Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis

Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford

Abstract—Security tools can help developers answer questions about potential vulnerabilities in their code. A better understanding of the types of questions asked by developers may help toolsmiths design more effective tools. In this paper, we describe how we collected and categorized these questions by conducting an exploratory study with novice and experienced software developers. We equipped them with Find Security Bugs, a security-oriented static analysis tool, and observed their interactions with security vulnerabilities in an open-source system that they had previously contributed to. We found that they asked questions not only about security vulnerabilities, associated attacks, and fixes, but also questions about the software itself, the social ecosystem that built the software, and related resources and tools. For example, when participants asked questions about the source of tainted data, their tools forced them to make imperfect tradeoffs between systematic and ad hoc program navigation strategies.

1 Introduction

Software developers are a critical part of making software secure, a particularly important task considering security vulnerabilities are likely to cause incidents that affect company profits as well as end users [4]. When software systems contain security defects, developers are responsible for fixing them.

To assist developers with the task of detecting and removing security defects, toolsmiths provide a variety of static analysis tools. One example of such a tool is Find Security Bugs (FSB) [38], an extension of FindBugs [37]. FSB locates and reports on potential software security vulnerabilities, such as SQL injection and cross-site scripting. Other tools, such as CodeSonar [34] and Coverity [35], can also be used to detect and remove potential security vulnerabilities. In fact, toolsmiths have created over 50, both free and commercial, static analysis tools to help developers secure their systems [33], [42], [44].

These tools provide, for instance, information about the locations of potential SQL injection vulnerabilities. Unfortunately, despite their availability, research suggests that developers do not use static analysis tools, partially because the tools provide information that does not adequately align with their information needs [12].

Our work addresses this problem by advancing our understanding of developers' information needs while interacting with a security-focused static analysis tool. To our knowledge, no prior study has specifically investigated developers' information needs while using such a tool. As we show later in this paper, developers need unique types of information while assessing security vulnerabilities, like information about attacks.

In non-security domains, work that identifies information needs has helped toolsmiths both evaluate the effectiveness of existing tools [1], and improve the state of program analysis tools [16], [30], [32]. Similarly, we expect that categorizing developers' information needs while using security-focused static analysis tools will help researchers evaluate and toolsmiths improve those tools.

INSERT NEW STUFF HERE!

To that end, we conducted an exploratory study with ten developers who had contributed to iTrust [40], a security-critical Java medical records software system. We observed each developer as they assessed potential security vulnerabilities identified by FSB. We operationalized developers' information needs by measuring questions — the verbal manifestations of information needs. We report the questions participants asked throughout our study and discuss the strategies participants used to answer their questions. Using a card sort methodology, we sorted 559 questions into 17 categories. The primary contribution of this work is a categorization of questions, which researchers and toolsmiths can use to inform the design of more usable static analysis tools.

Close relationship between information needs and assumptions. Assumptions can satisfy information needs. Can't figure something out? Make an assumption. Assumptions also drive information needs. Need information to validate assumptions. Made an assumption? See if it's right.

Reasons to look at assumptions:

1) Making incorrect assumptions causes developers to seek unnecessary information. This is the idea of "going down the rabbit hole." Ex. Assume the random number generator is implemented incorrectly. Start debugging the code that comprises that module, when really it was called with the wrong parameter. Wasting developer's resources – time.

2) Incorrect assumptions not only cause developers to explore the wrong paths, but can lead them to make incorrect conclusions. Assume the code is just called from tests. Conclude it is safe.

3) Correct assumptions suggest alignment between mental model and reality. Proxy for measuring success. Given the correct assumptions, the information you seek will be relevant to the solution.

- *M. Shell was with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332. E-mail: see http://www.michaelshell.org/contact.html*
- *J. Doe and J. Doe are with Anonymous University.*

2 Previous Work

In a preliminary version of this work [31] we focused solely on developers' information needs. In this work, we also consider the approaches developers take to acquire the information they need. This paper contains the four following significant additions to the previous paper: (1) Two new research questions, RQ2 and RQ3 in Section 4.1 (2) A more descriptive explanation of the study tasks in Section 4 (3) An expanded discussion of related work in Section 3 (4) A more concrete depiction of the notifications described in Section 7.2

3 Related Work

We have organized the related work into two subsections. Section 3.1 outlines some of the current approaches researchers use to evaluate security tools and Section 3.2 references other studies that have explored developers' information needs.

3.1 Evaluating Security Tools

Using a variety of metrics, many studies have assessed the effectiveness of the security tools developers use to find and remove vulnerabilities from their code [2], [22], [23].

Much research has evaluated the effectiveness of tools based on their false positive rates and how many vulnerabilities they detect [2], [5], [13]. For instance, Jovanovic and colleagues evaluate their tool, Pixy, a static analysis tool that detects cross-site scripting vulnerabilities in PHP web applications [13]. They considered Pixy 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 security-oriented static analysis, which creates static analyzers based on inputs from the user [22]. 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 [2]. 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 vulnerability-finding techniques [5]. They found combining manual testing and static analysis was most effective, because it located 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, these studies do not evaluate whether the tools address developers' information needs. Unlike existing studies, our study focuses on developers' information needs while assessing security vulnerabilities and, accordingly, provides a novel framework for evaluating security tools.

3.2 Information Needs Studies

Several studies have explored developers' information needs outside the context of security. Similar to our work, some existing studies determine these needs by focusing on the questions developers ask [14], [18], [19]. These three studies explore the questions developers ask while performing general programming tasks. In contrast to previous studies, our study focuses specifically on the information needs of developers while performing a more specific task, assessing security vulnerabilities. Unsurprisingly, some of the questions previously identified as general programming questions also occur while developers assess security vulnerabilities (e.g., Sections 5.3.2 and 5.3.3).

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 [19]. Furthermore, 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 [15]. They found that developers were able to complete more debugging tasks while 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 [7].

4 Methodology

We conducted an exploratory study with ten software developers. In our analysis, we extracted and categorized the questions developers asked during each study session. Section 4.1 outlines the research questions we sought to answer. Section 4.2 details how the study was designed and Sections 4.3, 4.4, and 4.5 describe how we performed our three phases of data analysis. Study materials can be found online [36].

4.1 Research Question

We want to answer the following research questions:

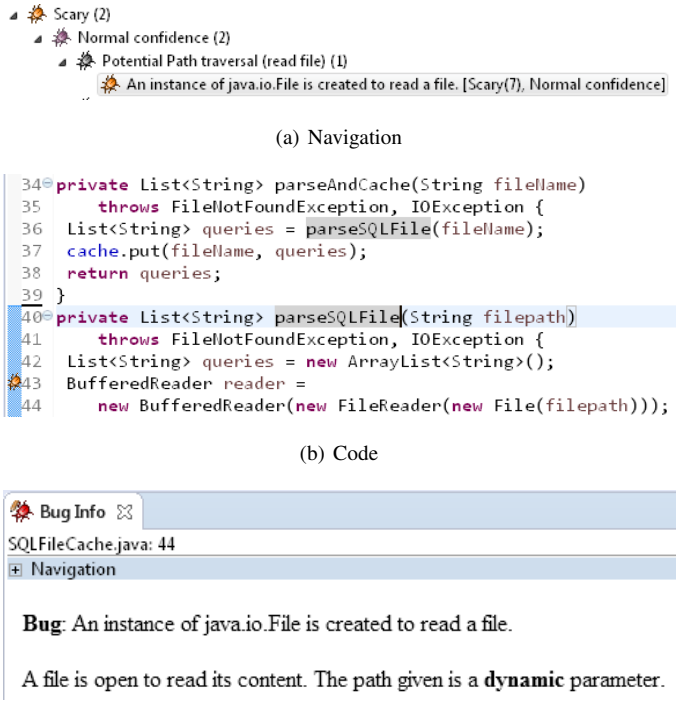
- **RQ1**: What information do developers need while using static analysis tools to diagnose potential security vulnerabilities?
- **RQ2**: What (successful and unsuccessful) strategies do developers use to acquire the information they need?
- **RQ3**: What (correct and incorrect) assumptions do developers make while executing these strategies?

We measured developers' information needs (RQ1) by examining the questions they asked. The questions that we identified are all available online [36]. We list several exemplary questions throughout Section 5 along side the strategies (RQ2) developers used to answer those questions. Finally, in Section 6 we discuss the assumptions (RQ3) developers made while executing their strategies.

4.2 Study Design

To ensure all participants were familiar with the study environment and Find Security Bugs (FSB), each in-person session started with a five-minute briefing section. The briefing section included a demonstration of FSB's features and time for questions about the development environment's configuration. During the briefing section, we informed participants of the importance of security to the application and that the software may contain security vulnerabilities.

Additionally, we asked participants to use a think-aloud protocol, which encourages participants to verbalize their thought



(a) Navigation

(b) Code

(c) Short Notification Text

Fig. 1. The study environment.

process as they complete a task or activity [27]. 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 study artifacts for data analysis.

Following the briefing period, participants progressed through encounters with four vulnerabilities. Figure 1 depicts the configuration of the integrated development environment (IDE) for one of these encounters. All participants consented to participate in our study, which had institutional review board approval, and to have their session recorded using screen and audio capture software. Finally, each session concluded with several demographic and open-ended discussion questions.

4.2.1 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 statute [39]. Participants were equipped with FSB, an extended version of FindBugs.

We chose FSB because it detects security defects and compares to other program analysis tools, such as those listed by NIST, [33] OWASP, [42] and WASC [44]. Some of the listed tools may include more or less advanced bug detection features. However, FSB is representative of static analysis security tools with respect to its user interface, specifically in how it communicates with its users. FSB provides visual code annotations and textual notifications that contain vulnerability-specific information. It summarizes all the vulnerabilities it detects in a project and allows users to prioritize potential vulnerabilities based on several metrics such as bug type or severity.

TABLE 1
Participant Demographics

Participant	Job Title	Vulnerability Familiarity	Experience Years
P1*	Student	●●◐○○	4.5
P2*	Test Engineer	●●●○○	8
P3	Development Tester	●●○○○	6
P4*	Software Developer	●●○○○	6
P5*	Student	●●●●○	10
P6	Student	●○○○○	4
P7	Software Developer	●●●●○	4.5
P8	Student	●●●○○	7
P9	Software Consultant	●●●○○	5
P10	Student	●●●○○	8

4.2.2 Participants

For our study, we recruited ten software developers, five students and five professionals. Table 1 gives additional demographic information on each of the ten participants. Asterisks denote previous use of security-oriented tools. Participants ranged in programming experience from 4 to 10 years, averaging 6.3 years. Participants also self-reported their familiarity with security vulnerabilities on a 5 point Likert scale, with a median of 3. Although we report on experiential and demographic information, the focus of this work is to identify questions that span experience levels. In the remainder of this paper, we will refer to participants by the abbreviations found in the participant column of the table.

We faced the potential confound of measuring participants questions about a new code base rather than measuring their questions about vulnerabilities. To mitigate this confound, we required participants to be familiar with iTrust; all participants either served as teaching assistants for, or completed a semester-long software engineering course that focused on developing iTrust. This requirement also ensured that participants had prior experience using static analysis tools. All participants had prior experience with FindBugs, the tool that FSB extends, which facilitated the introduction of FSB.

However, this requirement restricted the size of our potential participant population. Accordingly, we used a nonprobabilistic, purposive sampling approach [10], which typically yields fewer participants, but gives deeper insights into the observed phenomena. To identify eligible participants, we recruited via personal contacts, class rosters, and asked participants at the end of the study to recommend other qualified participants. Although our study involved only ten participants, we reached saturation [9] rather quickly; no new question categories were introduced after the fourth participant.

4.2.3 Tasks

First we conducted a preliminary pilot study ($n = 4$), in which participants spent approximately 10 to 15 minutes with each task and showed signs of fatigue after 60 minutes. To reduce the effects of fatigue, we asked each participant to assess four vulnerabilities. We do not report on data collected from this preliminary study.

When selecting tasks, we ran FSB on iTrust and identified 118 potential security vulnerabilities across three topics. To increase the diversity of responses, we selected tasks from mutually exclusive topics, as categorized by FSB. For the fourth task, we added a SQL injection vulnerability to iTrust by making minimal alterations to one of the database access objects. Our alterations preserved the functionality of the original code and were based on

examples of SQL injection found on OWASP [43] and in open-source projects. We chose to add a SQL injection vulnerability, because among all security vulnerabilities, OWASP ranks injection vulnerabilities as the most critical web application security risk.

For each task, participants were asked to assess code that "may contain security vulnerabilities" and "justify any proposed code changes." Table 2 summarizes each of the four tasks and the remainder of this section provides more detail about each task.

Task 1
The method associated with Task 1 opens a file, reads its contents, and executes the contents of the file as SQL queries against the database. Before opening the file, the method does not escape the filepath, potentially allowing arbitrary SQL files to be executed. However, the method is only ever executed as a utility from within the unit test framework. The mean completion time for this task was 14 minutes and 49 seconds.

Task 2
The method associated with Task 2 is used to generate random passwords when a new application user is created. FSB warns `Random` should not be used in secure contexts and instead suggests using `SecureRandom`. `SecureRandom` is a more secure alternative, but its use comes with a slight performance trade-off. The mean completion time for this task was 8 minutes and 52 seconds.

Task 3
The method associated with Task 3 reads several improperly validated string values from a form. These values are eventually redisplayed on the web page exposing the application to a cross site scripting attack. The mean completion time for this task was 13 minutes and 19 seconds.

Task 4
In the method associated with Task 4, a SQL Statement object is created using string interpolation, which is potentially vulnerable to SQL injection. FSB recommends using `PreparedStatements` instead. The mean completion time for this task was 8 minutes.

4.3 Data Analysis — Questions

To analyze the data, we first transcribed all the audio-video files using oTranscribe [41]. 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.

4.3.1 Question Criteria

Participants ask both explicit and implicit questions. Drawing from previous work on utterance interpretation [21], we developed five 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.**
  Ex: *Why aren't they using `PreparedStatements`?*
- **The participant makes a statement and explores the validity of that statement.**
  Ex: *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."**
  Ex: *I don't know that it's a problem yet.*
- **The participant clearly expresses uncertainty over a statement.**
  Ex: *Well, it's private to this object, right?*
- **The participant clearly expresses an information need by describing plans to acquire information.**
  Ex: *I would figure out where it is being called.*

4.3.2 Question Extraction

To make sure our extraction was exhaustive, the first two authors independently coded each transcript using the criteria outlined in the previous section. When we identified a statement that satisfied one or more of the above criteria, we marked the transcript, highlighted the participant's original statement, and clarified the question being asked. Question clarification typically entailed re-wording the question to best reflect the information the participant was trying to acquire. From the ten sessions, the first coder extracted 421 question statements; the other coder extracted 389.

It was sometimes difficult to determine what statements should be extracted as questions; the criteria helped ensure both authors 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.

4.3.3 Question Review

To remove duplicates and ensure the validity of all the questions, each transcript was reviewed jointly by the two authors who initially coded it. During this second pass, the two reviewers examined each question statement, discussing its justification based on the previously stated criteria. The two reviewers merged duplicate questions, favoring the wording that was most strongly grounded in the study artifacts. This process resulted in a total of 559 questions.

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%. The agreement scores suggest that the two reviewers consistently held similar interpretations of the question criteria.

It is also important to note that participants' questions related to several topics in addition to security. We discuss the questions that are most closely connected to security in Sections 5.2.1, 5.3.6, and 5.5.3. Although our primary focus is security, we are also interested in the other questions that participants posed, as those questions often have security implications. For example, researchers have observed that developers ask questions about data flow, like *What is the original source of this data*, even outside security [19]. However, in a security context, this question is particularly important, because potentially insecure data sources often require special handling to prevent attacks.

4.3.4 Question Sorting

To organize our questions and facilitate discussion, we performed an *open* card sort [11]. Card sorting is typically used to help struc-

TABLE 2
Four vulnerability exploration tasks

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

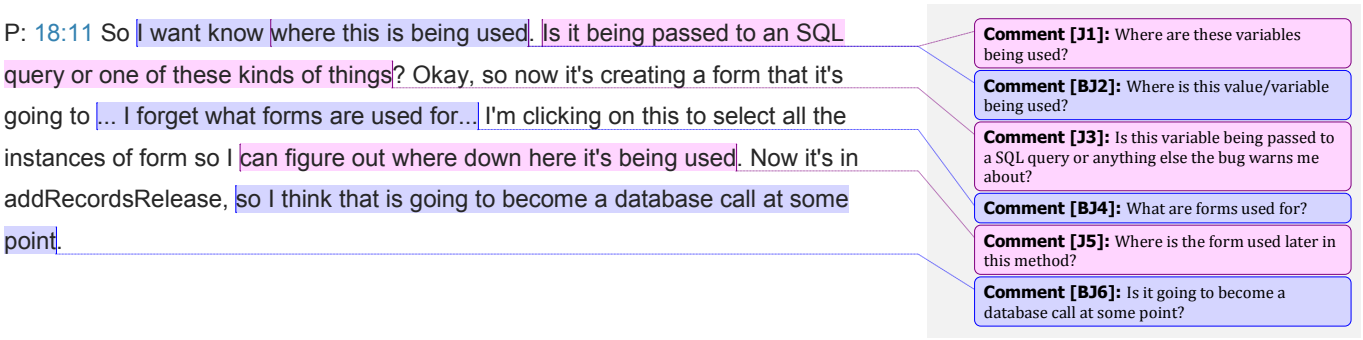


Fig. 2. Question merging process

ture data by grouping related information into categories. In an *open* sort, the sorting 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 needs. In this phase we focused on rephrasing similar questions 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* `PreparedStatements`*?* 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 five 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 3 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 researcher to sort. The first agreed with our categorization with a Cohen's Kappa of $\kappa = .63$. Between the first and second researcher we reworded and clarified some ambiguous questions. The second researcher exhibited greater agreement ($\kappa = .70$). These values are within the $.60 - .80$ range, indicating substantial agreement [17].

4.4 Data Analysis — Strategies

To answer RQ2 the first author performed two additional passes through the transcripts and recordings. In the first pass, we focused on identifying each participant's defect resolution strategy for each task. We define a defect resolution strategy as all the actions

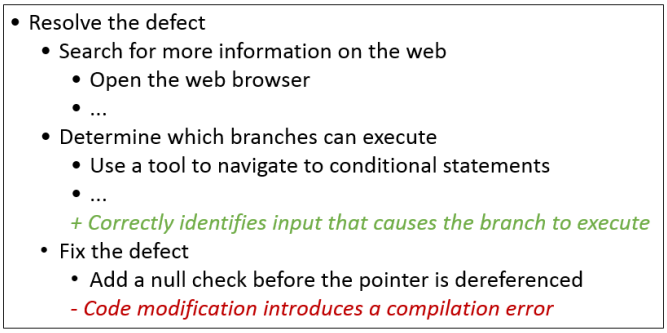


Fig. 3. Example strategy tree for a null dereference defect.

a developer takes to validate and resolve a defect. Such actions include invoking additional tools, modifying the source code, and searching the web, for example.

To represent these defect resolution strategies we drew from the notion of attack trees [24]. Attack trees encode the actions an attacker could take to exploit a system, including the idea that he may combine multiple actions to achieve a higher-level goal. Much like attack trees, our notation organizes actions hierarchically. We refer to this notation as a strategy tree. We use strategy trees to represent the hierarchical set of actions a developer takes to resolve a security defect. Figure 3 depicts an example strategy tree.

Given screen recordings and participant's think-aloud verbalizations, were able to identify their actions without inference. To do so, whenever a participant performed an action we added that action to the strategy tree.

During the second pass we added annotations to each strategy tree. As depicted in Figure 3 (lines prefixed with $+/-$), we annotated the trees whenever an action lead to a success or a failure. To best understand which strategies contributed to success, we measured success/failure as granularly as possible. Rather than annotating the entire tree as successful or not, we annotated the sub-strategies that compose the tree. Participants could succeed

with some strategies while failing with others. For example, participants could succeed in locating relevant information, but fail to interpret it correctly. Note: not all sub-strategies were observably successful/failure-inducing.

4.5 Data Analysis — Assumptions

To answer RQ3, we identified participants' assumptions and analyzed each assumption to determine if it was correct. Because implicit assumptions are so pervasive [6], it would be intractable to enumerate all of them. Participants likely made many assumptions silently. For example, they may have implicitly assumed the Eclipse environment was configured correctly and all the relevant source files were accessible. Instead of speculating about all of these assumptions, we only consider the assumptions that are explicitly triggered within the transcripts.

To identify the explicit assumptions participants made, we searched the transcripts for keywords such as, *assume*, *assuming*, *presume*, *reckon*, *believe*, and *guess*. This search process returned some false-positives. For example, one participant referred to the interviewer's assumptions rather than making an assumption of his own asking, "Am I assumed to have some prior knowledge?" To filter only the assumptions participants made about the task or the code, first author inspected each search result. After determining the result was an assumption, we evaluated its correctness. For example, one participant's assumed an external library was implemented securely. We determined this assumption was incorrect by locating a vulnerability in the library using the common vulnerability and exposures database.[1]

5 Results

5.1 Interpreting the Results

In the next four sections, we discuss our study's results using the categories we described in the previous section. Due to their large number, we grouped the categories to organize and facilitate discussion about our findings. Table 3 provides an overview of these groupings.

For each category, we selected several questions to discuss. A full categorization of questions can be found online [36]. The numbers next to the category titles denote the number of participants that asked questions in that category and the total number of questions in that category — in parenthesis and brackets respectively. Similarly, the number in parenthesis next to each question marks the number of participants that asked that question.

The structure of most results categories consists of three parts: an overview of the category, several of the questions we selected, and a discussion of those questions. However, some sections do not contain any discussion either because participants' intentions in asking those questions were unclear, or participants asked the questions without following up or attempting to answer them at all.

When discussing the questions participants asked for each category, we will use phrases such as "*X* participants asked *Y*." Note that this work is exploratory and qualitative in nature. Though we present information about the number of participants who ask specific questions, the reader should not infer any quantitative generalizations.

1. cve.mitre.org

5.2 Vulnerabilities, Attacks, and Fixes

5.2.1 *Preventing & Understanding Attacks (10){11}*

Unlike other types of code defects that may cause code to function unexpectedly or incorrectly, security vulnerabilities expose the code to potential attacks. For example, the Servlet Parameter vulnerability (Table 2) introduced the possibility of SQL injection, path traversal, command injection, and cross-site scripting attacks.

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)

Participants sought information about the types of attacks that could occur 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 read the notification about the Path Traversal vulnerability and stated, "I guess I'm thinking about different types of attacks." Before reasoning about how a specific attack could be executed, he wanted to determine which attacks were relevant to the notification.

Participants also sought information about specific attacks from the notification, asking how particular attacks could exploit a given vulnerability. Participants hypothesized about specific attack vectors, how to execute those attacks, and how to prevent those attacks now and in the future. Seven participants, concerned about false positives, asked the question, *Is this a real vulnerability?* To answer that question, participants searched for hints that an attacker could successfully execute a given attack in a specific 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.

Strategy Analysis Results

Participants used various strategies to answer the question, *What are the possible attacks that could occur?* For some vulnerabilities, participants read FSB's vulnerability information. When FSB did not provide sufficient information, participants turned to the web, searching on sites like Google and StackOverflow.

These strategies for determining the set of vulnerabilities to consider were prone to two types of failures. First, because web search engines are not fully aware of a developer's programming context, they may return a superset of the relevant attacks. For example, searching for attacks that exploit unvalidated input vulnerabilities returns results about cross site scripting attacks, injection attacks, and buffer overflow attacks. However, due to Java's automatic array bounds checking, buffer overflow attacks are not feasible in the vast majority of Java programs. Considering buffer overflow attacks would distract developers from other more relevant attacks.

Secondly, by executing these strategies some participants erroneously considered only a subset of the possible attacks. This failure was especially evident for the Servlet Parameter vulnerability, where a cross site scripting attack was feasible, but a SQL injection attack was not. Some participants correctly determined the data was sanitized before reaching the database, dismissed SQL injection, and prematurely concluded the vulnerability was a

TABLE 3
Organizational Groups and Emergent Categories

Group	Category	Clusters	Location in Paper
Vulnerabilities, Attacks, and Fixes	Preventing and Understanding Potential Attacks	11	Section 5.2.1
	Understanding Alternative Fixes and Approaches	11	Section 5.2.2
	Assessing the Application of the Fix	9	Section 5.2.3
	Relationship Between Vulnerabilities	3	Section 5.2.4
Code and the Application	Locating Information	11	Section 5.3.1
	Control Flow and Call Information	13	Section 5.3.2
	Data Storage and Flow	11	Section 5.3.3
	Code Background and Functionality	17	Section 5.3.4
	Application Context/Usage	9	Section 5.3.5
	End-User Interaction	3	Section 5.3.6
Individuals	Developer Planning and Self-Reflection	14	Section 5.4.1
	Understanding Concepts	6	Section 5.4.2
	Confirming Expectations	1	Section 5.4.3
Problem Solving Support	Resources and Documentation	10	Section 5.5.1
	Understanding and Interacting with Tools	9	Section 5.5.2
	Vulnerability Severity and Rank	4	Section 5.5.3
	Notification Text	3	Section 5.5.4
	Uncategorized	10	

false positive. By failing to consider cross site scripting attacks, participants overlooked the program path that exposed a true attack.

5.2.2 *Understanding Alt. Fixes & Approaches (8){11}*

When resolving security vulnerabilities, participants explored alternative ways to achieve the same functionality more securely. For example, while evaluating the potential SQL Injection vulnerability, participants found resources that suggested using the `PreparedStatement` class instead of Java `Statement` class.

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)

Some notifications, including those for the SQL Injection and Predictable Random vulnerabilities, explicitly offered alternative fixes. In other cases, participants turned to a variety of sources, such as StackOverflow, official documentation, and personal blogs for alternative approaches.

Three participants specifically cited StackOverflow as a source for alternative approaches and fixes. P7 preferred StackOverflow as a resource, because it included real-world examples of broken code and elaborated on why the example was broken. Despite the useful information some participants found, often the candidate alternative did not readily provide meta-information about the process of applying it to the code. 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.

While attempting to assess the Servlet Parameter vulnerability, P8 decided to explore some resources on the web and came across a resource that appeared to be affiliated with OWASP [43]. Because he recognized OWASP as "the authority on security," he clicked the link and used it to make his final decision regarding the vulnerability. It seemed important to P8 that recommended approaches came from trustworthy sources.

Strategy Analysis Results

Participants strategies for acquiring information about alternative fixes centered around three information sources — FSB, the web, and other modules in the code. Even when FSB informed participants about a specific alternative, participants sought supplementary information from the other two external sources.

For example, during Task 4, FSB suggested using `PreparedStatements`. In search of information about `PreparedStatements'` syntax, participants navigated to the project's related code modules they suspected would contain `PreparedStatements`. Participants largely found examples from these related modules helpful, because they were easy to translate back to the original method.

For Task 2, participants similarly sought examples of `SecureRandom`, but were ultimately less successful. Unaware that the project contained examples of `SecureRandom`, in this case, participants turned to online resources. Although some participants understood the relevant online API documentation, others struggled to compare the online documentation to the original method.

5.2.3 *Assessing the Application of the Fix (9){9}*

Once participants had identified an approach for fixing a security vulnerability (Section 5.2.2), they asked questions about applying the fix to the code. For example, while considering the use of `SecureRandom` to resolve the Predictable Random vulnerability, participants questioned the applicability of the fix and the consequences of making the change. The questions in this category differ from those in ***Understanding Alternative Fixes and Approaches*** (Section 5.2.2). These questions focus on the process of applying and reasoning about a given fix, rather than identifying and understanding possible fixes.

Will the notification 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)

When searching for approaches to resolve vulnerabilities, participants gravitated toward fix suggestions provided by the notification. As noted above, the notifications associated with the Predictable Random vulnerability and the SQL Injection vulnerability both provided fix suggestions. All participants proposed solutions that involved applying one or both of these suggestions. Specifically, P2 commented that it would be nice if all the notifications contained fix suggestions.

However, unless prompted, none of the participants commented on the disadvantages of using fix suggestions. While exploring the Predictable Random vulnerability, many participants, including P1, P2, and P6, decided to use `SecureRandom` without considering any alternative solutions, even though the use of that suggested fix reduces performance. It seems that providing suggestions without discussing the associated trade-offs appeared to reduce participants' willingness to think broadly about other possible solutions.

Strategy Analysis Results

Although participants were not asked to make any modifications to the code, they did describe their strategies for applying and assessing fixes. Many participants looked for ways to apply fixes automatically. However, none of the FSB notifications included quick fixes. Still, some participants unsuccessfully attempted to apply quick fixes by clicking on various interface elements.

To assess whether the fix had been applied correctly, participants commonly described one particular heuristic. P1, for instance, would make the change and simply "see if the bug [icon] goes away." This strategy typically ensures the original vulnerability gets fixed, but fails when the fix introduces new defects.

5.2.4 Relationship Between Vulnerabilities (4){3}

Some participants asked questions about the connections between co-occurring vulnerabilities and whether similar vulnerabilities exist elsewhere in the code. For example, when participants reached the third and fourth vulnerabilities, they began speculating about the similarities between the vulnerabilities they inspected.

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

Strategy Analysis Results

!StrategyNotes

- Really not much follow up on this one...
- Only one occurrence. P3 for Task 3 incorrectly guessed all the nearby defects were the same (wrong because some did not need sanitation) Otherwise, nothing more general to talk about here.

5.3 Code and the Application

5.3.1 Locating Information (10){11}

Participants asked questions about locating information in their coding environments. In the process of investigating vulnerabilities, participants searched for information across multiple classes and files. Unlike Sections 5.3.2 and 5.3.3, questions in this category more generally refer to the process of locating information, not just about locating calling information or data flow information.

Where is this used in the code? (10)
Where are other similar pieces of code? (4)
Where is this method defined? (1)

All ten participants wanted to locate where defective code and tainted values were in the system. Most of these questions occurred in the context of assessing the Predictable Random vulnerability. Specifically, participants wondered where the potentially insecure random number generator was being used and whether it was employed generate sensitive data like passwords.

In other cases, while fixing one method, four participants wanted to find other methods that implemented similar functionality. They hypothesized that other code modules implemented the same functionality using more secure patterns. For example, while assessing the SQL Injection vulnerability, P2 and P5 both wanted to find other modules that created SQL statements. All participants completed this task manually by scrolling through the package explorer and searching for code using their knowledge of the application.

Strategy Analysis Results

For the majority of participants, scrolling through the source file was a key component of their strategies for locating information. Eclipse provides many tools designed to help users locate specific types of information. For example, OPEN DECLARATION locates method declarations, FIND REFERENCES locates references. More generally Eclipse provides a customizable SEARCH tool for locating other information. Despite the availability of such dedicated tools, many participants unsuccessfully scrolled through the package explorer and open files before using tools.

5.3.2 Control Flow and Call Information (10){13}

Participants sought information about the callers and callees of potentially vulnerable methods.

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)

Participants asked some of these questions while exploring the Path Traversal vulnerability. While exploring this vulnerability, many participants eventually hypothesized that all the calls originated from the same test class, therefore were not user-facing, and thus would not be called with tainted values. Three participants explicitly asked, *Are all calls coming from the same class?* In fact, in this case, participants' hypotheses were partially correct. Tracing up the call chains, the method containing the vulnerability was called from multiple classes, however those classes were all contained within a test package.

Even though all participants did not form this same hypothesis, all ten participants wanted call information for the Path Traversal Vulnerability, often asking the question, *Where is this method being called?* However, participants used various 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 calls. Other participants used the Eclipse's MARK OCCURRENCES tool (code highlighting in Figure 1), which, to a lesser extent, was error-prone for the same reason. Further, it only highlighted calls within the current file.

Participants additionally employed Eclipse's FIND tool, which found all occurrences of a method name, but there was no guarantee that strings returned referred to the same method. Also, it returned references that occurred in dead code or comments. Alternatively, Eclipse's FIND REFERENCES tool identified proper references to a single method. Eclipse's CALL HIERARCHY tool enabled users to locate calls and traverse the project's entire call structure. That said, it only identified explicit calls made from within the system. If the potentially vulnerable code was called from external frameworks, CALL HIERARCHY would not alert the user.

Strategy Analysis Results

!StrategyNotes

- Discussion of strategies already in this section
- Also comes up in the flow navigation section

5.3.3 Data Storage and Flow (10){11}

Participants often wanted to better understand data being collected and stored: where it originated and where it was going. For example, participants wanted to determine whether data was generated by the application or passed in by the user. Participants also wanted to know if the data touched sensitive resources like a database. Questions in this category focus on the application's data — how it is created, modified, or used — unlike the questions in Section 5.3.2 that revolve around call information, essentially the paths through which the data can travel.

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)

Participants asked questions about the data pipeline while assessing three of the four vulnerabilities, many of these questions arose while assessing the Path Traversal vulnerability.

While exploring this vulnerability, participants adapted tools such as the CALL HIERARCHY tool to also explore the program's data flow. As we discussed in ***Control Flow and Call Information***, the CALL HIERARCHY tool helped participants identify methods' callers and callees. Specifically, some participants used the CALL HIERARCHY tool to locate methods that were generating or modifying data. Once participants identified which methods were manipulating data, they manually searched within the method for the specific statements that could modify or create data. They relied on manual searching, because the tool they were using to navigate the program's flow, CALL HIERARCHY, did not provide information about which statements were modifying and creating data.

Strategy Analysis Results

!StrategyNotes

- Also discussed this a lot in program flow discussion
- Most relevant to task 3.
- Rather than trace individual pieces of data used an aggregation strategy.
- Input parameters were all put into the form. Validation was correct for some, but not others.
- P6 got tired out, didn't want to trace all 12 variables. Incorrectly concluded the validation was correct for all the parameters
- P7 struggled with this task, without a good strategy to see how the data was used he proposed adding validation code (even though it already existed)

5.3.4 Code Background and Functionality (9){17}

Participants asked questions concerning the background and the intended function of the code being analyzed. The questions in this category differ from those in Section 5.3.5 because they focus on the lower-level implementation details of the code.

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)
How much effort was put into this code? (1)

Participants were interested in what the code did as well as the history of the code. For example, P2 asked about the amount of effort put into the code to determine whether he trusted that the code was written securely. He explained, "People were rushed and crunched for time, so I'm not surprised to see an issue in the servlets." Knowing whether the code was thrown together haphazardly versus reviewed and edited carefully might help developers determine if searching for vulnerabilities will likely yield true positives.

Strategy Analysis Results

To answer questions in this category, participants relied primarily on their prior knowledge of the system. We observed relatively few instances where participants executed additional strategies to substantiate their existing knowledge. This strategy, or lack thereof, was susceptible to failures when participants overestimated their knowledge of the system. For example, while assessing the Path Traversal vulnerability, P2 incorrectly assumed that the methods in the DBBulder module were called on startup.

5.3.5 Application Context and Usage (9){9}

Unlike questions in Section 5.3.4, these questions refer to system-level concepts. For instance, often while assessing the vulnerabilities, participants wanted to know what the code was being used for, whether it be testing, creating appointments with patients, or generating passwords.

What is the context of this vulnerability/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 ever being used? (2)

Participants tried to determine if the code in the Potential Path Traversal vulnerability was 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, participants sometimes used tools for traversing the call hierarchy; using these types of tools allowed them to narrow their search to only locations where the code of interest was being called.

Strategy Analysis Results

Participants were particularly interested in differentiating between two contexts — test code and application code. As P4 explained, test code does not ship with the product, so it is held to a different standard from a security perspective. To glean the context of a given method (e.g. whether it was a test method), participants looked for certain keyword indicators, such as "test" or "test case". Additionally, for tests, participants scrolled through the package explorer in search of test packages. Unfortunately, sometimes these heuristic strategies mislead participants. For instance, P9 incorrectly inferred the code for Task 3 was contained within a test class.

5.3.6 End-User Interaction (8){3}

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 were 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.

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

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 was being validated in the event that the input did come from the user. For these participants, finding the answer required manual inspection of surrounding and relevant code. For instance, P6 found a `Validator` method, which he manually inspected, to determine if it was doing input validation. He incorrectly concluded that the `Validator` method adequately validated the data.

When assessing the Potential Path Traversal vulnerability, four participants asked whether end-user input reached the code being analyzed. P2 used CALL HIERARCHY to answer this question by tracking where the method the code is contained in gets called; for him, the vulnerability was a true positive if user input reached the code. P1 and P6 searched similarly and determined that because all the calls to the code of interest appeared to happen in methods called `testDataGenerator()`, the code was not vulnerable.

Strategy Analysis Results

Participants' strategies for answering questions about end user interaction typically included strategies from other categories. In other words, participants gathered evidence by Locating Information (Section 5.3.1), examining Control Flow (Section 5.3.2), retracing Data Flow (Section 5.3.3) information, and recalling information about the Code Background (Section 5.3.4). Participants successfully answered questions in this category when they successfully synthesized the results of these other efforts.

5.4 Individuals

5.4.1 Developer Planning and Self-Reflection (8){14}

This category contains questions that participants asked about themselves. The questions in this category involve the participants' relationship to the problem, rather than specifics of the code or the vulnerability notification.

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)
Have I seen this before? (1)
Where am I in the code? (1)

Participants most frequently asked if they understood the situation, whether it be the code, the notification, or a piece of documentation. For instance, as P6 started exploring the validity of the SQL Injection vulnerability, he wanted to know if he fully understood the notification before he started exploring, so he went back to reread the notification before investigating further. These questions occurred in all four vulnerabilities.

Strategy Analysis Results

!StrategyNotes

- Pretty sparse in the strategy analysis.
- Resolved internally. Though we asked participants to vocalize their thoughts might have been difficult to get at this
- closest strategy we have is for why do I care P2, V1. Difficult interpreting the severity scale

5.4.2 Understanding Concepts (7){6}

Some participants encountered unfamiliar terms and concepts in the code and vulnerability notifications. 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.

What is this concept? (6)
How does this concept work? (4)
What is the term for this concept? (2)

Participants often clicked links leading to more information about specific concepts. For example, while assessing the Potential Path Traversal vulnerability, P2, unsure of what path traversal was, clicked the link labeled "path traversal attack" provided by FSB to get more information. If a link was not available, they went to the web to get more information or noted that the notification could have included more information on those concepts. While parsing the information provided for the Predictable Random vulnerability, P7 and P8 did not know what CSRF token was. The notification for this vulnerability did not include links so they searched the web for more information. When asked what information he would like to see added to the notification for the Servlet Parameter vulnerability, which also did not include any links, P4 noted he would have liked the notification to include what a servlet is and how it related to client control.

Strategy Analysis Results

As mentioned above, participants' strategies for finding information about concepts included clicking the links provided by FSB. Some participants hesitated to click the links provided by FSB, because they were not clear on what concepts the links pertained to. Instead, these participants preferred to search the web for information about concepts.

We observed participants using several noteworthy sub-strategies while searching online. To find information pertaining to their particular situation, several participants copied and pasted content directly from the FSB notification into their web browser. Participants also iteratively refined their search terms. For example, after performing a search that failed to return relevant results, P1 added specificity to his search by adding the term 'Java'. Finally, we noticed some consistency in the strategies participants were using to find information about concepts — several different participants used the exact same search terms.

5.4.3 Confirming Expectations (4){1}

A few participants wanted to be able to confirm whether the code accomplishes what they expected. The question asked in this category was, *Is this doing what I expect it to?*

Strategy Analysis Results

!StrategyNotes

- Ensure validation forms are correct. (P6 again). Didn't actually confirm the correctness...
- Not much here... 6 occurrences in strategy analysis

5.5 Problem Solving Support

5.5.1 Resources and Documentation (10){10}

Many participants indicated they would use external resources and documentation to gain new perspectives on vulnerabilities. For example, while assessing the Potential Path Traversal vulnerability, participants wanted to know what their team members would do or if they could provide any additional information about the vulnerability.

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 prevent or resolve this? (5)
Is this a reliable/trusted resource? (3)
What type of information does this resource link me to? (2)

All ten participants had questions regarding the resources and documentation available to help them assess a given vulnerability. Even with the links to external resources provided by two of the notifications, participants still had questions about available resources. Some participants used the links provided by FSB to get more information about the vulnerability. Participants who did not click the links in the notification had a few reasons for not doing so. For some participants, the hyperlinked text was not descriptive enough for them to know what information the link was offering; others did not know if they could trust the information they found.

Some participants clicked FSB's links expecting one type of information, but finding another. For example, P2 clicked the first link, labeled "WASC: Path Traversal," while trying to understand the Potential Path Traversal vulnerability hoping to find information on how to resolve the vulnerability. When he did not see that information, he attempted another web search for the same information. A few participants did not know the links existed, so they typically used other strategies, such as searching the web.

Other participants expressed interest in consulting their team members. For example, when P10 had difficulty with the Potential Path Traversal vulnerability, he stated that he would normally ask his team members to explain how the code worked. Presumably, the code's author could explain how the code was working, enabling the developer to proceed with fixing the vulnerability.

Strategy Analysis Results

Participants' primary strategy involved consulting the code (i.e. variable and class names) along with its sparse comments. Participants seemed to successfully use this information to understand the syntax of the application code (e.g. which variables were being stored in which data structures). However, these strategies sometimes failed to convey security-relevant semantic information. For example, as we discuss more in the surrounding sections, participants struggled to determine whether variables containing user input had been sanitized (5.3.6). Further, this strategy fell short when participants sought documentation for IDE tools or external APIs, in which case the information was not readily available within the code itself.

Supplementing their primary strategy, participants gathered additional information from web resources. Additionally, participants described their plans to contact team members for help, although they could not do so within the confines of the study.

5.5.2 Understanding and Interacting with Tools (8){9}

Throughout the study participants interacted with a variety of tools including FSB, CALL HIERARCHY, and FIND REFERENCES. While interacting with these tools, participants asked questions about how to access specific tools, how to use the tools, and how to interpret their output.

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 tool do I need for this? (1)
How can I annotate that these strings have been escaped and the tool should ignore the warning? (1)

Participants asked questions about accessing the tools needed to complete a certain task. Participants sometimes sought information from a tool, but could not determine how to invoke the tool or possibly did not know which tool to use. The question, *What tool do I need for this?* points to a common blocker for both novice and experienced developers, a lack of awareness [26].

Strategy Analysis Results

We observed two types of strategies in this category, tool selection strategies and tool evaluation strategies. While selecting tools, participants often knew what functionality they wanted to achieve, but were unsure how to find a tool to achieve that functionality. For example, P2 knew he wanted to navigate to methods up the call hierarchy, but struggled to identify an appropriate tool. His tool selection strategy involved first Ctrl-hovering over the current method's name followed by right clicking the method name. This

strategy failed because the tool was not available in the Ctrl-hover menu and he failed to recognize the tool in the right-click menu.

Rather than search for the ideal tool, some participants opted to opportunistically invoke tools and evaluate the appropriateness of their output. Unfortunately, these strategies were susceptible to misinterpretations. For example, one participant opened, closed, and reopened the CALL HIERARCHY tool several times, unable to determine whether it was appropriate.

5.5.3 Vulnerability Severity and Ranking (5){4}

FSB estimates the severity of each vulnerability it encounters and reports those rankings to its users (Table 2). Participants asked questions while interpreting these rankings.

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

Most of these questions came from participants wanting to know more about the tool's method of ranking the vulnerabilities in the code. For example, after completing the first task (Potential Path Traversal), P1 discovered the tool's rank, severity, and confidence reports. He noted how helpful the rankings seemed and included them in his assessment process for the following vulnerabilities. As he began working through the final vulnerability (SQL Injection), he admitted that he did not understand the tool's metrics as well as he thought. He wasn't sure what whether the rank (15) was high or low and if yellow was a "good" or "bad" color. Some participants, like P6, did not notice any of the rankings until after completing all four sessions when the investigator asked about the tool's rankings.

Strategy Analysis Results

!StrategyNotes

- Tried to understand severity rankings, but didn't place much weight on them for determining bugs actual severity. (loose connection)
- Not much else for strategies (only 4 occurrences)

5.5.4 Notification Text (6){3}

FSB provided long and short descriptions of each vulnerability (Figure 1). Participants read and contemplated these notifications to guide their analysis.

What does the notification text say? (5)
What is the relationship between the notification text and the code? (2)
What code caused this notification to appear (2)

Beyond asking about the content of the notification, participants also asked questions about how to relate information contained in the notification back to the code. For example, the Predictable Random vulnerability notes that a predicable random value could lead to an attack when being used in a secure context. Many participants attempted to relate this piece of information back to the code by looking to see if anything about the code that suggested it is in a secure context. In this situation, the method containing the vulnerability was named `randomPassword()`, which suggested to participants that the code was in a secure context and therefore a vulnerability that should be resolved.

Strategy Analysis Results

Participants defect resolution strategies appeared to include reading portions of the notification text. Some participants read the entirety of the message before proceeding, whereas others read just the short descriptions. Several participants only read the visible parts of the message, neglecting to scroll further to reveal the rest of the message.

More interestingly, participants' strategies seemed to depend on their experience. The three participants (P3, P4, and P6) who reported the lowest familiarity with vulnerabilities started 11 of 12 tasks by immediately reading the notification text. Conversely, the participants (P5 and P7) who reported the most familiarity with vulnerabilities only started reading the notification text for 2 of 8 tasks.

6 Assumptions

In this section we discuss the results of the assumption analysis...

7 Discussion

In this section we discuss two main implications of our work.

7.1 Flow Navigation

When iTrust performed security-sensitive operations, participants wanted to determine if data originated from a malicious source by tracing program flow. Similarly, given data from the user, participants were interested in determining how it was used in the application and whether it was sanitized before being passed to a sensitive operation. Questions related to these tasks appear in four different categories (Sections 5.3.1, 5.3.2, 5.3.3, 5.3.6). We observed participants using three strategies to answer program flow questions, strategies that were useful, yet potentially error-prone.

First, when participants asked whether data comes from the user (*a user-facing source*), and thus cannot be trusted, or if untrusted data is being used in a sensitive operation, participants would navigate through chains of method invocations. When participants navigated through chains of method invocations, they were forced to choose between different tools, where each tool had specific advantages and disadvantages. Lightweight tools, such as FIND and MARK OCCURRENCES, could be easily invoked and the output easily interpreted, but they often required multiple invocations and sometimes returned partial or irrelevant information. For example, using MARK OCCURRENCES on a method declaration highlights all invocations of the method within the containing file, but it does not indicate invocations in other files. On the other hand, heavyweight tools, such as CALL HIERARCHY and FIND REFERENCES, return method invocations made from anywhere in the source code, but were slower and clumsier for participants. Moreover, even heavyweight tools do not return all invocations when looking for tainted data sources; for instance, CALL HIERARCHY does not indicate when methods are being called from outside the system by a framework.

Second, when participants asked whether a data source was user-facing, participants would make inferences based on class names. For instance, any class that started with `Test` participants assumed was as JUnit test case, and thus was not user-facing, and therefore not a potential source of tainted data. When participants made inferences based on class names, their inferences were

generally correct that the class name accurately described its role. However, this strategy fails in situations where the word "Test" is overloaded; this happens in iTrust where "Test" can also refer to a medical laboratory test.

Third, a common strategy for participants was to rely on their existing knowledge of sensitive operations and data sources in the application. When participants relied on existing knowledge of sensitive operations and data sources, such reliance may be failure-prone whenever the code has been changed without their knowledge. Indeed, prior research suggests that developers are less knowledgeable about unstable code [8]. Additionally, when a developer only contributes to a portion of the system, as is often the case in the open source community [25], he may be unable to reason about the system-wide implications of a change.

Much like work that examines more general programming tasks [19], we observed that participants would have benefited from better program flow navigation tools while investigating security vulnerabilities. Although researchers have proposed enhanced tools to visualize call graphs [20] and trace control flow to its origin [3], in a security context, these tools share the same limitations as the existing heavyweight tools. Existing tools like CodeSonar [34] and Coverity provide source-to-sink notifications for analyzing security vulnerabilities, but take control away from the programmer by forcing the developer into a tool-dictated workflow.

We envision a new tool that helps developers reason about control flow and data flow simultaneously, by combining the strengths of existing heavy and lightweight tools. We imagine such a tool could use existing heavyweight program analysis techniques, but still use a lightweight user interface. For example, such a tool might use a full-program, call hierarchy analysis technique in the back end, but use a MARK OCCURRENCES-like user interface on the front end. To indicate calls from outside the current class, additional lightweight notifications would be needed. Such a tool could support both lightweight and systematic investigation of the flow of potentially tainted data.

7.2 Structured Vulnerability Notifications

FSB provided explanatory notifications of potential vulnerabilities. However, to completely resolve vulnerabilities, participants performed many cognitively demanding tasks beyond simply locating the vulnerability and reading the notification, as is evidenced by the breadth of questions they asked. To resolve potential vulnerabilities, we observed participants deploying a mix of several high-level strategies including: inspecting the code; navigating to other relevant areas of the code; comparing the vulnerability to previous vulnerabilities; consulting documentation and other resources; weighing existing knowledge against information in the notification; and reasoning about the feasibility of all the possible attacks. Yet, these strategies were limited in three respects.

Participants used error-prone strategies even when more reliable tools and strategies were available. For example, in Section 5.5.1, we noted that participants, unaware of the relevant hyperlinks embedded within the notification text, searched for links to external resources using web search tools. The web searches often returned irrelevant results. However, when the interviewer pointed out the embedded links after the session, participants stated that they probably should have clicked them.

Second, even after choosing an effective strategy, participants were often unaware of which tools to use to execute the strategy. For example, while assessing the Servlet Parameter vulnerability, participants wanted to determine whether certain input parameters were ever validated, but were not aware of any tools to assist in this process. Previous research suggests that both novice and experienced developers face problems of tool awareness [26].

Third, regardless of the strategies and tools participants used, they had to manually track their progress on each task. For example, the Servlet Parameter vulnerability involved twelve tainted parameters and introduced the possibility of several types of attacks. Participants had to reason about each of those attacks individually and remember which attacks they had ruled out. In a more general programming context, researchers have warned about the risks of burdening developers' memories with too many concurrent tasks — overburdening developers' attentive memories can result in *concentration failure* and *limit failure* [28].

We envision an approach that addresses these limitations by explicating developers' strategies in the form of hierarchically structured checklists. Previous research suggests that checklists can effectively guide developers [29]. We propose a structure that contains hierarchical, customizable tasks for each type of notification. For example, the structure would contain high-level tasks, such as "Determine which attacks are feasible," and subsequently more actionable nested subtasks, such as "Determine if a SQL injection attack is feasible" or "Determine if an XSS attack is feasible." This structure would also include a checklist-like feature that allows users to save the state of their interaction with a particular notification — for example, checking off which attack vectors they have already ruled out — diminishing the risk of *concentration failure* and *limit failure*. Additionally, each task could include links to resources that relate specifically to that task and tool suggestions that could help developers complete the task.

7.3 Contextual Search?

8 Threats to Validity

In this section we discuss the internal and external threats to the validity of our study.

We faced the internal threat of recording questions that participants asked because they lacked a basic familiarity with the study environment. We took two steps to mitigate this threat. First, we required participants to have experience working on iTrust. Second, at the beginning of each session, we briefed participants on FSB and the development environment. During these briefing sessions, we gave participants the opportunity to ask questions about the environment and study setup, though we cannot say for certain that participants asked all the questions they had at that time.

Thus, some of the questions we identified may reflect participants' initial unfamiliarity with the study environment and FSB. Since we are interested broadly in developers' information needs, the initial questions they ask about a new tool and environment still are an important subset to capture.

Because this study was conducted in a controlled environment rather than an industrial development setting, our results also face a threat to their external validity. Though we cannot and do not claim that we have identified a comprehensive categorization of all security-related questions all developers might ask, we have made several efforts to mitigate this threat. First, we included both students and professionals in our sample, because developers with more or less experience might ask different questions. Further, participants were equipped with FSB, a representative open source

static analysis tool with respect to its user interface. Finally, we chose iTrust, an industrial-scale open-source project as our subject application.

Relatedly, participants may have spent an unrealistic amount of time (either too much or too little) on each task due to working outside their normal work environment. To counteract this threat, we did not restrict the amount of time alloted for each task. Further, whenever a participant asked the interviewer what to do next, the interviewer provided minimal guidance, typically prompting the participant to proceed as she would in her normal work environment.

9 Conclusion

This paper reported on a study conducted to discover developers' information needs while assessing security vulnerabilities in software code. During the study, we asked ten software developers to describe their thoughts as they assessed potential security vulnerabilities in iTrust, a security-critical web application. We presented the results of our study as a categorization of questions. Our findings have several implications for the design of static analysis tools. Our results suggest that tools should help developers, among other things, navigate program flow to sources of tainted data.

10 Acknowledgments

We would like to thank our study participants. Special thanks to Xi Ge, Anthony Elliott, Emma Laperruque, and the Developer Liberation Front[2] for their assistance. This material is based upon work supported by the National Science Foundation under grant numbers 1318323 and DGE–0946818.

References

[1] N. Ammar and M. Abi-Antoun. Empirical evaluation of diagrams of the run-time structure for coding tasks. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 367–376. IEEE, 2012.

[2] 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*, pages 97–106. IEEE, 2011.

[3] M. Barnett, R. DeLine, A. Lal, and S. Qadeer. Get me here: Using verification tools to answer developer questions. Technical Report MSR-TR-2014-10, February 2014.

[4] 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*, pages 235–244. ACM, 2002.

[5] 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*, pages 1–6. IEEE, 2013.

[6] N. Fairclough. *Analysing discourse: Textual analysis for social research*. Psychology Press, 2003.

[7] 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*, pages 175–184. ACM, 2010.

[8] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Trans. Softw. Eng. Methodol.*, 23(2):14:1–14:42, Apr. 2014.

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

[10] G. Guest, A. Bunce, and L. Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

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

2. research.csc.ncsu.edu/dlf/

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

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

[14] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353. IEEE Computer Society, 2007.

[15] 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*, pages 151–158. ACM, 2004.

[16] 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*, pages 127–134. IEEE, 2012.

[17] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):pp. 159–174, 1977.

[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*, pages 185–194. ACM, 2010.

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

[20] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 117–124. IEEE, 2011.

[21] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

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

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

[24] S. Mauw and M. Oostdijk. Foundations of attack trees. In D. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer Berlin Heidelberg, 2006.

[25] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[26] E. Murphy-Hill, R. Jiresal, and G. C. Murphy. Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 42:1–42:11, New York, NY, USA, 2012. ACM.

[27] J. Nielsen, T. Clemmensen, and C. Yssing. Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 101–110. ACM, 2002.

[28] C. Parnin and S. Rugaber. Programmer information needs after memory failure. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 123–132. IEEE, 2012.

[29] K. Y. Phang, J. S. Foster, M. Hicks, and V. Sazawal. Triaging checklists: a substitute for a phd in static analysis. *Evaluation and Usability of Programming Languages and Tools (PLATEAU) PLATEAU 2009*, 2009.

[30] 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*, page 43. ACM, 2012.

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

[32] 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*, pages 119–126. IEEE, 2013.

[33] Nist source code security analyzers. http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html.

[34] Codesonar. http://grammatech.com/codesonar.

[35] Coverity. http://coverity.com/.

[36] Security questions experimental materials. http://www4.ncsu.edu/~bijohnso/security-questions.html.

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

[38] Find security bugs. http://h3xstream.github.io/find-sec-bugs/.
[39] Hippa statute. http://hhs.gov/ocr/privacy/.
[40] itrust software system. http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start.
[41] Otranscribe. http://otranscribe.com.
[42] Owasp source code analysis tools. http://owasp.org/index.php/Source_Code_Analysis_Tools.
[43] Owasp. http://owasp.org/index.php/Main_Page.
[44] Web application security consortium static code analysis tools. http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList.