Tool Support for Questions Developers Ask About Security Vulnerabilities

Justin Smith, Brittany Johnson, and Emerson Murphy-Hill
North Carolina State University
Raleigh, NC 27606

Bill Chu and Heather Richter Lipford
University of North Carolina at Charlotte
Charlotte, NC 28223
{billchu, heather.lipford}@uncc.edu

***Abstract*—Program analysis tools should help developers answer the security-relevant questions; to design tools that help developers answer their questions, we must first understand what questions they ask. However, we lack a catalog and categorization of such questions in relation to security. We attempted to build this categorized catalog by conducting a laboratory experiment with novice and experienced software developers, observing their interactions with security vulnerabilities in iTrust, a Java medical records software system. Results of our study have several implications for program analysis tools aiming to help developers assess security vulnerabilities in their code, such as providing support for reasoning about the different possible attacks that could stem from a given security vulnerability.**

I. Introduction

Software developers are a critical part of making software secure. When there is a security fault in a software system, it is up to the developers to determine the best way to remove the vulnerability and ensure proper function of the system in the future. There are a variety of ways developers can learn about, assess, and fix insecure code, such as peer interactions or usage of tools. It is particularly important that effectives tools exist for secure systems, as security vulnerabilities are more likely to cause incidents that affect company budgets as well as end users [**?**].

Static analysis tools are made available to help developers detect potential security flaws in their code. One of example of such a tool is FindBugs, a static analysis tool for finding potential defects in software code. [1] FindBugs is able to report various potential security defects, such as SQL injection and cross site scripting. There is also an extension for FindBugs, called Find Security Bugs (FSB), that focuses its reporting efforts specifically on security vulnerabilities [2]. Other tools, such as CodeSonar [3] and Coverity [4], can also be used to detect and remove potential security vulnerabilities.

Unfortunately, however, research suggests that many developers may not be using these tools; part of this reason for this is that they have difficulty interpreting the output [**?**]. Part of this difficulty for developers building secure systems stems from the hard to answer questions developers have about the security of their code [**?**]. Though not the goal of their work, many researchers have taken the findings of LaToza and colleagues and been able to work towards improving the state-of-the-art for program analysis tools [**?**], [**?**], [**?**]. Because LaToza's work only addresses security questions in passing and does not focus on the process of assessing potential security vulnerabilities, we do not have such a collection of questions to use to build better, more usable tools for security. Our work extends existing work to security by providing a similar catalog of the questions that developers ask when assessing potential security vulnerabilities.

In this paper, we report a study conducted to create a better understanding of the information developers seek with using static analysis tools to identify and assess security vulnerabilities in code. We conducted a laboratory experiment with 10 developers familiar with the iTrust software system. [5] We observed each developer as they assessed potential security vulnerabilities reported by FSB and report the kinds of questions they asked while doing so along with the strategies they used, if any, to answer them. Using a card sort methodology, we sorted the 559 questions into 17 categories.

Our work makes three contributions. First, we present a list of questions that software developers ask when examining the security of their code. Second, based on our observations, we discuss the strategies that developers use to answer some of their questions. Finally, we evaluate the existing tool support for answering security-related questions.

The remainder of the paper is organized as follows. In Section II, we discuss previous work related to our own. Section III outlines the methodology we used to conduct our study and analyze our data. Next, we discuss the findings of our study in four parts. Section IV reports findings related to vulnerabilities, attacks and fixes. Section V discusses questions pertaining to the code and the application in which the code resides. Section VI reports findings pertaining the individuals and their thought processes. Section VII reports questions surrounding resources for solving the problem. Each section of findings includes observations made and implications for tool design that can be drawn based on these observations. Finally, we conclude the paper with a discussion of future work VIII.

II. Related Work

We have organized the related work into three sections. Section II-A outlines the predominant approaches researchers

[1] findbugs.sourceforge.net
[2] h3xstream.github.io/find-sec-bugs/
[3] http://www.grammatech.com/codesonar
[4] http://www.coverity.com/
[5] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start

use to evaluate security tools. Section II-B surveys the work done to facilitate developers' understanding of code, and section II-C references similar studies that have explored the questions developers ask when modifying, understanding, or debugging their code.

A. Evaluation of Techniques

Developers use several techniques and tools to help secure their systems. Using a variety of metrics, many studies have assessed the effectiveness of the tools and techniques developers use to find and remove vulnerabilities from their code [**?**], [**?**], [**?**].

Much research has evaluated the effectiveness of tools and techniques based on their false positive rates and how many vulnerabilities they detect [**?**], [**?**], [**?**]. Jovanovic and colleagues attempt to address the problem of vulnerabilities in web applications by implementing and evaluating PIXY, a static analysis tool that detects cross-site scripting vulnerabilities in PHP scripts [**?**]. They considered their tool effective because of its low false positive rate (50%) and its ability to find vulnerabilities previously unknown. Similarly, Livshits and Lam evaluated their own approach to static analysis for detecting security vulnerabilities; the static analyzers used are created from specifications provided by the user [**?**]. They also found their tool to be effective because it had a low false positive rate.

Austin and Williams compared the effectiveness of four existing techniques for discovering security vulnerabilities: systematic and exploratory manual penetration testing, static analysis, and automated penetration testing [**?**]. Comparing the four approaches based on number of vulnerabilities found, false positive rate, and efficiency, they reported that no one technique was capable of discovering every type of vulnerability. Dukes and colleagues conducted a case study comparing static analysis and manual testing techniques used to find security vulnerabilities [**?**]. They found that the combination of manual testing and static analysis was most effective, because it found the most vulnerabilities.

These studies use various measures of effectiveness, such as false positive rates or vulnerabilities found by a tool, but none focus on how developers interact with the tool. Further, they do not evaluate whether the tools under study answer questions that developer have about the code or vulnerabilities. Unlike existing studies, our study explores the questions developers ask when attempting to understand the vulnerabilities in their code.

B. Developer Understanding

In non-security domains, research has shifted focus from the technical effectiveness of program analysis tools to usability [**?**], [**?**], [**?**]. These studies sought to evaluate how tools communicate information to developers and how a tool can enhance a developer's understanding of the code. Such studies examine proposed and existing approaches using various qualitative methods such as interviews and surveys.

Some research surrounding tool effectiveness shifted focus from technical effectiveness of program analysis tools to usability [**?**], [**?**], [**?**]. The goal of these studies was to evaluate how tool designers can help developers cope with existing or proposed approaches, through comparison of proposed and existing approaches and various qualitative methods, such as interviews and surveys.

Ayewah, Pugh, and colleagues conducted a series of interviews and surveys, along with a controlled user study, in an attempt to better understand the practices and needs of developers when using the static analysis tool FindBugs [**?**], [**?**].

Khoo and colleagues focused their efforts on improving the user interface of existing tools by developing and evaluating PATH PROJECTION, a user interface toolkit to help developers navigate through and understand the errors in their code [**?**].

Johnson and colleagues conducted semi-structured interviews with professional developers to determine the reason developers have for using, and possibly not using, static analysis tools to find defects in their codes [**?**]. Their findings suggest that one of the biggest reasons developers may have for not using tools to analyze their code is difficulty understanding the output provided.

Layman and colleagues conducted a study with developers to explore the factors developers consider when deciding whether they will fix a defect or not [**?**]. They found that, among other criteria considered, developers consider the description of the fault critical to assessing its importance. Based on their findings, they discuss design implications for automated fault detection tools.

These studies discuss the ability of developers to effectively use existing tools and ways to improve them, however, these studies do not focus on the information needs of developers when using static analysis tools in their code. The goal of our study is to identify the security-related questions developers ask when approaching security vulnerabilities so that tool designers understand the information requirements of security-minded developers.

C. Answering Developer Questions

Several studies have explored the knowledge requirements of developers when coding. Similar to our work, some existing studies focus on the questions developers ask to determine these knowledge requirements [**?**], [**?**]. In contrast, our study focuses specifically on the knowledge requirements of developers when assessing security vulnerabilities.

Much of the work on answering developer questions has occurred in the last decade. LaToza and Myers surveyed professional software developers to understand the questions developers ask during their daily coding activities, focusing on the hard to answer questions [**?**]. Futhermore, after observing developers in a lab study, they discovered that the questions developers ask tend to be questions revolving around searching through the code for target statements, or reachability questions [**?**]. Ko and Myers developed WHYLINE, a tool meant to ease the process of debugging code by helping answer "why

did" and "why didn't" questions developers have [**?**]. They found that developers were able to complete more debugging tasks when using their tool than they could without it. Fritz and Murphy developed a model and prototype tool to assist developers with answering the questions they want to ask based on interviews they conducted [**?**].

Our work builds on the work of LaToza and Myers, which found that some of the hard-to-answer questions developers have surround the implications of code changes on the security of their code. Our study discovers the questions developers have about potential code vulnerabilities and discusses the implications of the changes they may make when attempting to resolve them.

III. Methodology

We conducted a laboratory experiment with 10 software developers. In our analysis, we extracted and categorized the questions developers asked during each study session. Section III-A outlines the research questions we sought to answer. Section III-B details how the study was designed and section III-C describes how we performed data analysis.

A. Research Questions

We want to answer the following research questions:

- **RQ1**: What types of security related questions do developers ask?
- **RQ2**: How do current tools and approaches support developers in answering these questions?

B. Study Design

Each session started with a five-minute briefing section, followed by encounters with four vulnerabilities. All participants consented to have their session recorded using screen and audio capture software. Finally, each session concluded with several demographic and open-ended discussion questions.

1) Materials: Participants used Eclipse to explore vulnerabilities in iTrust, an open source Java medical records web application that ensures the privacy and security of patient records according to the HIPAA statue.[6] During the briefing session, participants were given time to familiarize themselves with the development environment and ask any questions about the experimental setup. Participants were equipped with a version of FindBugs extended with FSB. We chose FSB because it is open source, lightweight, and similar to other static analysis tools for detecting security vulnerabilities in web applications, such as those listed by NIST[7], OWASP[8], and WASC.[9] Figure 1 depicts the configuration of the integrated development environment (IDE) for one of the tasks.

[6]hhs.gov/ocr/privacy/
[7]samate.nist.gov/index.php/Source_Code_Security_Analyzers.html
[8]owasp.org/index.php/Source_Code_Analysis_Tools
[9]projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList

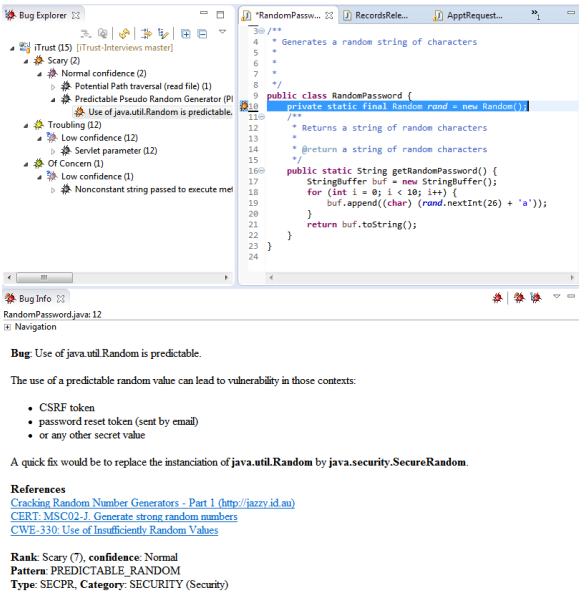

Fig. 1. The environment participants were presented with.

2) Participants: To simulate a more realistic work environment, participants were required to have significant experience working on the iTrust code base. All participants either completed or served as teaching assistants for a semester-long software engineering course that focused on developing iTrust.

For our experiment, we recruited 10 software developers, 6 students and 4 professionals. Participants ranged in experience from 4 to 10 years, averaging 6.3 years. All interviews were conducted in-person. We recruited participants from personal contacts and class rosters, using snowball sampling to find additional qualified participants. Since extracting data from each interview required intensive analysis, we ceased recruitment when we felt that we had reached saturation and few new questions were emerging from each additional interview.

Table I gives additional demographic information on each of the 10 participants. Since the focus of our study was to identify the questions developers ask independent of their background, we report on demographic information to ensure a degree of diversity in our sample and contextualize participants' responses.

3) Tasks: Each participant was asked to assess four vulnerabilities. We presented each participant with this number of vulnerabilities, because in preliminary pilot sessions, participants spent approximately 10-15 minutes with each vulnerability and showed signs of fatigue after 60 minutes.

When selecting tasks, we gave preference to existing iTrust vulnerabilities and sought a broad set of question topics. FSB uses topic abbreviations to group similar error messages. For example the abbreviation for the topic Path Traversal is PT and applies to notifications that warn about potential path traversal. FSB reports four types of vulnerabilities so, to cover a broad set of question topics, we wanted to make sure we included one of each type in our experiment. FSB identified three types of naturally occurring vulnerabilities: cross site scripting, path

TABLE I
DEMOGRAPHICS OF STUDY PARTICIPANTS

Participant	Job Title	Security Vulnerability Familiarity	Programming Experience(years)
P1	Student	★★½☆☆	4.5
P2	Test Engineer	★★★☆☆	8
P3	Development Tester	★★☆☆☆	6
P4	Software Developer	★★☆☆☆	6
P5	Student	★★★★☆	10
P6	Student	★☆☆☆☆	4
P7	Front-end Software Developer	★★★★☆	4.5
P8	Student	★★★☆☆	7
P9	Software Consultant	★★★☆☆	5
P10	Student	★★★☆☆	8

traversal, and predictable random vulnerabilities. We inserted a SQL injection vulnerability by making minimal alterations to one of the database access objects. Our modification preserved the functionality of the original code and was based on simple, toy examples of SQL injection presented by OWASP. Table II summarizes each of the four vulnerabilities.

During the briefing section, we asked participants to pretend they are in charge of security for iTrust and to approach the vulnerabilities as if they were in their normal work environment. Additionally, we asked them to use a think-aloud protocol, which encourages the participant to verbally announce their thought process as they complete a task or activity [**?**]. Specifically, they were asked to, "Say any questions or thoughts that cross your mind regardless of how relevant you think they are." We recorded both audio and the screen as experimental artifacts for data analysis.

The session moderator was equipped with the following questions, but had the freedom to omit questions and ask follow-up questions based on the participant's responses.

- What are you exploring/trying to figure out right now?
- What information do you need to proceed?
- Can you explain what this warning is trying to tell you?
- Based on your understanding of this error message, what is the percentage likelihood you would modify this section of he code?
- How would you fix the problem? Where would you start?
- On a scale of 1-5, how confident are you in your understanding of the error message?
- On a scale of 1-5, how confident are you that you are making the correct judgments?
- What information would you like to see added to the error message?
- What part of the message is most helpful?
- Look at the error message one last time. Is there any information you initially disregarded?
- Would you like to share any other thoughts about this vulnerability?

C. Data Analysis

We analyzed session data using an approach inspired by grounded theory [**?**]. First, we transcribed all the audio-video files using oTranscribe.[10] Each transcript, along with the associated recording, was analyzed by two of the authors for implicit and explicit questions. The two question sets for each session were then iteratively compared against each other until the authors reached agreement on the question sets. In the remainder of this section, we will detail the question extraction process and question sorting processes, including the criteria used to determine which statements qualified as questions.

1) Question Criteria: Participants ask both explicit and implicit questions. Drawing from previous work on utterance interpretation, we developed 5 criteria to assist in the uniform classification of participant statements [**?**]. A statement was coded as a question only if it met one of the following criteria:

- **The participant explicitly asks a question.**
  Example: *Why aren't they using Prepared Statements?*
- **The participant makes a statement and explores the validity of that statement.**
  Example: *It doesn't seem to have shown what I was looking for. Oh, wait! It's right above it...*
- **The participant uses key words such as, "I assume," "I guess," or "I don't know."**
  Example: *I don't know that it's a problem yet.*
- **The participant clearly expresses uncertainty over a statement.**
  Example: *Well, it's private to this object, right?*
- **The participant clearly expresses a knowledge requirement by describing plans to acquire information.**
  Example: *I would figure out where it is being called.*

2) Question Extraction: Using the criteria outlined in the previous section, two of the authors independently coded each session. When we identified an explicit question or implicit question, which is a statement that satisfies one or more of the criteria, we marked the transcript, highlighted the participant's original statement, and clarified the question being asked. Question clarification typically entailed rewording the question to best reflect the information the participant is trying to acquire. From the 10 sessions, the first coder extracted 421 question statements; the other coder extracted 389.

To make sure our list of questions was exhaustive, two independent reviewers coded the transcripts using the criteria. It was sometimes difficult to determine what statements should

[10] otranscribe.com

TABLE II
FOUR VULNERABILITY EXPLORATION TASKS

Vulnerability	Short Description	Tool's Rank	Tool's Confidence
Potential Path Traversal	An instance of java.io.File is created to read a file.	"Scary"	Normal
Predictable Random	Use of java.util.Random is predictable.	"Scary"	Normal
Servlet Parameter	The method getParameter returns a String value that is controlled by the client.	"Troubling"	Low
SQL Injection	[Method name] passes a non-constant String to an execute method on an SQL statement.	"Of Concern"	Low

Fig. 3. Result of phase one of card sorting.

Fig. 4. Sorting cards into categories based on emergent themes.

be extracted as questions; the criteria helped ensure that both reviewers only highlighted the statements that reflected actual questions. Figure 2 depicts a section of the questions extracted by both authors from P8 prior to review.

3) Question Review: To remove duplicates and ensure the validity of all the questions, each transcript was reviewed jointly by the two authors that initially coded it. During the second pass, the two reviewers examined each question statement, discussing the justification for each question based on the previously stated criteria. The two reviewers merged duplicate questions, favoring the wording that was most strongly grounded in the experiment artifacts.

Each question that was only identified by one author required verification. If the other author did not agree that such a question met at least one of the criteria, the question was removed from the question set and counted as a disagreement. The reviewers were said to agree when they merged a duplicate or verified a question.Depending on the participant, inter-reviewer agreement ranged from 91% to 100%. Across all participants, agreement averaged to 95%. High agreement score indicates that the two reviewers consistently held similar interpretations of the question criteria.

Figure 2 depicts a section of the questions extracted by both authors from P8 prior to review.

4) Question Sorting: To organize our questions and facilitate discussion, we performed an *open* card sort [**?**]. Card sorting is typically used to help structure data by grouping related information into categories. In an *open* sort, the process begins with no notion of predefined categories. Rather, sorters derive categories from emergent themes in the cards.

We performed our card sort in three distinct stages: clustering, categorization, and validation.

In the first stage, we formed question clusters by grouping questions that identified the same information requirements (Figure 3). In this phase we focused on phrasing similar questions consistently and grouping duplicates. For example, P1 asked, *Where can I find information related to this vulnerability?* P7 asked, *Where can I find an example of using prepared statements?* and P2 asked, *Where can I get more information on path traversal?* Of these questions, we created a question cluster labeled *Where can I get more information?* At this stage, we discarded 5 unclear or non pertinent questions and organized the remaining 554 into 155 unique question clusters.

In the second stage, we identified emergent themes and grouped the clusters into categories based on the themes. For example, we placed the question *Where can I get more information?* into a category called ***Resources/Documentation***, along with questions like *Is this a reliable/trusted resource?* and *What information is in the documentation?* Table III contains the 17 categories along with the number of distinct clusters each contains.

To validate the categories that we identified, we asked two independent researchers to sort the question clusters into our categories. Rather than sort the entire set of questions, we randomly selected 43 questions for each rater to sort. The first rater agreed with our categorization with a Cohen's Kappa of $\kappa = .63$. Between the first and second rater we reworded and clarified some ambiguous questions. The second rater exhibited greater agreement ($\kappa = .70$). These values are within the $.60 - .80$ range, indicating substantial agreement [**?**].

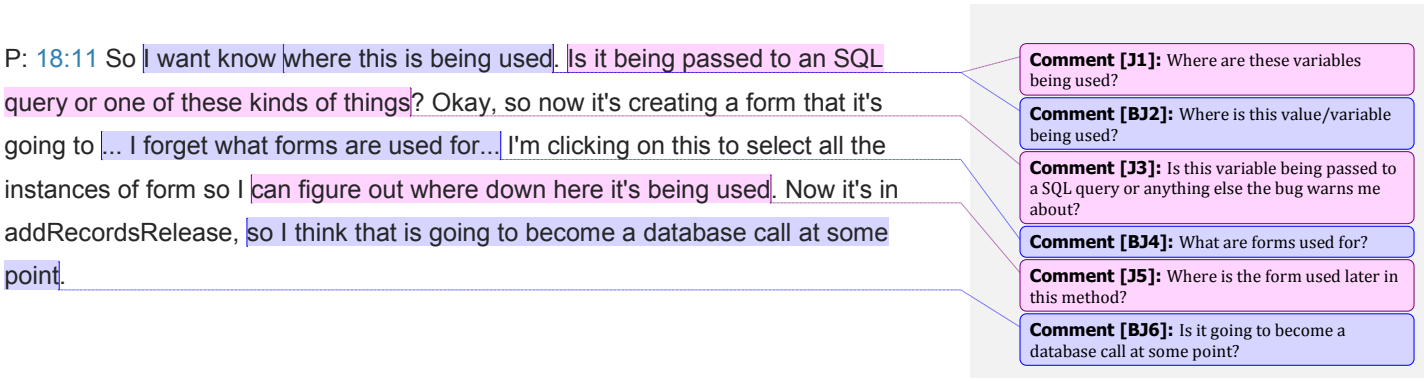

Fig. 2. Question merging process

TABLE III
EMERGENT CATEGORIES FROM CARD SORT

Group	Category	Clusters	Location in Paper
Vulnerabilities, Attacks, and Fixes, Oh My!			
	Understanding Alternative Fixes and Approaches	11	Section IV-A
	Preventing and Understanding Potential Attacks	11	Section IV-B
	Assessing the Application of the Fix	9	Section IV-C
	Relationship Between Bugs	3	Section IV-D
Code and the Application			
	Locating Information	11	Section V-A
	Control Flow and Call Information	13	Section V-B
	Data Storage and Flow	11	Section V-C
	Code Background and Functionality	17	Section V-D
	Application Context/Usage	9	Section V-E
	End-User Interaction	3	Section V-F
Individuals			
	Developer Planning and Self-Reflection	14	Section VI-A
	Understanding Concepts	6	Section VI-B
	Confirming Expectations	1	Section VI-C
Problem Solving Support			
	Bug Severity and Rank	4	Section VII-C
	Understanding and Interacting with Tools	9	Section VII-B
	Resources and Documentation	10	Section VII-A
	Error Messages	3	Section VII-D
	Uncategorized	10	

IV. RESULTS: VULNERABILITIES, ATTACKS, AND FIXES, OH MY!

In the next 4 sections, we discuss the results of our experiment using the categories we described in the previous section. Due to the large number of categories, we put the categories into groups to organize and facilitate discussion about our findings; groups are listed in column 1 of Table III. For each category, we list distinct questions and the number of participants that asked a variation of that question in parentheses. For each category, we discuss the common thread that ties questions in that category together. We list each distinct question we extracted with the number of participants that asked that question in parentheses. We detail our observations of the participants and contextualize each category by discussing existing tool support and shortcomings.

A. *Understanding Alternative Fixes and Approaches (8)*

When resolving security vulnerabilities, participants encountered alternative approaches that appeared to achieve the desired functionality more securely. For example, when evaluating the SQL injection vulnerability, participants found resources that suggested using the Prepared Statement Class instead of Java Statements. Participants found many sources for alternative fixes, including web resources and even code located in other parts of the project. When presented with alternative approaches, participants compared the alternative with the current code and assessed the applicability of the alternative. Eleven of the 155 questions we extracted pertained to understanding the various facets of alternative approaches or solutions, such as:

Does the alternative function the same as what I'm currently using? (6)
What are the alternatives for fixing this? (4)
Are there other considerations to make when using the alternative(s)? (3)
How does my code compare to the alternative code in the example I found? (2) Why should I use this alternative

method/approach to fix the vulnerability? (2)
When should I use the alternative? (1)
Is the alternative slower? (1)

Observations Eight participants had questions that fit into this category. The FSB notifications for the SQL Injection and Predictable Random vulnerabilities explicitly offer alternative fixes. As noted, the message associated with the SQL Injection vulnerability suggests switching to use Prepared Statements. The message associated with the Predictable Random vulnerability suggests switching to use *java.security.secureRandom*. In other cases, participants turned to a variety of sources, such as StackOverflow, official documentation, and personal blogs for alternative approaches. Three participants specifically mentioned StackOverflow as a source for better understanding alternative approaches and fixes. P7 preferred StackOverflow as a resource, because it included real-world examples and elaborated on what was wrong with the code as it was currently written. While attempting to assess the Servlet Parameter vulnerability (Table II), P8 decided to explore some resources on the web and came across a resource that appeared to be affiliated with OWASP. [11] Because he recognized OWASP as "the authority on security," he clicked the link and used it to make is final decision regarding the vulnerability. Despite the useful information that some participants found, often the candidate alternative did not readily provide meta-information about trade-offs or the process of applying suggestions. For example, P9 found a suggestion on StackOverflow that he thought might work, but it was not clear if it could be applied to the code in iTrust.

Tool Implications It seems especially important when dealing with potential security vulnerabilities that developers fully understand any changes they make to the code. However, based on our observations, developers lack easy-access to security-relevant information on potential alternatives. Rather, participants used ad-hoc strategies to seek different pieces of information from a variety of sources.

The questions in this category suggest that developers could use a tool that aggregates information on alternative secure coding practices. Our observations also suggest that such a tool should include real world examples and draw from trusted authorities on security. Finally, the tool should present information about the trade-offs associated with a particular alternative, such as decreased performance or the possibility of introducing additional vulnerabilities.

B. Preventing and Understanding Potential Attacks (10)

Unlike other types of code defects that may cause code to function unexpectedly or incorrectly, security vulnerabilities open the code up to potential attacks. For example, the servlet parameter vulnerability introduced the possibility of SQL injection, path traversal, command injection, and cross-site scripting attacks. Eleven of the 155 distinct questions we gathered belong in this category.

Is this a real vulnerability? (7)
What are the possible attacks that could occur? (5)
Why is this a vulnerability? (3)
How can I prevent this attack? (3)
How can I replicate an attack to exploit this vulnerability? (2)
What is the problem (potential attack)? (2)
How can this vulnerability lead to an attack? (1)
How should I address this problem? (1)
How do I find out if this is a real vulnerability?(1)

Observations Participants sought general information relating to the attacks in a given context. To that end, five participants asked *What are the possible attacks that could occur?* For example, within the first minute of his analysis, P2 stated that he was thinking about the different types of attacks that could target web applications, like iTrust.

On the other hand, participants also sought specific attack-information regarding the vulnerability at hand. Participants hypothesized about specific attack vectors, how to execute those attacks, and how to prevent those attacks now and in the future. Seven participants asked the question, *Is this a real vulnerability?* To answer that question, participants searched for hints that an attacker could actually execute an attack in the given context. For example, P10 determined that the Predictable Random vulnerability was "real," because an attacker could deduce the random seed and use that information to determine other users' passwords.

Tool Implications Unlike other contexts, when analyzing security vulnerabilities, developers ask questions pertaining to potential attacks. Trivially, the existence of this category suggests that security tools should be designed differently from other code analysis tools; they should present pertinent information about potential attacks attacks.

However, the questions in this category also suggest that developers operate in two distinct modes when reasoning about potential attacks. At times they inquire about all the possible attacks that could occur. At other times, they explore the feasibility of a specific attack. Security tools should support this modality by separating information about all the possible attacks from more detailed information about specific attacks.

C. Assessing the Application of the Fix (9)

Once participants had identified a specific approach for fixing a security vulnerability, they asked questions about applying the fix to the code. For example, when considering the use of *java.security.secureRandom* to resolve the Predictable Random vulnerability, participants questioned the applicability of the fix and consequences of making the change. The questions in this category differ from those in ***Understanding Alternative Fixes and Approaches*** (Section IV-A). These questions focus on the process of applying

[11] https://www.owasp.org/index.php/Main_Page

and reasoning about a given fix, rather than identifying and understanding possible fixes. We placed 9 of the 155 distinct questions into this category.

Will the error go away when I apply this fix? (5)
How do I use this fix in my code? (4)
How do I fix this vulnerability? (4)
How hard is it to apply a fix to this code? (3)
Is there a quick fix for automatically applying a fix? (2)
Will the code work the same after I apply the fix? (2)
What other changes do I need to make to apply this fix? (2)
Can these fix suggestions be applied to my code? (1)
Does the code stand up to additional tests prior to/after applying the fix? (1)

Observations

When searching for approaches to resolve vulnerabilities, participants gravitated toward quick fixes. The notifications associated with the Predictable Random vulnerability and the SQL Injection vulnerability both provided quick fix suggestions. All participants proposed solutions that involved applying one or both of these quick fixes. Additionally, two participants explicitly asked if quick fixes were available and P2 even commented that it would be nice if all the notifications contained quick fixes. However, unless prompted, none of the participants commented on the disadvantages of using proposed quick fixes, such as reduced performance or the possibility of introducing another defect.

P9 had a noteworthy experience while exploring the Predictable Random vulnerability. Unlike the other participants, he missed the part of the error message that described the "quick fix." He went on to detail his plans for searching the web for standard random number generation techniques. Then he recalled a technique that he had encountered before and explained that he might implement his own random hashing algorithm, but concluded that he would do more searching to find out the best way to apply that fix. Before proceeding to the next vulnerability, P9 reread the error message, noticed the quick fix and decided he would apply the quick fix instead.

Tool Implications Assuring the security of complex software systems requires developers to reason critically about the code. Quick fixes help developers dismiss vulnerabilities more quickly. However, especially in security, quicker might not always be better. In the case of P9, without the quick fix he searched the web for best-practices and drew conclusions from a broader set of resources. In some sense, his experience lead him to a better understanding of the vulnerability than those who read the entire notification. Our observations suggest that the presence of quick fixes alone may discourage developers from fully considering the implications of their proposed changes.

Existing tools, such as Quick Fix Scout begin to address this problem by describing what might happen if a quick fix is implemented [**?**]. Quick Fix Scout helps developers consider the implications of implementing a fix. Security tools that include quick fixes should also encourage developers to assess alternative fixes or whether they need to modify the code at all.

D. *Relationship Between Bugs (4)*

Some participants asked questions about the connections between co-occurring vulnerabilities and whether or not similar vulnerabilities exist elsewhere in the code. For example, when participants reached the third and fourth vulnerabilities, they began noticing and speculating about the similarities between the vulnerabilities they inspected. Three of the 155 distinct questions we found target the relationships between the vulnerabilities in the code.

Are all the bugs related in my code? (3)
Does this other piece code have the same bug as the code I'm working with? (1)
Are all of these notifications vulnerabilities? (1)

Observations

When inspecting the Servlet Parameter vulnerability, P8 wondered if other parts of the code contained similar vulnerabilities. NOT MUCH TO SAY HERE?

Tool Implications

Though only four of the ten participants asked questions about the relationship between bugs, these questions highlight information requirements that tools could support. When inquiring about the relationships between bugs, participants noted connections between bugs that related to similar topics. FSB organized the vulnerabilities it identified based on their locations in the code, or alternatively based on their seriousness. Grouping vulnerabilities based on abstract concepts might enable developers to focus on areas in which they have expertise, or identify and resolve similar bugs more quickly.

HELP: I know about CWE are there other canonical taxonomies? More importantly, do tools present bugs according to these taxonomies (group based on similar topic)?

V. Results: Code and the Application

A. *Locating Information (10)*

Participants asked questions about locating information in their coding environments. Eleven of the 155 extracted questions fit into this category. In the process of investigating vulnerabilities, participants searched for information across multiple classes and files. For example, when exploring the Path Traversal vulnerability, participants sought the locations in the code where the method was being called and where the path parameter was set.

Where is this used in the code? (10)
Where are other similar pieces of code? (4)
How do I track this information in the code? (2)
Where is this artifact? (1)
Is this artifact located in this class? (1)
Where is this method defined? (1)

Where is this class? (1)
Where is the next occurrence of this variable? (1)
How do I navigate to other open files? (1)

Observations. All 10 participants wanted to locate where defective code and tainted values were being used throughout the system. Most of these questions occurred in the context of assessing the Predictable Random and Servlet Parameter vulnerability. Specifically, participants wondered where the random number generator and the tainted values returned from the servlet were used. For example, P1 attempted to gather such information using existing tools like the code element highlighting offered by Eclipse, which shows where a code element is being used locally. Others mentioned wanting to know where the parameter was used, but did not take steps to actually find out.

For the most part, participants successfully leaned on existing tools for answering simple questions about locating information like finding occurrences of a variable or references to a specific class name. However, strategies relying on existing tools fell short when participants sought context-dependent information. The code highlighting tool displayed locations where variables were referenced and set locally, but did not explicitly inform developers of where the variables were put into a database or used in other security-critical contexts. Participants didn't just seek location information, they searched for places where information was used within a given context. For example, participants asked where the servlet parameters were being used, and more specifically, they asked where the servlet parameters were being re-displayed on a web page or where they were stored in a database.

In other cases, four participants wanted to find parts of the code that implemented similar functionality to what they were inspecting. For all of them, this was a manual process involving mostly scrolling through and exploring the code based on their knowledge of the application. For example, while assessing the SQL Injection vulnerability, P2 and P5 both wanted to know more about how the functionality under investigation was implemented in other parts of the code. P2 wanted to know, more specifically, what other parts of the code use Database Access Objects.

Tool Implications. The answers to some of these questions can be found using existing tool support found inside the IDE. For example, Eclipse has an 'open declaration' tool that can be used to find a method's definition. However, in the case of context-dependent questions, developer's could benefit from increased tool support. For example, if search tools had knowledge of the security-critical components in the code, they could prioritize results that occurred within or interacted with critical components. Currently developers must manually prioritize search results and determine whether each is security-relevant.

Developers could also benefit from tool support in searching their applications for code written similarly to the potentially vulnerable code. A tool that supports this process might include references to other parts of the project with similarly structured code, or code that calls a similar set of methods. Such a tool would help developers identify parts of the project that may also contain vulnerabilities, or conversely, strategies for resolving the vulnerability.

B. Control Flow and Call Information (10)

We also extracted questions relating to control flow and calling information. Developers seek information pertaining to the methods that get called, and whether they may get called with dirty parameters. Participants searched up the call chain, especially while evaluating the Potential Path Traversal vulnerability, trying to understand the possible files that could get opened. Many determined that since all the calls originated in test classes, they could dismiss the vulnerability. Thirteen of the 155 distinct questions extracted dealt with acquiring control flow and call information.

Where is the method being called? (10)
How can I get calling information? (7)
Who can call this? (5)
Are all calls coming from the same class? (3)
What gets called when this method gets called? (2)
What is the call hierarchy? (1)
What causes this to be called? (1)
How often is this code called? (1)

Observations. While exploring the Path Traversal vulnerability, participants sought information about the methods being called. The private method containing this vulnerability was only called by one method, *parseAndCache*, which was defined directly above it. *ParseAndCache*, which was also private, got called by a public method. Tracing up the chain, the method containing the vulnerability was eventually called from multiple classes that were all contained within a test package.

All 10 participants wanted call information for this vulnerability, often asking the question *where is this method being called?* However, participants used a mix of several different strategies to obtain the same information. The most basic strategy was simply skimming the file for method calls, which was error prone because participants could easily miss some calls. Other participants used the Eclipse's automatic code highlighting, which, to a lesser extent, was error prone for the same reason. Further, it only found calls within the current file. Another strategy participants employed was the use of a global textual search, which found all occurrences of a method name, but there was no guarantee that two names referred to the same method and also returned references that occurred in dead code or comments. Alternatively, Eclipse's Find References tool identifies proper references to a single method. Unlike the previous strategies, Eclipse's call hierarchy tool enables users to traverse a project's entire call structure; however, it does not identify calls made from framework hooks or APIs. Additionally, some participants were unaware of how to access and how to use the call hierarchy tool.

Eventually, many participants hypothesized that all the calls might originate from the same (test) class. Three participants explicitly asked *Are all calls coming from the same class?* To verify this assumption, participants primarily used the heuristic of visually examining the list of calling methods and their package names, scanning for the test keyword. This strategy does not scale particularly well and would fail if the methods were named incorrectly, or if they accidentally missed one.

Tool Implications. Particularly when resolving security vulnerabilities, developers are concerned with *all* of a method's potential call sites. Tools like find references and call hierarchy provide support for locating call information, yet the leave room for extension. Our findings suggest that tools that reason about, and notify developers of, calls that originate from outside the local code could help them answer the questions in this category.

Despite the existence of tools like 'find references' and the call hierarchy tool, based on our observations, developers face barriers preventing them from invoking sophisticated tools, perhaps due to a lack of awareness. Our findings suggest that developers could benefit from tools that reduce the cost of using a new tool. In particular developers could benefit from tools that apprise them of other useful tools like find references and call hierarchy. A tool that addresses these needs might detect when the developer is looking for call information (for example, using the 'find' tool) and suggest the appropriate tool for finding the information they require.

Tool Implications. Even with tools like 'find references' and the call hierarchy view tool, it may be difficult to answer questions like *how can I get calling information* and *who can call this* without having to manually, and mentally, aggregate information gathered from different tools. Further, even a combination of these tools are ineffective for someone who is unaware of their existence. A tool that addresses these types of needs might detect when the developer is looking for call information (for example, using the 'find' tool) and suggest the appropriate tool for finding the information they require. To increase visibility of the desired information, the tool might also highlight the relevant method as it appears in the call hierarchy or augment each method with the methods that can call it.

C. *Data Storage and Flow (10)*

Much like the questions related to ***Control Flow and Call Information***, participants also asked questions that pertained to the data being stored and carried throughout the program. Often participants wanted to understand the type of data being collected and stored, where the data came from, and where it was going. For example, participants wanted to determine where data was coming from, whether it was generated by the application or passed in by the user, and where it was going, i.e., whether it was being stored in the database. Eleven of the 155 distinct questions fit into this category.

Where does this information/data go? (9)
Where is the data coming from? (5)
How is data put into this variable? (3)
Does data from this method/code travel to the database? (2)
How do I find where the information travels? (2)
How does the information change as it travels through the programs? (2)
What does the variable contain? (1)
Is any of the data malicious? (1)

Observations. All 10 participants had questions regarding the data pipeline, or where data travels and how it changes as it travels. Participants asked questions about the data pipeline when assessing three of the four vulnerabilities; most often these questions arose while assessing the Servlet Parameter vulnerability. While exploring this vulnerability, none of the participants invoked tools specifically designed for exploring data flow. Instead, participants either stated their information needs without following up, or used the call hierarchy tool in combination with manual searching. For example, when P7 was determining the details of the potential vulnerability, he manually traced through method calls to determine if the data controlled by the client would reach the database.

The most popular questions in this category are *Where does this information go?* and *Where is the data coming from?*. Often when assessing the Servlet Parameter vulnerability, both these questions occurred. For instance, P6 wanted to know where the form data was set and where the data where going to see if the input is every validated. Similarly, while evaluating the Potential Path Traversal vulnerability, P1 wanted to know where the path originated to determine if it was coming from a secure source. Though both P1 and P6 knew the questions they needed answered, they did not attempt to use any of the tools, including manual search, to determine the answers. Often, like P1 and P6, participants would describe what they would do but not attempt to do it

Tool Implications. The call hierarchy tool in Eclipse provides a way of understanding control flow, but this static information does not give the full picture of how data travels through the program. The developer would have to retrieve this information, either from the documentation, by debugging the code, or by using a combination of search tools and manual inspection. Tools have been developed to do data flow analysis and report defects to programmers based on such analysis, however, they do not allow developers to explore the flow of data through their programs [**?**]. There has been some research and development surrounding the creation of data flow graphs for visualizing data dependencies in a program, however, data dependency graphs become unwieldy as applications grow large [**?**], [**?**]. It may be that some of the existing tools for data flow analysis could be augmented with functionality for exploring visualizations of the static data flow information already being gathered; this is one way tools could better support the answering of these types of data flow questions.

D. *Code Background and Functionality (9)*

Participants also asked questions concerning the background and intended function of the code being analyzed. Seventeen of the 155 distinct questions extracted fit into this category. The questions in this category focus on the code, such as determining the role a component or piece of code plays in the entire system or program. They ask about the history of the code and the context in which the code resides.

What does this code do? (9)
Why was this code written this way? (5)
Why is this code needed? (3)
Who wrote this code? (2)
Is this library code? (2)
Why are we using this API in the code? (2)
Are there tests for this code? (1)
Is this code doing anything? (1)
How much effort was put into this code? (1)

Observations. Nine of the 10 participants asked questions about the background of the code; most participants wanted to know what parts of the code did. Participants asked *What does this code do?* about all four vulnerabilities.

Participants were interested in the history of the code. For example, they asked why it was written a specific way, who wrote the code, or how much effort had been put into the code. P2 asked how much effort had been put into the code to determine whether he trusted that the code was written securely. While assessing the Potential Path Traversal vulnerability, some participants came to the realization that the authors of the code could help answer some questions. For example, P10 stated that he would normally ask his team members for background on how the code worked. Further, five participants wanted to know why the code was written the way it was written; this question occurred in all four vulnerabilities.

Tool Implications. Based on our observations, it seems that understanding the history and status of the code helps developers assess its security. If the code shows signs of neglect, uses deprecated libraries, or has not been tested and reviewed, some developers might be less likely to trust the security of the code. Developers seem interested in investigating the specific areas of code that exhibit these properties.

Version control systems already track some information about the authors and history of the code. Perhaps tools could use this information to prioritize vulnerabilities or direct developers to problematic areas.

E. *Application Context/Usage (9)*

Some participants wanted to know how code entities or pieces of code fit into the overall context of the application or system under analysis. The questions ranged from specific questions about a method or variable to general questions regarding the usage of the entire system. Nine of the 155 distinct questions revolved around how the code works in the context of a portion of or the entire system.

What is the context of this bug/code? (4)
Is this code used to test the program/functionality? (4)
What is the method/variable used for in the program? (3)
Will usage of this method change? (2)
Is the method/variable every being used? (2)
Are we handling secure data in this context? (1)
How does the system work? (1)

Observations. Four participants wanted to know the context in which the code being assessed was used or the context of the bug being assessed. Similarly, three participants found themselves trying to determine what a specific method or variable is used for in the application. Both of these questions came up for three of the four vulnerabilities.

P7 had questions concerning the context of the code associated with the Potential Path Traversal and Predictable Random vulnerabilities. For the former vulnerability, he wanted to know the general context in which the code is used; for the latter, he wanted to know the legal context in which the code is being used. P7, like others, asked these questions but did not follow through to answer them. P2, when thinking about how he might determine the context in which the code associated with the Predictable Random vulnerability is used, stated that the tool would have been more useful if it could tell him this information. He followed this statement with the notion that this sort of tool may be "unrealistic."

Related to determining the general context is determining if the code associated with the vulnerability is used to test the system. P2, P4, P9, and P10 asked whether the code they were examining occurred in classes that were only used to test the application. To answer this question, sometimes participants used tools for traversing the call hierarchy; using these types of tools allowed them to narrow their search to only locations the code of interest is being called. Compared to the tool support provided for determining general context, participants could answer this question relatively easily using existing tool support.

Tool Implications. It is not obvious what tool support for answering these kinds of questions would look like, especially when it is not obvious why participants wanted these bits of information. For example, one reason for needing this type of information is to better understand the code and likelihood that a bug is a vulnerability. However, it may be that participants merely wanted to know the answers to these questions with no high level goal in mind. Most of these questions can be answered using documentation, if any exists, written by the developers of the code; perhaps a tool that supports answering these questions would make developer written documentation more easily available and searchable for information of interest. When specifically helping developers determine what code is test code, this could be augmented to the notification so the developer is aware of that seemingly relevant detail up front.

F. *End-User Interaction (8)*

Questions in this category deal with how end users might interact with the system or a particular part of the system. Some participants wanted to know whether users could access critical parts of the code and if measures are being taken to mitigate potentially malicious activity. For example, while assessing the Potential Path Traversal vulnerability, participants wanted to know whether the path is sanitized somewhere in the code before it is used. Three of the 155 distinct questions we found fall into this category.

Is there input coming from the user? (4)
Does the user have access to this code? (4)
Does user input get validated/sanitized? (4)

Observations. Eight participants had questions regarding the level of interaction end-user have with the code of interest. Though none of these questions got asked more than others, some questions co-occurred for a given vulnerability. For example, when assessing the Potential Path Traversal vulnerability, P1 and P6 wanted to know if the input was coming from the user along with whether the input is being validated in the event that the input does come from the user. For these participants, finding the answer required manual inspection of surrounding and relevant code. For instance, when P6 manually inspected code and found a `Validator` method that he also manually inspected to determine if it was doing input validation.

When assessing the Potential Path Traversal vulnerability, four participants had questions regarding whether end-users had access to the part of the code being analyzed. P2 used the called hierarchy to answer this question by tracking where the method the code is contained in gets called; for him, the vulnerability only existed if the user had access to the code. P1 and P6 went on a similar mission and determined that because all the calls to the code of interest appeared to happen in methods called `testDataGenerator()`, the code should be fine as it is written. Though participants found the answers to their questions, it took time for them to get the answer using tools meant to answer other questions.

Tool Implications. Research exists that studies attack surfaces, or vectors that predict where an unauthorized user may be able to gain access to the system. The goal of their research area was to identify, measure, and reduce the size of attack surfaces [**?**], [**?**]. These types of tools can help answer *Is there input coming from the user?*, however, developers may lack tool support for navigating through and reasoning about attack surfaces to answer the others. Existing tools could extend their functionality to answer these questions by allowing developers to narrow in on points of interest in relation to the potential vulnerability being assessed. Answering *Does the user input get validated/sanitized?* may also require providing an explicit, perhaps visual, connection between the attack surface and the code the input touches.

VI. Results: Individuals

A. *Developer Planning and Self-Reflection (8)*

Another kind of question participants asked when assessing potential security vulnerabilities concerned their current status or plans for next steps in terms of assessing or removing the vulnerability. All of the questions in this category involve the developer's thoughts or the individual's relationship to the problem, rather than specifics of the code or the error notification. For example, when some participants thought they had an idea of what the problem might be, they stopped and took different measures to re-assess the code or notification to make sure they understood. Fourteen of the 155 distinct questions extracted fit into this category.

Do I understand? (3)
What should I do first? (2)
What was that again? (2)
Is this worth my time? (2)
Why do I care? (2)
What was I looking for? (1)
What do I know now? (1)
Have I seen this before? (1)
What's next? (1)
Where am I in the code? (1)
Is this my responsibility? (1)

Observations

MORE TO COME

Tool Implications

Though, in general, there is no trivial solution that can help developers answer many of these questions, there are ways to provide information or instructions developers can follow to more easily determine the answers for themselves. For example, one question participants asked was *Have I seen this before?* A tool that helps developers answer this question might keep track of previous times the developer has encountered, and perhaps fixed, this vulnerability. The tool could also include a link to the code where the vulnerability was located as well as a diff showing the changes the developer made to fix the vulnerability. Another question that a tool could potentially help answer is *What was that again?* A tool that would support answering this question could keep track of places in the code the developer visits, perhaps while they are using a tool like FSB, and make them easily accessible to the developer as they triage the code.

B. *Understanding Concepts (7)*

For some participants, the concepts or terms that the notification contained may not have been a part of their own mental model. In these situations, participants had questions surrounding one or more of the concepts relevant to the problem. For instance, while parsing the potential attacks listed in the notification for the Servlet Parameter vulnerability, some participants did not know what a CSRF token was. We categorized 6 of the 155 distinct questions

into this category.

What is this concept? (6)
How does this concept work? (4)
What is the term for this concept? (2)
Do these words have special meaning related to this concept/problem? (1)

Observations

Sometimes, the difficulty participants encountered was with the terminology used; others were not familiar with some of the general concepts being discussed. For example... We only collected general security vulnerability knowledge as a pre-study demographic so it is difficult to draw conclusions about the correlations between difficulty understanding the concepts being discussed and the participants' experience with that concept.

Tool Implications

In general, if a developer does not have experience with a particular feature or concept, they may have more difficulty understanding a problem in their code pertaining to that concept [**?**]. Not all developers have the same experiences, so it seems important that developers who do not understand the concepts involved in potential security vulnerabilities are able to easily find the means to gain that understanding. Sometimes, FSB provides links to information regarding the concepts involved. However, some participants did not use them because they did not know they existed until the first author pointed them out; because of this, they usually did a web search themselves to find more information.

A simple solution to providing easier access to the desired information is by placing links, when available, in a more visible area; one suggestion for placement is nested in the text of the error message next to the concept it has information about. One potential downfall to this solution is that developers that understand the concepts will likely not use the links, and maybe even find them distracting. Perhaps if tools had access to models that represent a developer's conceptual knowledge, similar to work done by Fritz and his colleagues to model developer familiarity with source code [**?**], tools could make an attempt at only adapting notifications if the developer's model suggests there is a need to answer these types of conceptual questions.

C. *Confirming Expectations (1)*

Is this doing what I expect it to?

Developers build and alter their mental models when exploring code [**?**], [**?**]. As they build and alter mental models, developers may seek confirmation that they fully understand the code and its intended function; for a subset of our participants, this was the case. We only identified one distinct question, however, there are many ways to interpret the questions and may require different pieces of information to answer it.

VII. Results: Problem Solving Support

A. *Resources and Documentation (10)*

Many participants found themselves in situations where they would use outside resources and documentation to decide how to proceed regarding a given vulnerability. For example, when assessing the Potential Path Traversal vulnerability, participants wanted to know what their team members would do or if they could provide any additional information regarding the vulnerability. Another kind of question we extracted pertained to the resources and documentation available regarding the vulnerability or source code. Ten of the 155 distinct questions fall into this category.

Can my team members/resources provide me with more information? (5)
Where can I get more information? (5)
What information is in the documentation? (5)
How do resources do to prevent or resolve this? (5)
Is this a reliable/trusted resource? (3)
What type of information does this resource link me to? (2)
What is the documentation? (1)

Observations

maybe talk about participants clicking links, sometimes missing them, sometimes not clicking them because not sure what kind of information it provides (or found themselves clicking things thinking they would find what they are looking for, but don't.

Tool Implications

Some tools link to external documentation that developers can use to attempt to answer some of these questions, FSB being one of them. Though links to external documentation and resources can provide answers to some of these questions, they do not help answer questions concerning the reliability of the resource nor do they provide developers with the ability to locate and use other external resources such as team members or others who have experience with the relevant code. It seems reliability of the resource would be especially important for making changes based on the resource in a secure system.

Lack of support for answering these questions, especially questions concerning reliability, may have contributed to participants not clicking the links in many situations. Tools might consider including information regarding the reliability of the resources provided; one way to do this is to associate the link with peers who have visited it. Also, to help answer questions about other team members' knowledge of the code, tools may be able to borrow ideas from previous research. Fritz and colleagues developed a degree of knowledge model that predicts how familiar a developer is with various source code elements, however, this model has not been operationalized in a tool that allows developers to have access to the degree of knowledge values for various developers on the code segment being analyzed [**?**].

B. *Understanding and Interacting with Tools (8)*

Throughout the study participants interacted with a variety of tools including FSB, the call hierarchy tool, and find references. While interacting with these tools, participants asked questions about how to access specific tools, how to properly use the tool, and how to interpret tool output. Nine of the 155 distinct questions we extracted fall into this category.

Why is the tool complaining? (3)
Can I verify the information the tool provides? (3)
What is the tool's confidence? (2)
What is the tool output telling me? (1)
What is the tool keybinding? (1)
What tool do I need for this? (1)
How is the information presented by the tool organized? (1)

Observations

Some of the questions we extracted seemed to deal with having access to or knowing how to use the tools needed to complete a certain task. Participants sometimes found themselves in situations where they needed information or to take action but could not determine how to invoke the tool or possibly did not know what tool they would use at all. This seems to point to a tool awareness or education problem.

Other questions dealt with the presentation of information provided by the tool. Participants wanted to be able to explore things about the tool's output, and typically had to manually determine the answers to their questions. For example, FSB reports its confidence on the potential defect in the notification, however, some participants found it difficult to process this information.

Tool Implications

Developers' ability to understand and effectively interact with tools has been the focus of many research studies, however, developers may still have difficulty answering some of the questions they have when using existing tools [**?**], [**?**], [**?**]. Many of these questions may require the input of other developers to be answered effectively. For example, research suggests that developers are more likely to trust and use a tool that has been recommended by one of their peers [**?**]. Research also suggests that through concise peer interaction, developers more effectively discover new tools [**?**]. A tool that supports answering these questions might allow developers to easily access information regarding peer interactions with the tools being used; the tool might also include advice based on mistakes the developer's peers have made.

C. *Bug Severity and Ranking (5)*

Some participants had questions regarding the severity or ranking of the bug being reported by FSB. These questions dealt with general bug severity questions and questions pertaining specifically to how the tool communicates severity and ranking. Four of the 155 distinct questions extracted fall into this category.

How serious is this bug? (2)
How do the rankings compare? (2)
What do the bug rankings mean? (2)
Are all these bugs the same severity? (1)

Observations

Tool Implications

Previous research on static analysis tools also found that developers may not always be able to understand the rankings or severity labels tools use, so it is not surprising this category emerged from our questions [**?**]. Techniques and tools have been developed for static analysis alert prioritization that can be used to reason about the severity and ranking of tool notifications [**?**], [**?**], [**?**]. It seems these kinds of tools would be useful when attempting to answer, or even eliminate, these kinds of questions. Based on our observations, another way to answer or prevent these questions is by using simple, one-dimensional metrics for reporting the severity or ranking of a potential vulnerability. For example, instead of FSB using phrases and numbers to represent severity, priority, and rank, perhaps FSB could use one scale where one value can tell you how important the vulnerability is in comparison to the space of other vulnerabilities.

D. *Error Messages (6)*

As expected, participants asked questions concerning the content of the error messages attached to the notifications. Mostly, this happened when participants referenced the bug information view offered by FSB to get a better understanding of the potential vulnerability. Three of the 155 distinct questions we extracted fall into this category.

What does the error message say? (5)
What is the relationship between the error message and the code? (2)
What code caused this error message to occur? (2)

Observations

More surprisingly, they also asked questions about how to relate information contained in the error message back to the code. For example, the predictable random vulnerability notes that a predicable random value is bad when being used in a secure context. Many participants attempted to related this piece of information back to the code by looking to see if anything about the code where the potential vulnerability was found suggests it is in a secure context. In this situation, the fact that the method the vulnerability was found in was called `randomPassword()` suggested to participants that the code was in a secure context and therefore a vulnerability that should be resolved.

P1 – It's called random password, so pretty big hit that I should probably change it to secure random like it asks

Tool Implications

Determining what the error message says is a relatively simple action for the developer to complete; most tools, FSB included, provide functionality for developers to access the details of a given error message. Tools also attempt to answer the question *what code caused the error message* by placing an icon in the gutter of the coding editor at presumably the location of the vulnerable code. However, if the error message is connected to multiple parts of the code, or a different part of the code all together, the icon location is not the best indicator of the connection between the code and the error message. Research has attempted to help developers make stronger connections between the error message they receive and the code relevant to it using visual overlays in the code editor [**?**]. These type of visual overlays may also be useful for answering the questions in this category.

VIII. Future Work

IX. Conclusion

This paper reported on an experiment conducted to discover the questions developers ask when identifying and assessing security vulnerabilities in software code. During the experiment, we asked 10 software developers to walk us through their thoughts as they assessed potential security vulnerability found in iTrust, a medical records web application. We presented the results of our experiment as a catalog of categorized questions developers ask when assessing security vulnerabilities. Our findings have several implications for the design of tools that can be used when attempting to answer the questions developers have, such as providing support for reason about possible system attacks.

Acknowledgment

We would like to thank our study participants. Special thanks to Xi Ge, Anthony Elliott, and the Developer Liberation Front [12] for their assistance. This material is based upon work supported by XXX and a National Science Foundation Graduate Research Fellowship under Grant No. DGE–0946818.

[12]http://research.csc.ncsu.edu/dlf/