Security-Related Questions Developers Ask About Their Code

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

***Abstract*—New tools and approaches should help developers answer the security-relevant questions that they would otherwise struggle with. 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. Our work identifies the security-related questions developers ask, explores the strategies that developers use to answer those questions, and assesses the existing tool support for those strategies. We contribute by informing toolsmiths of the security-related questions that developers ask.**

We identified developers' knowledge requirements by conducting 10 semi-structured interviews with novice and experienced software developers, observing their interactions with security vulnerabilities in iTrust, a Java medical records software system. From these interactions, we extracted 559 questions, which we reduced to 155 unique questions that fit into 17 categories.

I. Introduction

When analyzing source code, especially defective code, research suggests that during this knowledge acquisition process, developers hypothesize about the code, specifically by asking questions [1], [2]. There exists research that attempts to categorize and catalog the questions developers ask when coding [3]. Research on tool design and developer productivity have used available catalogs to design tools that support developers when attempting to answer these questions [4], [5].

Professionals working on security-critical applications also could benefit from improved tool support in their work environments. There exist tools that help developers detect potential security flaws in their code. However, research suggests they may not be using them because of difficulty interpreting the output [6]. Part of this difficulty for security developers may stem from the fact that some of the hard to answer questions developers have pertain to the security of their code [3].

Despite the importance of security, to our knowledge, no research exists that catalogs the specific security-related questions developers ask. Therefore, we lack a framework for assessing whether existing tools fully address the knowledge requirements of developers.

Research that determines the questions developers ask about their code has informed and motivated toolsmiths in other domains to develop better tools [4], [5], [7]. Our work extends these promising results to security by informing toolsmiths of the security-related questions that developers ask. In this paper, we report a study conducted with 10 developers familiar with the iTrust software system.[1] We observed each developer as they assessed potential security vulnerabilities in the code and report the kinds of questions they asked while doing so. 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 Section IV and implications of our findings in Section V. Finally, we conclude the paper with a discussion of future work VI.

II. Related Work

We have organized the related work into three sections. Section II-A outlines the predominant approaches researchers 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 [1], [8], [9].

Much research has evaluated the effectiveness of tools and techniques based on their false positive rates and how many vulnerabilities they detect [9]–[11]. 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 [10]. They considered their tool effective because of its low false positive rate (50%) and its ability to find vulnerabilities previously unknown. Similarly, Livshits

[1] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start

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 [1]. 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 [9]. 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 [11]. 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 other, non-security domains, research has shifted focus from the technical effectiveness of program analysis tools to usability [6], [12], [13]. 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 [6], [12], [13]. 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 [12], [14].

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 [13].

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 [6]. 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 [15]. 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, they do not pay due diligence to how developers interpret the reporting of security vulnerabilities. It is particularly important to consider security tools and vulnerabilities, because security vulnerabilities are more likely to cause incidents that affect company budgets as well as the end users [16]. Research also suggests that some of the harder questions to answer when analyzing code revolve around security and the implications of changes made to the code on security [3]. 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, however few such studies have focused specifically on the knowledge requirements of developers when assessing security vulnerabilities [3], [17], [18].

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 [3].

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 [18].

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 [2]. 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 [19].

These studies explore the questions developers ask when building and debugging software, however, they do not study the questions developers ask when they have a potential security vulnerability in their code. In fact, our work builds on the work of LaToza and Myers, which found that some of the hard-to-answer questions developers have pertain to 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 10 semi-structured interviews with software developers. In our analysis, we extracted and categorized the

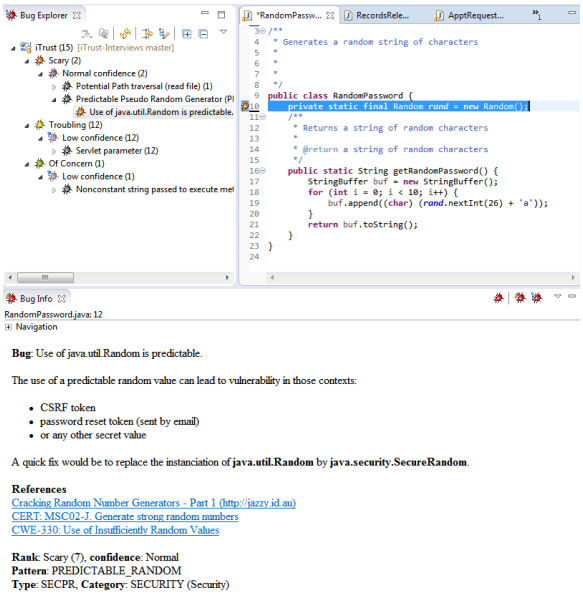

Fig. 1. The environment participants were presented with.

questions developers asked during these interviews. Section III-A outlines the research question we sought to answer. Section III-B details how the study was designed. Section III-C describes how we preformed data analysis.

A. Research Goals

We are motivated by the following research question:

- **RQ1**: What types of questions do developers when assessing the security of their code?

In our investigation, we also identified some of the strategies that developers use to answer such questions and assessed whether current tools support those strategies.

B. Study Design

Each interview 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 interview concluded with several demographic and open-ended discussion questions.

1) Materials: Participants used Eclipse to explore vulnerabilities in iTrust, a Java medical records application that ensures the privacy and security of patient records according to the HIPAA statue.[2] 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 an extended version FindBugs, Find Security Bugs.[3] We chose Find Security Bugs as a representative tool by comparing the available source code analysis tools listed by NIST, OWASP, and WASC. Figure 1 depicts the configuration of the IDE for one of the tasks.

[2]hhs.gov/ocr/privacy/
[3]http://h3xstream.github.io/find-sec-bugs/

2) Participants: We interviewed a total of 10 software developers, 6 students and 4 professionals. 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 few new questions were emerging from each additional interview.

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.

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 participant's responses.

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

FindBugs and Find Security Bugs group similar error messages using a system of tags. For example the Null Pointer tag (NP) applies to notifications that warn about the misuse of null. To cover a broader set of question topics, we selected vulnerabilities with four different tags. Find Security Bugs identified three types of naturally occurring vulnerabilities: cross site scripting, path traversal, and predictable random vulnerabilities. We inserted a SQL injection vulnerability by making minimal modifications to one of the database access objects. Our modifications preserved the functionality of the original code and were based on examples of SQL injection presented by OWASP. Table II summarizes each of the four vulnerabilities.

During the briefing section, participants were told that they were 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. Specifically, they were asked to, "Say any questions or thoughts that cross your mind regardless of how relevant you think they are."

The interviewer was equipped with the following questions, but had the freedom to omit questions or 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?

TABLE I
DEMOGRAPHICS OF STUDY PARTICIPANTS

Participant	Job Title	Security Vulnerability Familiarity	Programming Experience (years)
P1	Student	2.5/5	4.5
P2	Test Engineer	3/5	8
P3	Development Tester	2/5	6
P4	Software Developer	2/5	6
P5	Student	4/5	10
P6	Student	1/5	4
P7	Front-end Software Developer	4/5	4.5
P8	Student	3/5	7
P9	Software Consultant	3/5	5
P10	Student	3/5	8

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

- 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 interview data using a grounded theory approach [20]. First, we transcribed all the audio-video files using oTranscribe.[4] 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 interview 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 [21]. A statement was coded as a question only if it met one of the following criteria:

- **The participant explicitly asks a question.**
  P2: *Why aren't they using Prepared Statements?*
- **The participant makes a statement and explores the validity of that statement.**
  P6: *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."**
  P8: *I don't know that it's a problem yet.*
- **The participant clearly expresses uncertainty over a statement.**
  P10: *Well, it's private to this object, right?*
- **The participant clearly expresses a knowledge requirement by describing plans to acquire information.**
  P1: *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 interview. When we identified a statement that satisfied one or more of the criteria, we marked the transcript, highlighting the participant's original statement, and clarified the question being asked. From the 10 interviews, the first coder extracted 421 question statements; the other coder extracted 389.

Since participants used a think-aloud protocol, the interviews contained many statements that were difficult to classify as questions. The criteria help ensure that only the statements that reflected actual questions were classified as such. The interviews were coded by two independent reviewers to help identify more of the questions being asked, especially the difficult to interpret 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 interview 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 interview 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.

[4] otranscribe.com

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

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

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.

4) Question Sorting: To organize our questions and facilitate discussion, we preformed an *open* card sort [22]. 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 preformed 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 [23].

IV. Results

We list each distinct question we extracted and the count for each in parentheses. For each category, we discuss the common thread that ties questions in that category together. We also contextualize each category by discussing existing tool support and shortcomings.

A. *Code Background and Functionality (17)*

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

Data Overview

Participants also asked questions concerning the background and intended function of the code being analyzed. Nine of the 155 distinct questions extracted fit into this category. The questions in this category inquire about the code at an abstract level, such as determining the role a component or piece of code plays in the entire system or program.

Observations

P1 – It's code that's actually written by someone writing iTrust, so it's not library code.

Tool Implications

Answering many of these questions would require aggregated data, such as documentation that programmers contribute to as the code changes or measuring the amount of effort that

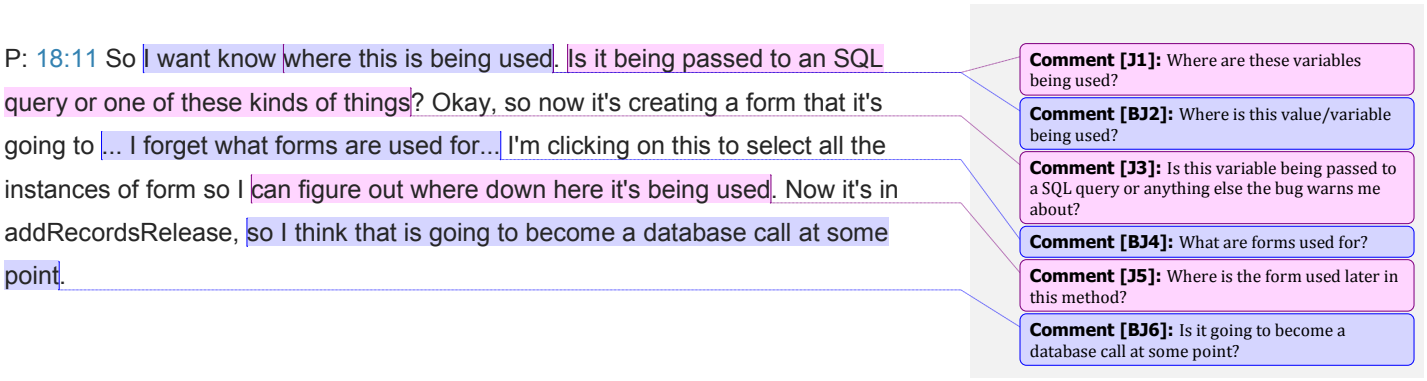

Fig. 2. Question merging process

TABLE III
EMERGENT CATEGORIES FROM CARD SORT

Category	Count	Location in Paper	Example
Code Background and Functionality	17	Section IV-A	*Example question...*
Developer Planning and Self-Reflection	14	Section IV-B	*Example question...*
Control Flow and Call Information	13	Section IV-C	*Example question...*
Data Storage and Flow	11	Section IV-D	*Example question...*
Understanding Alternative Fixes and Approaches	11	Section IV-E	*Example question...*
Locating Information	11	Section IV-F	*Example question...*
Preventing and Understanding Potential Attacks	11	Section IV-G	*Example question...*
Resources and Documentation	10	Section IV-H	*Example question...*
Application Context/Usage	9	Section IV-I	*Example question...*
Understanding and Interacting with Tools	9	Section IV-J	*Example question...*
Assessing the Application of the Fix	9	Section IV-K	*Example question...*
Understanding Concepts	6	Section IV-L	*Example question...*
Bug Severity and Rank	4	Section IV-M	*Example question...*
Relationship Between Bugs	3	Section IV-N	*Example question...*
End-User Interaction	3	Section IV-O	*Example question...*
Error Messages	3	Section IV-P	*Example question...*
Confirming Expectations	1	Section IV-Q	*Example question...*
Uncategorized	10		

has been put into the code thus far. This means that without proper tool support, developers interested in this information would have to seek out resources that have the information they need. Once they find the resource, they would have to find any and all information relevant to the program entity they are investigating, all while hoping the information is up to date and accurate. A tool that wants to accommodate these needs might collects these sorts of measures and provide developers easy access to it.

B. *Developer Planning and Self-Reflection (14)*

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

Miscellaneous

Data Overview

One kind of question participants asked when assessing potential security vulnerabilities in the code concerned their current status or plans for next steps in terms of understanding or assessing the vulnerability. Nine of the 155 distinct questions extracted fit into this category. 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. The most common occurring question in this category was...

Observations

Tool Implications Though these general questions may not be trivial to help developers answer, there may be 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.

C. *Control Flow and Call Information (13)*

What is the call hierarchy?
How can I get calling information?
Who can call this?
Where is the method being called?
What causes this to be called?
Are all calls coming from the same class?
What gets called when this method gets called?
How often is this code called?
Miscellaneous

Data Overview

We also extracted questions relating to control flow and calling information. Developers seek information information pertaining to the methods that get called, or do not get called, in the code. As the name of this category suggests, these questions can generally be answered using a tool for viewing the call hierarchy, when available. Eight of the 155 distinct questions extracted dealt with acquiring control flow and call information.

Observations

Eclipse includes a call hierarchy tool that, given a method, displays its callers and all its callees. The tool allows users to easily traverse the call hierarchy. While some participants located and employed the call hierarchy tool, others gravitated toward other more error-prone approaches, such as the 'find references' tool or manual visual inspection. In some cases, participants were unaware of the call hierarchy tool or how to use it. Others were predisposed to favor light-weight tools, such as inspecting code highlights, rather than navigate to and invoke the call hierarchy tool. It is unclear whether the call hierarchy tool is more efficient than tools like 'find references' for answering these kinds of questions, however participants who sought call information without the call hierarchy tool were more likely to make faulty assumptions or miss method calls.

P1 – Yeah, I don't know what this is, call hierarchy.

Tool Implications

Even with tools like 'find references' and the call hierarchy view tool, it may be difficult to easily/quickly answer questions like *how can I get calling information* and *who can call this* without having to manually, and mentally, aggregate the information gathered from each tool. Further, even tools like call hierarchy do not provide information on how parameters are modified between method calls. 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 desire. 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.

D. *Data Storage and Flow (11)*

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

Data Overview

Some of the questions we found 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. Eight of the 155 distinct questions fit into this category.

Observations

P1 – I haven't verified that. You might be modifying the string as it goes through. But it's probably the same string..

Tool Implications

The call hierarchy tool in Eclipse provides a way of understanding control flow, but this static information does not give an idea of how data travels when the program is run. The developer would have to retrieve this information, either from the documentation, by debugging the code, or some combination of both. There exists tools that do data flow analysis and report defects to programmers based on said analysis, however, they do not allow developers to explore the flow of data through their programs [10]. 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 [24], [25]. It may be that some of the existing tools for data flow analysis could be augmented with functionality for exploring a visual version 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.

E. *Understanding Alternative Fixes and Approaches (11)*

Why should I use this alternative method/approach to fix the vulnerability?
What are the alternatives for fixing this?
Does the alternative function the same as what I'm currently using?
When should I use the alternative?
Is the alternative slower?
Are there other considerations to make when using the alternative(s)?

How does my code compare to the alternative code in the example I found?
Miscellaneous

Data Overview

Sometimes participants looked elsewhere for information about their code or vulnerability. Primarily, participants used web resources or searched for similar code elsewhere in the project. When presented with alternative approaches or solutions, participants compared the alternative with the current code and assessed the applicability of the alternative. Seven of the 155 questions we extracted pertained to understanding the various facets of alternative approaches or solutions.

Observations

The process of understanding the alternative options was complicated by the variety of sources, such as StackOverflow, official documentation, and personal blogs, that all presented information in different formats. Often the candidate alternative did not readily provide meta-information about trade-offs or the process of applying suggestions. Many participants seemed to have a preference for StackOverflow as a means for understanding alternative approaches and fixes.

Tool Implications

When elaborating a suggested fix, or alternatives to achieving the same functionality, developers may have difficulty determining at a glance the relevant details regarding the suggested approach (as the questions in this category suggest). One way a tool might help address this would be by noting other places in the code the alternative the developer is considering has been used, if they exist. If the developer, or other developers of the code, have not used the alternative before, the tool could borrow behavior or design principles from WHYLINE by allowing developers to ask questions about the code in relation to the suggested approach [2]. The tool could then potentially use speculative analysis, which is used by tools like QUICK FIX SCOUT, to present trade-offs of using the suggested fix or keeping the code as it is [26].

F. ***Locating Information (11)***

Where is this used in the code?
Where are other similar pieces of code?
Where is this artifact?
Is this artifact located in this class?
Where is this method defined?
Where is this class?
Where is the next occurrence of this variable?
How do I track this information in the code?
How do I navigate to other open files?
Miscellaneous

Data Overview

Some participants asked questions regarding locating, or the ability to locate, information in their coding environment. Nine questions of the 155 extracted fit into this category. The distinction between this category and ***Code Background and Function*** is that this category includes questions pertaining to searches made in the coding environment for information, artifacts or code fragments. On the other hand, ***Code Background and Function*** pertains to understanding the code at a higher level that would not be answered by triaging the code.

Observations

P1 – But, I assume that it is coming from, lets see

Tool Implications

The answers to most of these questions can be found inside the IDE, which makes it more feasible for a tool to attempt to answer it. For example, Eclipse has an 'open declaration' tool that can be used to find a method's definition. A few of these questions, however, would require more effort on the developer's part to answer, therefore could benefit from tool support. For instance, it is not obvious how a tool would answer the question *how do I track this information in the code*; the answer could vary depending on what information the developer is trying to track. Similar to the suggestion made in Section IV-C, a tool that can automatically detect when a developer is attempting to track specific information could help by either helping the developer track the information of interest or pointing out a tool that can.

G. ***Preventing and Understanding Potential Attacks (11)***

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

Data Overview

We also extracted questions pertaining to the potential attacks that could arise from the vulnerability found in the code. These questions aimed to understand the potential attacks that would exploit a given vulnerability, how to execute those attacks, how to prevent those attacks now and in the future. Nine of the 155 distinct questions we gathered belong in this category.

Observations

P1 – doesn't give me much confidence that this is actually a real bug.. P2 – I'm thinking about different types of attacks

Tool Implications

Some FindBugs notifications attempt to provide developers with the means to answer some of these questions by providing links to relevant information. Many of the links provided

linked to information regarding why the code may be broken, however, do not provide information to improve understanding of what the potential attacks are.

Some of the questions, such as *how do I find out if this is a real vulnerability*, may not be as simple to answer by providing a link. This kind of question may require triangulation of information; some information from the web on the vulnerability itself and the potential attacks, and some information from fellow developers who may better understand the likelihood the bug is a vulnerability in their system. If there are resources, on the web or otherwise, that developers could use it does not seem that they are easy or intuitive to locate. One way tools can help is by providing easy access to the top web resources and developers to use when assessing the vulnerability; many of our participants preferred StackOverflow as a resource and a degree of knowledge model, like the one proposed by Fritz and his colleagues, could provide the developers most familiar with the code [27].

H. ***Resources and Documentation (10)***

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

Data Overview

Another kind of question we extracted pertained to the resources and documentation available regarding the vulnerability or source code. Many participants found themselves in situations where they would use outside resources to decide how to proceed. Seven of the 155 distinct questions fall into this category.

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, Find Security Bugs 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. Lack of support for answering these questions may have contributed to participants not clicking the links in many situations. 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 [27].

I. ***Application Context/Usage (9)***

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

Data Overview

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. Seven of the 155 distinct questions revolved around how the code works in the context of a portion of or the entire system.

Observations

Though all participants had experience coding in iTrust, and had some knowledge about the codebase, many still encountered situations where they wanted to know the usage or context of a segment of code within the application.

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.

J. ***Understanding and Interacting with Tools (9)***

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

Data Overview

Throughout the study participants interacted with a variety of tools including Find Security Bugs, the call hierarchy tool, and find references, for example. While interacting with these tools, participants asked questions about how to access specific tools and how to interpret their output. Seven of the 155 distinct questions we extracted fall into this category.

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, Find Security Bugs reports its confidence on the potential defect in the notification, however, some participants found it difficult to process this information.

Tool Implications

There exists a large body of research that explores developers' ability to understand and effectively interact with tools, however, many of these questions remain unanswered [2], [6], [13]. 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 [28]. Research also suggests that through concise peer interaction, developers more effectively discover new tools [29]. A tool that could model this concise interaction could potentially provide answers to many of these questions.

K. ***Assessing the Application of the Fix (9)***

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

Data Overview

Another kind of question we extracted pertained to the process resolving vulnerabilities. These questions pertain to understanding the application and implications of a potential fix. We categorized 9 of the 155 distinct questions into this category.

Observations

P1 – is there a quick fix/automatic application?

Tool Implications Though it is not clear how feasible it is to answer some of these questions, some can be answered by using a similar approach to that used by Muşlu and colleagues when developing Quick Fix Scout [26]. Their tool attempts to help developers answer the question *Does this fix introduce other bugs?* None of our participants asked this question specifically. However, questions like *How hard is it to apply?* and *Does the code stand up to additional tests prior to/after applying the fix?* could possibly be answered using a similar process.

L. ***Understanding Concepts (6)***

What is the term for this concept?
Do these words have special meaning related to this concept/problem?
How does this concept work?
What is this concept?
Miscellaneous

Data Overview For some participants, the concepts that the notification contained were incompatible with the developer's mental model. In these situations, participants seemed to have questions surrounding one or more of the concepts relevant to the problem. We categorized 6 of the 155 distinct questions into this category.

Observations

Sometimes, the difficulty participants encountered was with the terminology used; others were not familiar with some of the general concepts being discussed. In general, it seems that if a developer does not have experience with a particular feature or concept, it is less likely they will understand a problem in their code pertaining to that concept [].

Tool Implications

M. ***Bug Severity and Ranking (4)***

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

Data Overview

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

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 [6].

*N. **Relationship Between Bugs (3)***

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

Data Overview Some participants asked questions about the connections between co-occurring vulnerabilities and whether or not similar vulnerabilities exist elsewhere in the code. Three of the 155 distinct questions we found target the relationships between the vulnerabilities in the code.

Observations
Tool Implications

We speculate that developers might have these kinds of questions because they want to know if a vulnerability like the one they are assessing exists in other similar code fragments. Another reason for asking this kind of question is to determine the possibility of eliminating more than one vulnerability with one fix. Whatever the motivation may be, participants did not seem to have an intuitive or easy way of going about finding the answer to the questions asked in this category.

*O. **End-User Interaction (3)***

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

Data Overview

Questions in this category deal with how the end user might interact with the system. Participants wanted to know whether users could access critical parts of the code and if measures are being taken to mitigate potential malicious activity. Three of the 155 distinct questions we found fall into this category.

Observations

Many participants spent time and effort exploring the call hierarchy of the potential path traversal only to discover that the code was entirely contained in test classes.

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 [30], [31]. To our knowledge, developers lack tool support for navigating and reasoning about attack surfaces.

*P. **Error Messages (3)***

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

Data Overview

As expected, participants asked questions concerning the content of the error messages. Mostly, this happened when participants needed to look at the bug information to get a better understanding of the potential vulnerability. Three of the 155 distinct questions we extracted fall into this category.

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

*Q. **Confirming Expectations (1)***

Is this doing what I expect it to?
Developers build and alter their mental models when exploring code [32], [33]. 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.

V. Discussion

Data suggests that tools should focus on XYZ. Discuss interesting/contradictory categories and implications.

COME UP WITH SECTION HEADERS

VI. Future Work

VII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

REFERENCES

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

[2] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004, pp. 151–158.

[3] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 8.

[4] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of fine-grained code change history," in *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*. IEEE, 2013, pp. 119–126.

[5] F. Servant and J. A. Jones, "History slicing: assisting code-evolution tasks," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 43.

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

[7] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes, "Automatically locating relevant programming help online," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 2012, pp. 127–134.

[8] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," in *ACM SIGPLAN Notices*, vol. 40, no. 10. ACM, 2005, pp. 365–383.

[9] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 97–106.

[10] N. Jovanovic, C. Kruegel, and E. rda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 6–pp.

[11] L. Dukes, X. Yuan, and F. Akowuah, "A case study on web application security testing with tools and manual testing," in *Southeastcon, 2013 Proceedings of IEEE*. IEEE, 2013, pp. 1–6.

[12] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *Software, IEEE*, vol. 25, no. 5, pp. 22–29, 2008.

[13] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "Path projection for user-centered static analysis tools," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2008, pp. 57–63.

[14] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 workshop on Defects in large software systems*. ACM, 2008, pp. 1–5.

[15] L. Layman, L. Williams, and R. S. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 176–185.

[16] H. Chen and D. Wagner, "Mops: an infrastructure for examining security properties of software," in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 235–244.

[17] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering." in *ICSE*, 2014, pp. 12–13.

[18] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 185–194.

[19] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 175–184.

[20] B. G. Glaser and A. L. Strauss, *The discovery of grounded theory: Strategies for qualitative research*. Transaction Publishers, 2009.

[21] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[22] W. Hudson, *Card Sorting*. Aarhus, Denmark: The Interaction Design Foundation, 2013.

[23] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[24] S. Ghosh, "Method for generating a java bytecode data flow graph," May 15 2001, uS Patent 6,233,733.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[26] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 669–682, 2012.

[27] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 385–394.

[28] G. C. Murphy and E. Murphy-Hill, "What is trust in a recommender for software development?" in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 57–58.

[29] E. Murphy-Hill and G. C. Murphy, "Peer interaction effectively, yet infrequently, enables programmers to discover new tools," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011, pp. 405–414.

[30] P. K. Manadhata and J. M. Wing, "An attack surface metric," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 371–386, 2011.

[31] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: an application to android," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 274–277.

[32] J. J. Cañas, M. T. Bajo, and P. Gonzalvo, "Mental models and computer programming," *International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 795–811, 1994.

[33] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Mental representations constructed by experts and novices in object-oriented program comprehension," in *Proceedings of the Human-Computer Interaction Conference*. Springer, 1997, pp. 339–346.