not a perceived exploit was possible. If no exploit was observed or the developer determined that an exploit was not possible, the developer did not consider the code to be vulnerable. In other words, they seemed to believe the tool – that there was potential for a vulnerability. But they then examined the larger code context to determine if that issue could result in a security problem. This led to some developers pronouncing some vulnerability warnings as false positives, since they did not appear exploitable. For example:

"It might not be the most effective cross-site scripting but in this case I feel like just because of the way current users initialize, it's not really that big of an issue." -p13

"If it's not sanitized it's known as a XSS attack. It's going to depend on the implementation of account."-p1

This phenomenon was particularly noticeable in the case of output encoding warnings. For this warning, unsanitized output was printed to a webpage. Two developers indicated that this was not a vulnerability since the data should originate from a trusted source. The developers explicitly stated that input validation should be performed where data originates, and once in the system it should be considered trusted. This caused them to take no action and pronounce the warning to be a false positive.

"I would think that we would take care of this somewhere else, I would think we wouldn't want any kind of injected, we would validate the user name at the time when they select it, not here." -p3

"I have to say my initial gut feel when looking at this code would be that its a more of false positive type of an issue. Because, I would have assumed they would have been checked in when I got the user."-p6

However, secure programming best practices still advocate for sanitization of such output. If input validation was somehow not performed at a taint source due to an unknown vulnerability or due to unknown data flows in the system, the code would then be exploitable in a cross site scripting or log poisoning attack.

Developers actively tried to understand attack vectors. Although only 3 discussed the perspective of the attacker, in 8 cases developers evaluated a warning based on where possibly malicious content could enter the program. ASIDE generally places warnings where the vulnerability is best addressed. This means that for input validation warnings, ASIDE places the warning on the line where data enters the program. This type of warning may have helped developers identify attack vectors.

"The issue here is that they would put some code in the username and they would evaluate that."-p3

However, ASIDE did not provide concrete information on specific exploits, and this was a frequently requested addition to the "Read More" help pages.

When faced with annotation requests, developers sometimes asked questions about the purpose of the annotation. Four developers did not fully understand the reason for creating annotations. We did not mention access control vulnerabilities to participants, and it is possible that many did not understand what they were. It is also possible that they did not understand how the tool could use the annotations to detect vulnerabilities, particularly since we did not actually provide a warning of an access control vulnerability. Some seemed to view annotation more as a task they had to do for the tool, rather than providing input to the tool to help it detect issues. Thus, while participants were capable of performing the annotation task, they were not able to reason about the potential for access control vulnerabilities as well as they could the injection-based vulnerabilities. One implication is that we need to further improve the explanations surrounding the need for annotation and its purpose.

"I was confused as to whether or not I was actually checking anything or just simply annotating it."-p2

5.3 Vulnerability Severity and Ranking

Unlike some other static analysis tools, ASIDE does not provide any severity or confidence ratings for the warnings. However, developers judged the severity of the warning based upon the perceived severity of potential exploits. SQL injection was almost always seen as particularly severe. Cross site scripting was, on the whole, perceived to be less severe than SQL injection:

"I'm not sure cross site scripting is on top of my kinda security issues list in this case, i'd be more worried about this actually just working." -p13

In discussing severity, participants desired worst case examples to help them reason about the seriousness of the various warnings they encountered.

"I wished it would give examples of what the insertion of malicious code/ XSS what it might do. If you say to someone, someone could go through the stop sign, that's one thing. If you say someone could go through the stop sign and get killed, you see it as a different severity." -p8

While secure programming best practices recommends developers fix all vulnerabilities, participants clearly desired knowledge of how serious issues were in order to prioritize their changes. ASIDE did not help them judge this severity, but could easily include more detailed exploit information in the Read More explanations.

5.4 Understanding Alt Fixes and Approaches

Questions in this category related to understanding and comparing the alternatives for fixing the vulnerability warning. For the input validation and output encoding vulnerabilities, ASIDE provided a list of "quick fixes," generating code that sanitized the data based upon the type of input or output. For SQL injection, the detailed explanation advocated using Prepared statements, and provided examples, but did not automatically generate the code. Because of this support,

participants did not research additional alternatives outside of ASIDE, even though links for more information were provided in the detailed help pages for each warning. Developers seemed to understand the menu options, although there was confusion over the output encoding options of "Sanitize HTML" and "Sanitize URL." Some participants wanted to choose both. Thus, the encoding options were not as self-explanatory as the input validation options, and still need better explanation of how to choose between them.

Developers did carefully consider which options were the best fit for the code. For example, participants were asked to validate input for a username. ASIDE offered the option to "Allow Only Letters and Numbers", which was the best choice. However, participants who felt that usernames should contain only letters desired an "Allow Only Letters" option. Participants wanted the sanitization to be very specific to the type of characters which were valid in the given context, and were bothered by not having the exact right option provided. Four participants thought they could write their own solution that would be more specific to the use case, and mentioned using regular expressions to do so. In this case, they placed higher trust in their own solutions than those of the tool, likely due to their previous experience with regular expressions and these types of functions. Given developers' desires for fitting the use case, we should consider ways to help developers customize quick fixes or provide ways to link to additional options. We should also consider providing fixes for data types instead of use cases, since they would lend well to additional customization by the developer.

"Here I would like to see numbers only, but I don't quite see an option for that. I would probably go ahead and activate the letters and numbers quickfix and then modify it so that it's just numbers." -p1

Despite a few developers' desires to write their own sanitization functions, many developers strongly desired quick fixes whenever possible. Several even speculated on how to provide a quick fix option for SQL injection vulnerabilities, which would involve automatically turning dynamic statements into prepared statements in the code. Quick fixes may provide a significant boost to the effectiveness of static analysis since it reduces or removes the possibility of the developer improperly implementing a fix.

5.5 Assessing the Application of the Fix

Questions in this group related to how to apply the fixes and the impact on the code as a result. Developers did not question the difficulty of applying a quick fix, and expected this to be an easy process. Note that in this study we did not ask participants to generate and verify the fix. Instead, we queried developers for what they thought would happen and their concerns. They were hesitant to commit to saying that the chosen solution would make the code secure without viewing and testing the generated code. After choosing a quick fix for each vulnerability warning, developers were asked to respond with a number from 1 to 10 (with 1 being very weak and 10 being very strong) indicating how confident they were that the chosen quick fix would fix the problem. Among all tasks and all developers, the average given was 7.96 (SD 2.12), indicating generally high confidence.

While two developers expressed concern that the quick fix could break the code, the rest appeared to trust that it would not alter the functionality. This is in contrast to our previous evaluations of ASIDE with students, who were very fearful of implementing ASIDE's recommendations for fear of breaking their code. However, developers would often comment if they noticed some sort of logic issue or functionality concern in addition to the vulnerability. In many cases, they expressed concern that more steps may be necessary to properly secure the code beyond what ASIDE was flagging as an issue.

While they did not expect the quick fixes to break the code, developers were hesitant to trust the security capabilities of the generated code. This suggests that tools which provide quick fixes for vulnerabilities should generate code which is well commented and easy to read and understand in order to gain the developers' trust. Effort should also be placed into convincing the developers of the effectiveness of the solution.

5.6 Code Background and Functionality

Questions here relate to what the code is doing and why it's written the way it is. Participants felt it necessary to understand the application's functionality before applying a fix. Most of the time that participants spent thinking was spent trying to understand the underlying code. They often talked about how the code worked and sometimes justified their response to the warning based on it. Developers would also sometimes critique the code, particularly when they identified an issue in which the application logic did not make sense in a real world application. Lastly, participants sometimes expressed confusion over unneeded code and questioned whether or not it contributed to the vulnerability.

"I guess, the question is what is the account? Is the account like your username or is the account the actual, like a bank account name?"-p2

The study used code the developer was not familiar with, likely leading to greater time spent understanding the code context in order to support decisions. While our tool did not specifically help developers with understanding the background functionality, the warnings and tool interaction also did not seem to interfere. Developers were able to reason about the security implications based on the warnings, and relate the warning to the surrounding code. This demonstrates the potential benefits of embedding security warnings within the code development view.

5.7 Resources and Documentation

Questions in this grouping regarded how to find additional information and resources, and their reliability. As shown in Figure 2, ASIDE provides short explanations of warnings and quick fixes alongside the menu. Developers all read these explanations in detail. However, only 5 developers visited the more detailed "Read More" help page without prompting. Participants stated that the "Read More" option, shown in the menu with the rest of the quick fixes, did not stand out as containing additional information. Participants suggested placing a link to the detailed pages in the shorter menu explanations to make it more visible. While none did so, others mentioned they would be likely to google for more information if they wanted it.

Participants appeared to trust and appreciate the help pages,

and never questioned the correctness of the explanations or suggested solutions. In particular, participants praised the ability to expand and shorten sections as needed, as well as the copy/pastable concrete code examples:

"Ah ha! There's an example. That's a good example. How would you fix it? View example prepared statement. Ah, what'd you know. That's exactly what I was hoping it would show." -p1

"Yes! I feel confident now. That's exactly what I wanted to see." -p8

Thus, while our tool still needs to make these resources more visible, it appears that having the ability to explore layered explanations, growing ever more detailed, alongside their code was appreciated by developers and contributed to their understanding of what the tool was asking them to do.

6. CONCLUSIONS

We conducted a study to analyze how developers understand an interactive static analysis tool in the context of the questions we identified in our previous study. With only a few exceptions, our tool did seem to effectively communicate vulnerability information to developers. Providing this information within the code view helped developers understand the source and reason about solutions. We believe that contextualizing the text of messages and help pages improved understanding over our previous versions of our tool's interface. However, we also note the need to continue to improve the explanations for annotation requests, where developers are requested to provide security-related information. What did surprise us was how participants considered the possible exploits for a vulnerability, and their severity, and wanted more details of the implications of not fixing the issue. Participants were not so fearful of impacting functionality, but instead concerned with whether the solution was truly needed and actually resolved the security issue. Not surprisingly, participants desired quick fix options and concrete examples for mitigating vulnerabilities. Yet, they also expressed desire to customize these solutions to fit the specific use case in question. Thus, there may be a tension in supporting a set of clear and limited options for usability, and providing sufficiently flexible options to fit any code context.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] Owasp extended security api (esapi). https://www.owasp.org/index.php/Category:OWASP, 2016.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, Sept. 2008.

[3] M. Brandel. Cs online article. http://www.csoonline.com/article/2123602.html, 01/20/2009.

[4] 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*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[5] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 6 pp.–263, May 2006.

[6] S. Kerner. eweek.com article. software vulnerabilities lead to internal security problems kaspersky. http://www.eweek.com/security/software-vulnerabilities-lead-to-internal-security-problems-kaspersky.html, 12/05/2013.

[7] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[8] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 598–608, Piscataway, NJ, USA, 2015. IEEE Press.

[9] T. Schlein. The rise of the chief security officer: What it means for corporations and customers. http://www.forbes.com/sites/the-rise-of-the-chief-security-officer/, 2015.

[10] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259, New York, NY, USA, 2015. ACM.

[11] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill. A study of interactive code annotation for access control vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '15, Washington, DC, USA, 2015. IEEE Computer Society.

[12] 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*, ACSAC '11, pages 267–276, New York, NY, USA, 2011. ACM.

[13] 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*, CHI '12, pages 2707–2716, New York, NY, USA, 2012. ACM.

[14] 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.

[15] 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.