
Software developers play a critical role in securing software applications. Static analysis tools assist developers by detecting vulnerabilities early. The goal of this work is to examine how software developers use static analysis tools to resolve security vulnerabilities so that we can design more effective tools. To accomplish this goal, first we observe developers as they resolve security vulnerabilities—we identify both their information needs and current strategies for satisfying those needs. To gain additional perspective on how practices vary by experience, we also study how software developers who specialize in security use static analysis to contribute to the resolution process. Finally, we examine existing tools to understand how they support developers’ strategies.
ACKNOWLEDGEMENTS

There are so many people who I would like to thank for contributing to this work in countless ways. My most sincere apologies to anyone I overlooked.

My co-advisor, Emerson Murphy-Hill, who introduced me to software engineering and supported me throughout my entire graduate career. Thank you for believing in me and uplifting me with your vision, encouragement, infusions of good ideas, humor, and patience.

My co-advisor, Kathryn Stolee, for her very generous guidance and support, especially during the final stages of my PhD.

My committee members, Sarah Heckman, William Enck, Andrew J. Ko, and Adam Meade, for all your time, insight, and advice.

My research colleagues, Brittany Johnson, Titus Barik, and Denae Ford Robinson for the hard work they have contributed to my research. More importantly, for allowing me to lean on them as dear friends.

My collaborators Lisa Nguyen Quang Do and Chris Brown, who are both incredibly bright and a pleasure to work with. Lisa made significant contributions to my heuristic walkthrough study and Chris helped build the Flower tool.

My collaborators and mentors in industry: Nicholas Kraft at ABB; Ben Zorn, Christian Bird, Thomas Zimmermann at Microsoft Research; and Carson Potter at IBM.

My friends and colleagues at NCSU and beyond: Joymallya Chakraborty, Sarah Elder, Austin Henley, Nasif Imtiaz, Andrew McNamara, Justin Middleton, Pat Morrison, Rahul Pandita, Nischal Shrestha, Tyler Thomas, denizens of the 3228 lab, to name a few.

The participants in all my studies, whose time and cooperation truly enabled all of this research.
Above all, to my wife, Emma Laperruque, who inspires me. You’ve helped me persevere when I felt lost. This dissertation would not have been possible without your love and support.
# TABLE OF CONTENTS

LIST OF TABLES .................................................................................................................. v

LIST OF FIGURES ................................................................................................................ vi

Chapter 1 Introduction ......................................................................................................... 1
  1.1 Thesis ............................................................................................................................. 3
  1.2 Research Questions ...................................................................................................... 3

Chapter 2 Developers’ Information Needs and Strategies .................................................. 5
  2.1 Introduction .................................................................................................................... 5
  2.2 Methodology ................................................................................................................ 7
    2.2.1 Study Design ........................................................................................................ 7
  2.3 Data Analysis — Questions .......................................................................................... 15
    2.3.1 Question Criteria .................................................................................................. 16
    2.3.2 Question Extraction ............................................................................................. 17
    2.3.3 Question Review .................................................................................................. 18
    2.3.4 Question Sorting ................................................................................................. 19
  2.4 Data Analysis — Strategies .......................................................................................... 20
    2.4.1 Strategy Definitions .............................................................................................. 20
    2.4.2 Strategy Extraction ............................................................................................... 21
    2.4.3 Strategy Review .................................................................................................... 23
  2.5 Data Analysis — Assumptions ....................................................................................... 24
  2.6 Results ........................................................................................................................... 27
    2.6.1 Vulnerabilities, Attacks, and Fixes ....................................................................... 28
    2.6.2 Code and the Application ..................................................................................... 35
    2.6.3 Individuals ............................................................................................................ 47
    2.6.4 Problem Solving Support ...................................................................................... 49
  2.7 Discussion ...................................................................................................................... 57
    2.7.1 Flow Navigation .................................................................................................... 57
    2.7.2 Structured Vulnerability Notifications ................................................................ 60
    2.7.3 Context-Informed Web Search ............................................................................. 62
  2.8 Threats to Validity ......................................................................................................... 65
  2.9 Conclusion ..................................................................................................................... 67

Chapter 3 Security Specialists on Red Teams ..................................................................... 68
  3.1 Introduction ................................................................................................................... 68
  3.2 Methodology ................................................................................................................ 69
  3.3 Findings ........................................................................................................................ 72
    3.3.1 Workflow .............................................................................................................. 74
3.3.2 Security Culture .......................................................... 79
3.3.3 Software Engineering vs. Security Software Engineering .......... 83
3.4 Threats to Validity ............................................................ 86
3.5 Discussion ........................................................................ 88
  3.5.1 Organizationally Supporting Red Teams .......................... 88
  3.5.2 Designing Tools for Software Security Engineering ............. 89
  3.5.3 How Practitioners Perceive Security Research ................... 91
  3.5.4 Hacking Tools as Offensive Weapons and Defensive Tools ....... 91
  3.5.5 Recognizing the Impact of Software Security Engineering ....... 92
3.6 Conclusion ....................................................................... 93

Chapter 4 Evaluating the Usability of Static Analysis Tools for Security .... 95
  4.1 Introduction ...................................................................... 95
  4.2 Description of Selected Tools .............................................. 97
    4.2.1 Tool Selection Process ................................................. 97
    4.2.2 Find Security Bugs ..................................................... 98
    4.2.3 Flawfinder .................................................................. 103
    4.2.4 CTool ........................................................................ 105
  4.3 Methodology ..................................................................... 107
    4.3.1 Study Environment ..................................................... 107
    4.3.2 Heuristic Walkthroughs ............................................... 108
    4.3.3 Replication Package .................................................... 111
  4.4 Results ............................................................................ 111
    4.4.1 Missing Affordances ................................................... 112
    4.4.2 Missing or Buried Information ....................................... 114
    4.4.3 Scalability of Interface ................................................ 116
    4.4.4 Inaccuracy of Analysis ................................................ 119
    4.4.5 Code Disconnect ........................................................ 120
    4.4.6 Workflow Continuity ................................................... 122
  4.5 Limitations ...................................................................... 124
  4.6 Conclusion ..................................................................... 125

Chapter 5 Related Work .......................................................... 127
  5.1 Evaluating Security Tools .................................................. 127
  5.2 Improving the Usability of End User Security Tools ................... 130
  5.3 Usable Static Analysis Tools ............................................... 131
  5.4 Software Security Activities .............................................. 132

Chapter 6 Conclusion ................................................................ 135
  6.1 Implications ..................................................................... 137
  6.2 Recommendations for Toolsmiths ....................................... 138
6.3 Future Work .................................................. 138

BIBLIOGRAPHY .................................................. 141

APPENDICES ..................................................... 153

Appendix A  Study Materials from Information Needs .................. 154
A.1 Recruitment Email ........................................... 154
A.2 Procedure ..................................................... 155
A.3 Categorized Questions ....................................... 157
A.4 Assumptions ................................................... 164

Appendix B  Study Materials from Red Team Study .................... 169
B.1 Interview Script ............................................... 169
B.1.1 Introductions .............................................. 169
B.1.2 Background: ................................................ 170
B.1.3 Workflow: .................................................. 170
B.1.4 Collaboration .............................................. 171
B.1.5 Learning: .................................................... 172
B.1.6 Wrapups: ................................................... 172
B.2 Recruitment Email ............................................ 172

Appendix C  Study Materials from Heuristic Walkthroughs .......... 174
C.1 Heuristic Walkthrough Evaluation Guide .......................... 174
C.1.1 List of test suites .......................................... 174
C.1.2 Heuristic Walkthrough: Pass 1 ............................ 174
C.1.3 Heuristic Walkthrough: Pass 2 ............................ 176
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table 2.1</th>
<th>Participant Demographics</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.2</td>
<td>Four vulnerability exploration tasks</td>
<td>16</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Organizational Groups and Emergent Categories</td>
<td>26</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Informants</td>
<td>71</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Coverage of Informant Themes and Subcategories</td>
<td>73</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Evaluated tool interface dimensions</td>
<td>99</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Usability issues, grouped by theme</td>
<td>113</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1       Study environment, including: short vulnerability description in the
                  bug explorer (A); vulnerable code (B); long vulnerability description
                  (C). ................................................. 8
Figure 2.2       Question merging process ............................................. 16
Figure 2.3       Example strategy tree for a null dereference defect. ............... 22
Figure 2.4       Code view in the Eclipse IDE with Flower invoked during Task 1. The
                  image on the left shows the tool running in the method parseAndCache.
                  When the user clicks on the link in the editor (B1 or B2), they are taken
                  to the function getQueries, which is a caller of parseAndCache. 60
Figure 2.5       A mockup of a tool that presents successful strategies. ................. 62
Figure 2.6       Reference links provided by FSB for Path Traversal vulnerability ... 63
Figure 4.1       The Graphical User Interface of Find Security Bugs. .................. 100
Figure 4.2       The Find Security Bugs configuration settings. ......................... 101
Figure 4.3       RIPS’ exploit creator view. ............................................. 102
Figure 4.4       The Graphical User Interface of RIPS. .................................. 104
Figure 4.5       The Graphical User Interface of Flawfinder. ............................ 105
Figure 4.6       Instance of Overlapping Vulnerabilities ................................. 117
Figure 4.7       Scalability of RIPS’ function view. ..................................... 118
Figure 4.8       RIPS call graph visualization for more than 50 files. ................. 119
Figure 4.9       Duplicate entries in FSB’s navigation feature. ......................... 120
Figure 4.10      Instance of Mismatched Examples ........................................ 121
Static analysis tools, like Spotbugs [Spo], Checkmarx [Che], and Coverity [Cov], enable software developers to detect security vulnerabilities early in the development process. These tools locate and report on potential software security vulnerabilities, such as SQL injection and cross-site scripting, even before code executes. Detecting these defects early is important, because security vulnerabilities are most likely to cause incidents that affect end users [Che02] and long-lingering defects may be even more expensive to fix [Pre05]. According to a recent survey by Christakis and colleagues, developers seem to recognize the importance of detecting security vulnerabilities with static analysis; among several types of code quality issues, developers rank security issues as the highest priority for static analysis tools to detect [Chr16].

Unfortunately, evidence suggests existing tools are not easy for developers to use. Researchers cite several related reasons why these tools do not help developers resolve defects, for instance, the tools: “may not give enough information” [Joh13]; produce “bad warning messages” [Chr16]; and “miscommunicate” with developers [Joh16]. As a result, developers
struggle to resolve security vulnerabilities due to the poor usability of security tools [Gre16]. This is problematic, because, as Chess and McGraw argue, usability is essential to actually make software more secure: “Good static analysis tools must be easy to use, even for non-security people. This means that their results must be understandable to normal developers who might not know much about security and that they educate their users about good programming practice” [Che04].

Evaluating the efficacy of security-oriented static analysis tools has been a popular topic for researchers [Zit04; Aus13; Rea16; Ema08]. However, prior work has largely overlooked the usability of these tools, instead focusing on functional properties such as the types of vulnerabilities tools detect (or fail to detect), false alarm rates, and performance. Acar and colleagues draw attention to this current gap in the literature, explaining, “Usable security for developers has been a critically under-investigated area.” [Aca16]

This dissertation responds to the gap in the literature by investigating the usability problem plaguing security-oriented static analysis tools. In particular, we address this problem by studying developers’ information needs and strategies for acquiring the information they need while using security-oriented static analysis tools.

Several prior studies have explored developers’ information needs outside the context of security. For instance, prior work has examined developers’ information needs pertaining to: change tasks [Sil08]; social and collaborative aspects of development in software ecosystems [Fri10; Hae13]; and the day-to-day activities of teams [Ko07; LaT10b]. These prior studies have helped toolsmiths both evaluate the effectiveness of existing tools [Amm12], and improve the state of program analysis tools [Kon12; Ser12; Yoo13]. Similarly, we expect that building on this line of research will be an important first step toward realizing the benefits seen in other domains. We expect that categorizing developers’ information needs while using security-focused static analysis tools will help researchers evaluate and tool-
smiths improve static analysis tools for security. Understanding these information needs leads us to the thesis statement of this dissertation:

1.1 Thesis

To diagnose potential security vulnerabilities while using static analysis tools, developers must strategically synthesize four categories of information about: attacks, vulnerabilities, the software itself, and the social ecosystem that built it.

1.2 Research Questions

To support this thesis statement, we structure this dissertation around the following research questions:

**Information Needs (RQ1):** What information do developers need while using static analysis tools to diagnose potential security vulnerabilities?

In Chapter 2, we argue that static analysis tools should help developers acquire the information they need to resolve vulnerabilities. In order for tools to actually provide that information, we must first identify what information developers need. To do so, we conducted a study where we observed developers thinking aloud while diagnosing security vulnerabilities. Answering this research question, which we do in Chapter 2, supports the thesis statement because it reveals that developers draw upon “four categories of information” and enumerates those categories.

**Strategies (RQ2):** What are developers’ strategies for acquiring the information they need?
Prior work by Bhavnani and John shows that task success is not just determined by knowledge of information and tools; users also need to know good strategies [Bha00]. This suggests that the information needs we identify in RQ1 might not completely explain developers’ success or failure. Therefore, we performed two additional analyses of the data collected from the information needs study (RQ1) to identify developers’ strategies. Answering this research question, which we also do in Chapter 2, supports the thesis statement, because it reveals that developers “must strategically synthesize...information”.

**Specialization (RQ3):** How do specialized developers on security red teams use static analysis?

As we explain in Chapter 3, dedicated teams of security specialists are often responsible for security in organizations. We explore this research question to ensure our thesis statement covers those developers who are specialized in security, since they play an important role in securing software and their needs and strategies may differ from other developers. Answering this research question supports the thesis statement, because it helps to ensure the thesis applies to an important group “developers” who are often responsible for security within organizations.

**Usable Tools (RQ4):** How do existing static analysis tools support developers’ information needs and strategies?

In RQ1 and RQ2 we study developers using a single static analysis tool. To answer this research question we evaluate three additional tools. Answering this question helps to generalize our thesis beyond that single tool. By answering this question, we also demonstrate how the findings from previous questions can be used to identify usability issues in practice. Answering this research question, which we do in Chapter 4, contextualizes our findings from the first three RQs, moving from what information developers need in an ideal case, to understanding how existing tools address those needs.
CHAPTER TWO

DEVELOPERS’ INFORMATION NEEDS AND STRATEGIES

2.1 Introduction

This chapter describes a study in which we observed developers using a static analysis tool to resolve security vulnerabilities. The findings from this chapter provide a foundation for the rest of the dissertation and help us understand how software developers resolve security vulnerabilities with static analysis.

In this chapter, we answer RQ1 and RQ2. Examining RQ1 (What information do developers need while using static analysis tools to diagnose potential security vulnerabilities?) first, we identified developers’ information needs while using a static analysis tool to diagnose security vulnerabilities. To answer RQ2 (What are developers’ strategies for acquiring the information they need?), we explored how developers acquire the information they need through strategies. Rather than actively pursuing strategies, programmers also make simplifying assumptions to satisfy their information needs [Ko04]. Therefore, we also considered

_I’d like to thank Brittany Johnson for contributing to the study presented in this chapter._
the assumptions participants made to cover the cases when participants did not actively demonstrate a strategy.

To identify developers’ information needs and strategies, we conducted an exploratory think-aloud study with ten developers who had contributed to iTrust [Hec18], a security-critical medical records software system written in Java. We observed each developer as they assessed potential security vulnerabilities identified by a security-oriented static analysis tool, Find Security Bugs (FSB) [Fsb]. We operationalized developers’ information needs by measuring questions — the verbal manifestations of information needs. Using a card sort methodology, we sorted 559 questions into 17 categories of information needs. Further, we conducted additional analysis to identify developers’ strategies for acquiring the information they need. The contents of this chapter were first published at the Foundations of Software Engineering conference [Smi15] and were later extended in Transactions in Software Engineering [Smi18].

The contributions of this chapter are:

• A categorization of questions developers ask while resolving security defects.

• A catalog of developers’ defect resolution strategies, organized by the information need each strategy addresses. For example, to understand how to implement a fix, participants strategically surveyed multiple sources of information, including the web, the tool’s notification, and other modules in the code.

• A description of the common assumptions that undermined those strategies. For example, during some tasks, participants incorrectly assumed input validation had been handled securely.


2.2 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 2.2.1 details how the study was designed and Sections 2.3, 2.4, and 2.5 describe how we performed our three phases of data analysis. Study materials can be found online [Exp] and in Appendix A.

2.2.1 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 process as they complete a task or activity [Nie02]. 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.
Figure 2.1 Study environment, including: short vulnerability description in the bug explorer (A); vulnerable code (B); long vulnerability description (C).

The use of java.util.Random is predictable. For example, when the value is used as:

- a CSRF token: a predictable token can lead to a CSRF attack as an attacker will know the value of the token
- a password reset token (sent by email): a predictable password token can lead to an account takeover, since an attacker will guess the URL of the change password form
- any other secret value

A quick fix could be to replace the use of java.util.Random with something stronger, such as java.security.SecureRandom.
Following the briefing period, participants progressed through encounters with four vulnerabilities. Figure 2.1 depicts the configuration of the Eclipse for Java 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.

2.2.1.1 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 [Hip]. The code base comprises over 50,000 lines of code, including the test packages. 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, [Sec] OWASP, [Owab] and WASC [Codb]. 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.

2.2.1.2 Participants

For our study, we recruited ten software developers, five students and five professionals via our personal connections and class rosters. We recruited both students and professionals to diversify the sample; our analysis does not otherwise discriminate between these two
Table 2.1 Participant Demographics

<table>
<thead>
<tr>
<th>Participant</th>
<th>Job Title</th>
<th>Vulnerability Familiarity</th>
<th>Experience Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1*</td>
<td>Student</td>
<td>★★★★★★</td>
<td>4.5</td>
</tr>
<tr>
<td>P2*</td>
<td>Test Engineer</td>
<td>★★★★★★</td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>Development Tester</td>
<td>★★★★★★</td>
<td>6</td>
</tr>
<tr>
<td>P4*</td>
<td>Software Developer</td>
<td>★★★★★★</td>
<td>6</td>
</tr>
<tr>
<td>P5*</td>
<td>Student</td>
<td>★★★★★★</td>
<td>10</td>
</tr>
<tr>
<td>P6</td>
<td>Student</td>
<td>★★★★★★</td>
<td>4</td>
</tr>
<tr>
<td>P7</td>
<td>Software Developer</td>
<td>★★★★★★</td>
<td>4.5</td>
</tr>
<tr>
<td>P8</td>
<td>Student</td>
<td>★★★★★★</td>
<td>7</td>
</tr>
<tr>
<td>P9</td>
<td>Software Consultant</td>
<td>★★★★★★</td>
<td>5</td>
</tr>
<tr>
<td>P10</td>
<td>Student</td>
<td>★★★★★★</td>
<td>8</td>
</tr>
</tbody>
</table>

Groups. Table 2.1 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 chapter, 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 [Gue06], 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 [Gla09] rather quickly; no new question categories were introduced after the fourth participant.

2.2.1.3 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 about an hour. To reduce the effects of fatigue, we asked each participant to assess just four vulnerabilities. We do not report on data collected from this pilot study.

The tasks we chose encompass a subset of the vulnerability remediation activities in the wild. In a talk given at an RSA conference, Dan Cornell describes the activities involved in vulnerability remediation tasks in industry [Cor12]. He illustrates how these tasks compare to tasks outside the lab. By his account, vulnerability remediation includes activities such as planning, setting up development environments, performing functional testing, and deployment. In this chapter, we focus more narrowly on how developers diagnose vulnerabilities. Therefore, we do not ask participants to perform these auxiliary tasks. Later, we conducted interviews with professional security specialists to cover these broader aspects of vulnerability resolution (Chapter 3).

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 [Owaa] 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 2.2 summarizes each of the four tasks and the remainder of this section provides more detail about each task, including excerpts from the lines FSB initially flagged. All participants progressed through the tasks in a fixed order. The tasks were not otherwise related to each other, aside from the fact that they were all located in iTrust. Although we are not specifically studying task completion time, we report the mean completion time for each task.

**Task 1**
The method associated with Task 1, `parseSQLFile`, 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. Therefore, the method is only ever passed a predefined set of filepaths that cannot be maliciously manipulated.

To complete this task, participants needed to recognize that `parseSQLFile` was only used in tests and that the filepaths were essentially hard-coded, which could be accomplished by examining all of `parseSQLFile`’s call locations. The mean completion time for this task was 14 minutes and 49 seconds.
private List<String> parseSQLFile(String path) {
    FileReader r = new FileReader(new File(path));
    ...
}

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 (such as password generation) and instead suggests using SecureRandom, a more secure alternative. Using SecureRandom does impose a slight performance trade-off, however participants were not explicitly instructed that performance was a concern. Correct fixes for this vulnerability replace Random with SecureRandom, ensure the number generator is securely seeded, and appropriate API calls are used. The mean completion time for this task was 8 minutes and 52 seconds.

public class RandomPassword {
    
    private static final Random r = new Random();
    ...
}

Task 3
The method associated with Task 3 reads several improperly validated string values from a form. Entering an apostrophe (') into any of the fields modifies the underlying Java server page (JSP) and permits form manipulation. Additional modification of the form fields
can produce more unexpected behavior. The values entered into the form are eventually redisplayed on the web page exposing the application to a cross site scripting attack. Correct fixes for this task either modify the JSP to escape output or modify the validation methods to reject apostrophes. Either of these fixes require participants to navigate away from the file containing the FSB warning. The mean completion time for this task was 13 minutes and 19 seconds.

```java
protected void doPost(HttpServletRequest req, HttpServletResponse response)
{
    currentMID = req.getParameter("currentMID");
    recPhone = req.getParameter("recPhone");
    recEmail = req.getParameter("recEmail");
    ...
}
```

**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 `PreparedStatement` instead. In this case, although the inputs had likely already been sanitized elsewhere in the application, the cost of switching to `PreparedStatement` is negligible. Furthermore, the standard as implemented by other database access objects (DAOs) in iTrust is to use the more secure `PreparedStatement` class. Therefore, a correct fix would be to convert the method to use `PreparedStatement`. The mean completion time for this task was 8 minutes.
public void addApptRequest(ApptBean bean)
{
    Statement stmt;
    ...

    String query = String.format(
        "INSERT INTO appointmentrequests
        (appt_type, patient_id, doctor_id,
        sched_date, comment, pending, accepted)
        VALUES (%s, %d, %d, %s, %s, %d, %d);",
        bean.getApptType(),
        bean.getPatient(),
        bean.getHcp(),
        bean.getDate(),
        bean.getComment(),
        ...
        stmt.executeUpdate(query);
        ...
    }

2.3 Data Analysis — Questions

To analyze the data for RQ1, we first transcribed all the audio-video files using oTranscribe [Otr]. Each transcript, along with the associated recording, was analyzed by two researchers for questions. The two question sets for each session were then iteratively compared against each other until the researchers reached agreement on the question sets.
**Table 2.2** Four vulnerability exploration tasks

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

**Figure 2.2** Question merging process

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.

**2.3.1 Question Criteria**

Drawing from previous work on utterance interpretation [Let87], 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.

2.3.2 Question Extraction

To make sure our extraction was exhaustive, the two researchers 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 rewording 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 researchers only highlighted the statements that reflected actual questions. Figure 2.2 depicts a section of the questions extracted by both researchers from P8 prior to review.
2.3.3 Question Review

To remove duplicates and ensure the validity of all the questions, each transcript was reviewed jointly by the two researchers 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. Including questions that were only identified by one reviewer, this process resulted in a total of 559 questions.

Each question that was only identified by one researcher required verification. If the other researcher 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 2.6.1.1, 2.6.2.6, and 2.6.4.3. Although our primary focus is security, we are also interested in the other questions that participants posed, as those questions often have security implications and require special considerations in the context of security. For example, researchers have observed that developers ask questions about data flow, like, *What is the original source of this data*, even outside security [LaT10b]. A developer concerned with security might need to consider that attackers are one potential source of data. Data originating from an attacker could be crafted to disguise its malicious nature or to do more damage to the system. In such a context, the implications are somewhat unique.
to security, because, for example, potentially insecure data sources often require special handling to prevent attacks.

### 2.3.4 Question Sorting

To organize our questions and facilitate discussion, we performed an *open* card sort [Hud13]. Card sorting is typically used to help structure 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 154 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 2.3 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 [Lan77].

2.4 Data Analysis — Strategies

In this section, we describe our methodology for answering RQ2, “What are developers’ strategies for acquiring the information they need?.”

2.4.1 Strategy Definitions

We define a defect resolution strategy as all the actions a developer takes to validate and resolve a defect. Such actions include invoking tools, modifying the source code, and searching the web, among many other things. We adapt Bhavani and John's related definition of a strategy, “a method of task decomposition that is non-obligatory and goal directed,” to determine which actions to include in a strategy [Bha97].

Consider the following example which illustrates this notion of non-obligatory actions and how we use it as a stopping criteria for our strategy analysis. Say a developer’s goal is to get more information about the code. One strategy would be to read the code itself, another would be to read the documentation. These are both considered strategies because they help the developer achieve a goal, and the developer is not obligated to choose either one of these approaches. Now consider a negative example where a developer’s goal is to read the code. We wouldn’t consider the developer’s act of fixating his gaze on the screen a
strategy, because it is obligatory (assuming the developer does not also use a screen reader or other similar technology).

There are many ways to represent strategies. Our screen and audio recordings represent strategies with high fidelity because they contain everything that happened during the study. However, this format is difficult to work with and does not summarize the essence of a strategy. Another format we considered was to synthesize each participant’s strategies into unstructured textual summaries. This format has the advantage over raw recordings of being easy to search and consume, but it does not represent the relationships between strategies.

Ultimately, we created a new notation for representing strategies, called strategy trees. Strategy trees are based on the idea of attack trees [Mau06]. Attack trees encode the actions an attacker could take to exploit a system, including the idea that the attacker may combine multiple actions to achieve a higher-level goal. Much like attack trees, our representation organizes actions hierarchically. Whereas attack trees describe an attacker’s actions, we use strategy trees to represent the hierarchical set of actions a developer takes to resolve a security defect. Figure 2.3 depicts an example strategy tree.

### 2.4.2 Strategy Extraction

We performed two additional passes through the transcripts and recordings. This analysis took place after the questions analysis had been completed and was conducted separately. In the first pass, we considered participants’ screen recordings and think-aloud verbalizations to identify their defect resolution strategy for each task. We watched the videos, pausing approximately every minute to take notes. Using the definitions of strategies and stopping criteria described in the previous section (Section 2.4.1) we decided what to
include in the strategy trees. This process resulted in 40 strategy trees, which we finally referenced against each other to ensure the use of consistent terminology.

During the second pass we added annotations to each strategy tree. As depicted in Figure 2.3 (lines prefixed with $\pm$), we annotated the trees whenever an action led 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. Some sub-strategies were not observably successful/failure-inducing and were, therefore, not annotated.

The following criteria, though not complete, guided us when in determining when strategies were successful or unsuccessful. Whenever we observed an outcome that con-
tributed to a participant’s understanding of the problem or demonstrated progress toward a solution, it was marked as a success. Whenever we observed the participant making a verifiably incorrect statement, abandoning a strategy without achieving the desired result, or overlooking information they sought, it was marked as a failure.

2.4.3 Strategy Review

A second researcher who had not been involved in the original strategy tree extraction assessed the completeness and validity of the strategy tree analysis. Rather than duplicate the strategy tree extraction process, the researcher reviewed the existing trees. Extracting the strategy trees from scratch consumed a considerable amount of time, approximately 50 hours. Roughly, the process involved watching each of the 40 task videos, pausing every minute to take notes, and iteratively cross-checking all the trees for consistent terminology. In comparison, reviewing the existing trees took 10 hours.

To assess the completeness of the strategy trees, the reviewer was instructed to, “consider whether each strategy tree is missing any important elements.” The reviewer commented on 3 of the 40 tasks (7.5%). These discrepancies were resolved after they were discussed by the researchers. Relatively few discrepancies between evaluators suggests the trees capture most strategies participants executed.

To validate the extracted strategy trees, the second researcher reviewed each tree while watching the screen/audio recording corresponding to that tree. The reviewer was instructed to confirm the presence of each strategy tree element in each video and placed a check mark next to tree elements as they were observed. This process resulted in the reviewer checking off all of the tree elements, which suggests our strategy trees accurately reflect actions that actually occurred.
2.5 Data Analysis — Assumptions

Like strategies, assumptions can satisfy developers information needs. Unlike strategies, assumptions do not comprise any actions and instead represent an accepted truth without proof. To draw a programming analogy, one can think of assumptions as null strategies — satisfying an information need without investigation.

We identified participants’ assumptions and analyzed each assumption to determine if it was correct. For the purposes of our study, assumptions are defined rather narrowly so that they can be identified unambiguously. Specifically, we only consider the assumptions participants explicitly state using the word “assume” or related words. This choice enables us to accurately identify assumptions, but underapproximates the assumptions made during our study.

Participants likely made many assumptions implicitly that we did not capture. Because such implicit assumptions are pervasive [Fai03], it would be intractable to identify them all. For example, participants 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 participants explicitly stated.

To identify the explicit assumptions participants made, we searched the transcripts for keywords. To derive the search set, we started with the following base words: assume, presume, reckon, believe, and guess. Next, we applied lemmatisation to identify all the inflected forms of these words, such as hypothesize, speculate, suppose, etc. Finally, we searched the transcripts for the stems of all the words identified by lemmatisation.

This search process returned some false-positives. For example, one participant referred to the interviewer’s assumptions rather than making an assumption of his/her own asking,
“Am I assumed to have some prior knowledge?” To filter only the assumptions participants made about the task or the code, we inspected each search result.

After determining the result was an assumption, we evaluated its correctness. For example, one participant (P4) 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.\(^1\) We discuss this particular assumption in Section 2.6.2.4 and the rest of the assumptions throughout Section 2.6.

\(^1\)cve.mitre.org
<table>
<thead>
<tr>
<th>Group</th>
<th>Category</th>
<th>Tasks</th>
<th>Clusters</th>
<th>Location in Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerabilities, Attacks, and Fixes</td>
<td>Preventing and Understanding Potential Attacks</td>
<td>T1 – T4</td>
<td>11</td>
<td>Section 2.6.1.1</td>
</tr>
<tr>
<td></td>
<td>Understanding Approaches and Fixes</td>
<td>T1 – T4</td>
<td>10</td>
<td>Section 2.6.1.2</td>
</tr>
<tr>
<td></td>
<td>Assessing the Application of the Fix</td>
<td>T1 – T4</td>
<td>9</td>
<td>Section 2.6.1.3</td>
</tr>
<tr>
<td></td>
<td>Relationship Between Vulnerabilities</td>
<td>T2 – T4</td>
<td>3</td>
<td>Section 2.6.1.4</td>
</tr>
<tr>
<td>Code and the Application</td>
<td>Locating Information</td>
<td>T1 – T4</td>
<td>11</td>
<td>Section 2.6.2.1</td>
</tr>
<tr>
<td></td>
<td>Control Flow and Call Information</td>
<td>T1 – T4</td>
<td>13</td>
<td>Section 2.6.2.2</td>
</tr>
<tr>
<td></td>
<td>Data Storage and Flow</td>
<td>T1, T3, T4</td>
<td>11</td>
<td>Section 2.6.2.3</td>
</tr>
<tr>
<td></td>
<td>Code Background and Functionality</td>
<td>T1 – T4</td>
<td>17</td>
<td>Section 2.6.2.4</td>
</tr>
<tr>
<td></td>
<td>Application Context/Usage</td>
<td>T1 – T4</td>
<td>9</td>
<td>Section 2.6.2.5</td>
</tr>
<tr>
<td></td>
<td>End-User Interaction</td>
<td>T1, T3, T4</td>
<td>3</td>
<td>Section 2.6.2.6</td>
</tr>
<tr>
<td>Individuals</td>
<td>Developer Planning and Self-Reflection</td>
<td>T1 – T4</td>
<td>14</td>
<td>Section 2.6.3.1</td>
</tr>
<tr>
<td></td>
<td>Understanding Concepts</td>
<td>T1 – T4</td>
<td>6</td>
<td>Section 2.6.3.2</td>
</tr>
<tr>
<td></td>
<td>Confirming Expectations</td>
<td>T1 – T4</td>
<td>1</td>
<td>Section 2.6.3.3</td>
</tr>
<tr>
<td>Problem Solving Support</td>
<td>Resources and Documentation</td>
<td>T1 – T4</td>
<td>10</td>
<td>Section 2.6.4.1</td>
</tr>
<tr>
<td></td>
<td>Understanding and Interacting with Tools</td>
<td>T1 – T4</td>
<td>9</td>
<td>Section 2.6.4.2</td>
</tr>
<tr>
<td></td>
<td>Vulnerability Severity and Rank</td>
<td>T1 – T4</td>
<td>4</td>
<td>Section 2.6.4.3</td>
</tr>
<tr>
<td></td>
<td>Notification Text</td>
<td>T1 – T4</td>
<td>3</td>
<td>Section 2.6.4.4</td>
</tr>
<tr>
<td></td>
<td>Uncategorized</td>
<td>T1 – T4</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
2.6 Results

In the next four sections, we discuss our study's results using the categories we described in Section 2.3. Due to their large number, we grouped the categories to organize and facilitate discussion about our findings. Table 2.3 provides an overview of these groupings. The table also describes which tasks each question category occurred in (e.g., we observed questions from the Understanding Concepts category in all four tasks).

For each category, we selected several questions to discuss. A full categorization of questions can be found in Appendix A along with the full descriptions of participants strategies and assumptions [Exp]. 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.

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.

The structure of most results categories consists of five parts: an overview of the category, several of the questions we selected, a discussion of those questions (RQ1), a discussion relating the category to questions from previous studies, and a discussion of how participants answer the questions in that category (RQ2). Some sections contain less discussion than others either because participants’ intentions in asking questions were unclear, or participants asked questions without following up or attempting to answer them at all.
To answer RQ2, we describe developers’ defect resolution strategies. These strategies for answering individual questions often overlap, especially within a given category. Accordingly, to present a coherent discussion and avoid duplication, we discuss strategies for each category rather than for each question.

In this section we also describe participants’ correct and incorrect assumptions (also RQ2) as well as elaborate on how those assumptions contributed to defect resolution. While performing their tasks, participants made different types of assumptions. Overall, we observed 73 total assumptions — 27 incorrect and 46 correct. We identified assumptions across all participants, except for P3, who may have made assumptions without stating them explicitly. Additionally, participants stated at least one assumption during each task.

2.6.1 Vulnerabilities, Attacks, and Fixes

2.6.1.1 Preventing and Understanding Attacks (10)

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 2.2) 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 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, P2 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.

Previous studies have identified related information needs pertaining to preventing and understanding bugs, problems, defects, and failures. For example, several questions from the prior literature appear similar to the questions we identified: “Is this a problem?” [Ko07]; “What does the failure look like?” [Ko07]; “How do I debug in this environment?” [LaT10b]. However, the difference in terminology between these prior findings and ours (problems/-failures/bugs vs. attacks) reflects the novelty of our contribution. As we discuss in Section 2.3, attacks require special consideration because they originate from a malicious agent.

**Strategies and Assumptions:** Participants used various strategies to answer questions about attacks. When available, 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 getting information about different types of attacks were prone to two types of failures. First, because web search engines were not fully aware of participants’ programming contexts, they returned information about a superset of the relevant attacks. For example, P2 searched for attacks that exploit unvalidated input vulnerabilities. Searching the web 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, including iTrust. Devoting more effort to buffer overflow attacks would have distracted P2 from the 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 false positive. By failing to consider cross site scripting attacks, participants overlooked the program path that exposed a true attack.

### 2.6.1.2 Understanding Approaches and Fixes (8) [10]

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.
Some notifications, including those for the SQL Injection and Predictable Random vulnerabilities, explicitly offered fix suggestions. 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 promising suggestion on StackOverflow, 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 [Owaa]. Because the participant recognized OWASP as “the authority on security,” the participant clicked the link and used it to make a final decision regarding the vulnerability. It seemed important to P8 that recommended approaches came from trustworthy sources.

Previous studies have similarly observed that developers ask questions like those in this category. For instance, Ko and colleagues report developers ask, “What data structures or
functions can be used to implement this behavior?” while considering the ‘space of existing reusable code’ [Ko07]. Additionally, it has similarly been reported that developers ask, “Which function or object should I pick?” [LaT10b] and need information about alternative upstream frameworks (“Comparison with similar upstreams”) [Hae13]. These previous findings pertain to general programming tasks or implementations of new behavior, which is the primary difference compared with our results. Our results reveal the salience of these questions to developers working on defective code.

**Strategies and Assumptions:** Participants’ strategies for acquiring information about 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.

### 2.6.1.3 Assessing the Application of the Fix (9)

Once participants had identified an approach for fixing a security vulnerability (Section 2.6.1.2), they asked questions about applying the fix to the code. For example, while consid-
ering 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 Approaches and Fixes (Section 2.6.1.2). 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.

Based on the identification of these questions, assessing the application of fixes is important in a security context, but perhaps not unique to security. Prior studies have identified some questions that are similar to those asked by our participants: “Did I make any mistakes in my new code?” [Ko07]; “What are the implications of this change for API clients, security, concurrency, performance, platforms, tests, or obfuscation?” [LaT10b]; “Will this completely solve the problem or provide the enhancement?” [Sil08].

**Strategies and Assumptions:** 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.

### 2.6.1.4 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)

Previous studies report questions about relationships between code elements, but not defects [Fri10; LaT10b]. For instance, “Which other code that I worked on uses this code pattern / utility function?” [Fri10]. Ko and colleagues briefly discuss the relationships between defects as it relates to the question: “Is this a legitimate problem?” Sometimes developers answered this question by determining if the same problem had been reported by others as a duplicate [Ko07].

**Strategies and Assumptions:** Participants asked a few passing questions about the relationships between vulnerabilities, but were not explicitly asked to relate the vulnerabilities to each other across tasks. As a result, we did not observe the emergence of any predominant strategies or assumptions in this category.

### 2.6.2 Code and the Application

#### 2.6.2.1 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 2.6.2.2 and 2.6.2.3, 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 to 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.

Asking ‘Where’ questions and locating information has been broadly discussed in the prior literature, though the types of information sought depend on the study. Some developers asked questions about locating recent changes made by teammates such as “Where has code been changing [this week]?” and “Where have changes been made related to you?” [Fri10]. Others ask questions about the functionality of the code, more resembling the questions we identified: “Where is this functionality implemented?” [LaT10b]; “Where is this defined?” [LaT10b]; “Where is this variable or data structure being accessed?” [Sil08]; “Where are instances of this class created?” [Sil08]. None of Ko and colleagues’ questions refer to locating information, but they report, as we will discuss subsequently, that participants used search tools to find information. [Ko07].

**Strategies and Assumptions:** For the majority of participants, scrolling through the source file was a key component of their strategies for locating information. As Robillard
and colleagues observed, such unstructured navigation may signal ineffectiveness [Rob04]. 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 from the start, many participants first scrolled through the package explorer and open files, failed to find the information they needed, then switched to using tools. We hypothesize that developers did not use search tools because of a lack of familiarity and that knowledge of tools improves developers’ effectiveness in resolving security defects.

2.6.2.2 Control Flow and Call Information (10)

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?

Call information is not a security-specific information need. For instance, prior studies have identified the following related questions: “What's statically related to this code?” [Ko07]; “When during the execution is this method called?” [Sil08]; “Where is this method called or type referenced?” [Sil08]. Further, Latoza and Myers identified a similar category of ‘Hard-to-answer questions’ in their study. [LaT10b] Many questions they report overlap with those that we report. However, they do not report questions about the distribution of call sites, in other words, whether calls all originate from the same source or multiple possible call sites. As we discussed above, this type of question had security implications, because it helped our participants assess whether a vulnerability could be exploited.

**Strategies and Assumptions:** Participants used various strategies to obtain information about control flow. 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 Eclipse’s MARK OCCURRENCES tool (code highlighting near (B) in Figure 2.1), 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 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.

We hypothesize that developers first default to the tools and techniques, like scrolling or using MARK OCCURRENCES, that are easiest for them to invoke. Which tools are easier to use may depend on an individual developer's familiarity.

2.6.2.3 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 2.6.2.2 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.

In their study, Sillito and colleagues identified two questions that relate to the questions in this category: “What data can we access from this object?” and “What data is being modified in this code?” [Sil08]. Additional questions arose in the study by Latoza and Myers, who identified a similarly named category [LaT10b]. Unlike these prior studies, we observed some unique questions with security implications. In our study, for example, participants asked whether data reached sensitive contexts (e.g., databases or web displays) and also whether all data passed through sanitization/validation before reaching those contexts. This suggests that developers might require dedicated tool support — beyond what is provided by general navigation tools — to answer these questions.

**Strategies and Assumptions:** 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.

Some participants failed because they used aggregation strategies, rather than tracing individual data elements. This was particularly evident for Task 3, where 12 variables were aggregated in a form object. By tracing individual variables, some participants uncovered the incorrect validation function. Others, like P6, grew tired of tracing each variable and
instead considered the form as a whole. As a result, the participant overlooked the validation function and incorrectly concluded the vulnerability was a false positive.

Rather than investigate, participants also made assumptions about where data comes from, whether it be from a user or hard-coded test data. For Task 1, P1 incorrectly assumed data was passed in from a user. He followed up on this weakly-held assumption by further investigating the origin of the data. P1’s successful investigation strategy, which included the use of the CALL HIERARCHY tool, led him/her to correct the initial assumption and resolve the defect correctly.

2.6.2.4 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 2.6.2.5 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 P2 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.

Questions about code background have been commonly reported by previous studies: “How much work [have] people done?” [Fri10]; “Who created the API [that I’m about to change]?” [Fri10]; “Why was this code implemented this way?” [Ko07]; “Why was it done this way?” [LaT10b]; “Why wasn't it done this other way?” [LaT10b]; “Was this intentional, accidental, or a hack?” [LaT10b]; “When, how, by whom, and why was this code changed or inserted?” [LaT10b]. One difference in a security context is the subtext of the questions. As we discussed above, our participants asked about rushed components and components designed by novices, because those components might be more vulnerable. Answering questions about code background could help developers allocate their scarce security resources.

**Strategies and Assumptions:** To answer questions in this category, participants relied primarily on their prior knowledge of the system. Although they were prohibited by our study design, developers in collocated teams also seek this design rationale from the author of the code through face-to-face conversations [Ko07]. We observed relatively few instances where participants executed additional strategies, such as looking at version control information or system documentation, 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 DBBuilder module were only called during development and deployment when iTrust first starts up.

Participants also sought information about external libraries and commonly made assumptions about these external libraries, their correctness, and their functionality. Some assumptions centered around the proper usage of an external library. For instance, while skim-
ming SecureRandom’s documentation during Task 2, P8 incorrectly assumed SecureRandom does not provide a `nextInt` method. Based on that assumption, P8 proposed a workaround that was valid, but duplicated the functionality `nextInt` already provided.

Other assumptions pertained to the security of external libraries. For example, P7 correctly assumed that SecureRandom handles the secure seeding of random number generators. In contrast with P8, P7’s correct assumption led to a more succinct solution. P4 stated that he/she typically uses the Django framework for web applications and assumed that using such external library frameworks meant that best security practices were being followed. Though this assumption did not directly impact any of P4’s tasks, it illustrates a potentially troubling trust for external libraries. Unfortunately, web framework libraries like Django are susceptible to their own vulnerabilities, many of which have been enumerated in online databases.² We hypothesize that developers look for shallow cues, like a familiar name (Django) or certain keywords (Secure) while assessing the trustworthiness of external libraries.

2.6.2.5 Application Context and Usage (9)[9]

Unlike questions in Section 2.6.2.4, 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)

²cve.mitre.org
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.

Prior studies report that developers ask about how code is generally intended to be used: “What is the purpose of this code” [Ko07] “What is the intent of this code?” [LaT10b]. More closely related to the questions in this category, developers ask, “How do test cases relate to packages/classes?” [Fri10]. As we will discuss below, in a security context participants asked this type of question to determine whether to scrutinize a module.

Strategies and Assumptions: 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 answer their questions about tests, 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.

Participants’ strategies for this task were undermined by the FSB “Bug Explorer” view, because it obscured many of the usual cues participants would have used to differentiate between contexts. In iTrust, tests are organized into a test package, which is separate from the application code. By default, the “Bug Explorer” view closes the package explorer and omits package names from the notification text. Without visible package names or the package explorer, some participants were misled. For instance, P9 incorrectly inferred the code for Task 3 was contained within a test class before eventually correcting the mistake,
“This one I would be more inclined to ignore, because it’s a test method I believe... Wait no, it’s not a test method. No, it’s not.” Had P9 not corrected this mistake, P9 would have misdiagnosed the error and ignored a true positive.

### 2.6.2.6 End-User Interaction (8)}

Questions in this category deal with how end users might interact with the system or a particular part of the system. Some participants wanted to know whether users could access critical parts of the code and if measures 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. While working on the same task, four participants also asked whether end-user input reached the code being analyzed.

The questions in this category are most closely related to reachability questions, and with some effort could be rephrased as such [LaT10a]. For instance, *Is there input coming from the user?*, asks whether there is a feasible upstream trace originating at a user input function reaching the current statement. Otherwise, prior studies that report information needs in general programming contexts say little about end-user interactions.
However, these questions do pertain to security research on attack surfaces [How05] and attack surface approximation [The15]. An attack trace, or the sum of all paths for untrusted data into and out of a system, describes where end-user input interacts with a system. We hypothesize that providing developers attack surface information, such as whether a program point is on the attack surface, could help them answer the questions in this category.

**Strategies and Assumptions:** For participants, answering their questions required manual inspection of the source code. For instance, P6 found a `Validator` method, which P6 manually inspected, to determine if it was doing input validation. He incorrectly concluded that the `Validator` method adequately validated the data.

P2 used `CALL HIERARCHY` to trace end-user input; P2 assumed the vulnerability was a true positive if user input reached the vulnerable line. 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.

More generally, 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 2.6.2.1), examining Control Flow (Section 2.6.2.2), retracing Data Flow (Section 2.6.2.3) information, and recalling information about the Code Background (Section 2.6.2.4). Participants successfully answered questions in this category when they successfully synthesized the results of these other efforts.
2.6.3 Individuals

2.6.3.1 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, P6 wanted to fully understand the notification before starting to explore, so P6 went back to reread the notification before investigating further. These questions occurred in all four vulnerabilities. We identified two questions in the prior literature that could be categorized here: “What am I supposed to work on?” \[Fri10\] and “Is the problem worth fixing?” \[Ko07\].

**Strategies and Assumptions:** Compared to other categories, the strategy and assumption analysis results in this category are sparse. Although we asked participants to think-aloud, they seemed to resolve questions in this category internally.
2.6.3.2 Understanding Concepts (7)

Some participants encountered unfamiliar terms and concepts in the code and vulnerability notifications, which prompted them to ask questions.

*What is this concept? (6)*

*How does this concept work? (4)*

*What is the term for this concept? (2)*

For some vulnerability patterns, FSB links to resources that define relevant concepts. Otherwise, information about concepts and terminology could be found online. When asked what information P4 would like to see added to the notification for the Servlet Parameter vulnerability, which did not include any links, P4 noted that the notification should include links defining a servlet is and how it related to client control.

**Strategies and Assumptions:** Participants’ strategies for finding information about concepts included clicking the links provided by FSB. For example, while assessing the Potential Path Traversal vulnerability, P2, unsure of what path traversal was, clicked a link, labeled “path traversal attack,” provided by FSB to get more information. However, this strategy failed when the hyperlink text poorly described the links’ contents. Unable to determine the quality and type of information hidden behind FSB’s links, some participants preferred to search the web for information about concepts.

If a link was not available or if its contents were unclear, participants went to the web to get more information. For instance, P7 and P8 both searched the web for the same unfamiliar term (‘CSRF token’) while working on the Predictable Random vulnerability — for this task, FSB did not provide any links.
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 demonstrated the simple, yet successful, strategy of iteratively refining their search terms. For example, after performing a search that failed to return relevant results, P1 added specificity to the 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.

2.6.3.3 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?*

**Strategies and Assumptions:** We did not observe many overarching strategies or assumptions in this category, in part because participants rarely signaled to us when they were testing their expectations. When we did observe participants questioning their expectations, their strategies varied situationally.

2.6.4 Problem Solving Support

2.6.4.1 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 P2 did not see that information, P2 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, P10 stated that normally team members could 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.

Prior information needs studies discuss the use of resources and documentation. For instance, Fritz and Murphy report that developers rely on team members for code review, “Who has knowledge to do code review?” In software ecosystems, downstream users rely on the availability of good documentation [Hae13]. Ko and colleagues’s questions do not refer directly to documentation or help resources [Ko07]. Instead, documentation is discussed as a means of obtaining answers to questions. It seems quality help resources and documentation are valued in many contexts, including but not limited to security.

**Strategies and Assumptions:** 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 basic code constructs (i.e. control structures, variable types, method calls, and which variables were being stored in which data structures). However, these strategies sometimes failed to convey security-relevant semantic information. Further, this strategy fell short when participants sought documentation for IDE tools or external APIs; such 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.

### 2.6.4.2 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 [MH12].

Other information needs studies discuss how developers use tools to answer questions and that there are mismatches between questions asked and those that tools can answer [Ko07; Sil08]. However, relatively few studies report any questions about tools themselves.

**Strategies and Assumptions:** 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 wanted to navigate to methods up the call hierarchy, but struggled to identify an appropriate tool. P2’s 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 P2 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.

When FSB failed to effectively communicate information about the location of defects to participants, they made assumptions. For Task 3, P10 incorrectly assumed FSB was indicating an issue with the database access objects (DAOs), rather than the sanitization methods. He probably made this assumption, because FSB placed its bug markers in the file containing the DAOs and not in their associated sanitization methods. Based on that assumption, P10 proposed adding a sanitization method to those objects. This fix is incorrect, because sanitization methods already existed in another class and just needed to be modified. The proposed fix would have resolved the vulnerability locally, but also would have introduced redundant code and violated iTrust’s architectural convention of organizing all validator classes in the /validate folder. Furthermore, any component still using the faulty sanitization methods would still be vulnerable.

2.6.4.3 Vulnerability Severity and Ranking (5)

FSB estimates the severity of each vulnerability it encounters and reports those rankings to its users (Table 2.2). 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 the assessment process for the following vulnerabilities. As P1 began working through the final vulnerability (SQL Injection), P1 admitted that he did not understand the tool's metrics as well as he thought. He wasn't sure 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.

Participants in Ko and colleagues’ study made similar inquiries about the severity of defects, asking: “Is this a legitimate problem?” [Ko07]; “How difficult will this problem be to fix?” [Ko07]; “Is the problem worth fixing?” [Ko07].

**Strategies and Assumptions:** Severity ratings typically help developers triage vulnerabilities. Due to the limitations of our controlled study, we preselected the vulnerabilities and the order in which they would appear. As a result, working with the severity rankings was not critical for participants to complete their tasks. Therefore, we observed few assumptions and relatively shallow strategies in this category; some participants asked questions about the rankings, but most did not follow up with an investigation.
2.6.4.4 Notification Text (6)

FSB provided short and long descriptions of each vulnerability (See (A) and (C) in Figure 2.1, respectively). 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 predictable 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.

As we discussed in Section 2.6.4.2 prior information needs studies discuss the use of tools, but identified relatively few questions about the tools themselves. Similarly, relatively few questions about tool notifications have been reported by previous studies. Sillito and colleagues identify, “Where in the code is the text in this error message or UI element?” which is similar to our, *What is the relationship between the notification text and the code?* [Sil08]. Ko and colleagues report that their participants struggled to make sense of tool notifications, “Three developers used static analysis tools to check for fault-prone design patterns, but could not understand the tools’ recommendations” [Ko07]. These
questions likely arise in our study, because our focus was on how participants interacted with a static analysis tool.

**Strategies and Assumptions:** Participants’ defect resolution strategies appeared to include reading portions of the notification text. 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 by reading the notification text for 2 of 8 tasks.

The distinctions between these two workflows affected how participants built hypotheses. Low-familiarity participants built their initial hypothesis from the notification and tested that hypothesis against the code. On the other hand, high-familiarity participants used the notification to refine their already-formed hypotheses. It remains an open question whether tool notifications currently support one of these workflows more than the other. The prominence of explanatory text and reference information over resolution shortcuts might suggest FSB caters to the hypothesis-building novices. Either way, the presence of two distinct workflows suggests tools should support both.

Participants’ strategies also varied in terms of how much of the notification they read. Some participants read the entirety of the message before proceeding, whereas others only read the short descriptions. Several participants read the visible parts of the long description, but neglected to scroll further to reveal the rest of the message. Often, this led to problems. For instance, P3 struggled with the Predictable Random vulnerability because P3 initially failed to find documentation for `SecureRandom`. P3 started the task by reading only the first half of the long description. After finally finding the documentation online and completing the task, the interviewer directed P3’s attention to the overlooked second half of the long description. P3 read the rest of the description and realized it contained the
sought-after information, “I didn't scroll down that far. I see it even has a link to generating strong random number. I could have clicked on this link.”

As an aside, it is possible that pointing out the FSB links to P3 at this point in the study could have influenced the subsequent two tasks. However, the FSB notifications for the two subsequent tasks did not include any links, so even if P3 had been artificially influenced by our interruption, the interruption likely had little effect on the final two tasks.

2.7 Discussion

In the previous section we described the categories of information developers need to resolve security vulnerabilities. We found that developers need diverse categories of information pertaining not just to where potential problems exist; we found that developers need information about: attacks, vulnerabilities, the software itself, and even the social ecosystem that built it.

The previous section also describes the strategies we observed developers using to acquire the information they need. In this section, we summarize how some of those strategies led to developers’ successes and failures and outline visions for three tools based on these findings.

2.7.1 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 2.6.2.1, 2.6.2.2, 2.6.2.3, 2.6.2.6). 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 [Fri14]. Additionally, when a developer only contributes to a portion of the system, as is often the case in the open source community [Moc02], the developer may be unable to reason about the system-wide implications of a change.

Much like work that examines more general programming tasks [LaT10b], 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 [Lat] and trace control flow to its origin [Bar14], in a security context, these tools share the same limitations as the existing heavyweight tools. Existing tools like CodeSonar [Coda] 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.

To address these strategic navigation failures, we implemented tool, called Flower (Figure 2.4), as a plugin for the Eclipse IDE [Smi17]. We compared Flower to Eclipse's existing suite of navigation tools in a preliminary evaluation with eight programmers.
performing two code navigation tasks. We failed to identify a significant difference in overall completion time and correctness between Flower and the baseline. Upon further inspection, participants began navigating strictly faster with Flower, however they were slower when they reached complex control structures.

2.7.2 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 2.6.4.1, 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.

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 [Par12].

Based on these observations, we designed a mockup for a new tool, called Strategy-Check, that could address this coordination challenge developers face (Figure 2.5). Much like our design for strategy trees, Strategy-Check represents strategies as hierarchical lists. This organization reflects the notion that certain low-level strategies could contribute to higher-level strategies. Previous research suggests that decomposing programming problems into goals and subgoals in this way can reduce cognitive load and enable more effective learning [Mor15]. Our design also includes checkboxes to allow developers to track their progress. Previous research suggests that checklists can effectively guide developers [Pha09]. Finally, each encapsulated strategy includes links to resources that relate specifically to that task and tool suggestions that could help developers complete the task. Previous
research suggests that both novice and experienced developers face problems of tool awareness [MH12] and timely tool recommendations might be the most effective [Bro17]

2.7.3 Context-Informed Web Search

Accessing web resources was an important element of participants’ strategies across several categories (Section 2.6.1.1, Section 2.6.1.2, Section 2.6.3.2, Section 2.6.4.1). To find success, participants sought information that was scattered around different sites. Depending on the task and an individual’s approach, participants found critical clues hidden on StackOverflow, official documentation pages, and personal blogs, for instance.

To reach these helpful web resources, participants either followed the links that FSB had curated, or used an online search engine. Both of these approaches are failure-prone. However, by drawing inspiration from the positive aspects of each of these approaches,
we will propose a new tool-assisted approach for gathering resources from the web while diagnosing a static analysis defects. We will refer to this approach as context-informed web search.

FSB includes links to web resources in its description of some defects under the header “References” (Figure 2.6). FSB’s designers assumedly deem these references relevant and helpful, and, when references to the right types of information were available, participants tended to use them. Unfortunately, the links in each references section cover a limited number of topics. As in the example depicted by Figure 2.6, the reference material might describe the defect and how to exploit it, but not provide code examples for remedies. In practice, participants sought information and visited sites beyond what was available in the finite reference sections. To summarize, the references encapsulate the tool designer’s expertise and knowledge of a specific defect pattern, but are not tailored to the user’s needs.

Participants also used search engines to locate web resources, sometimes as a primary means, other times after having exhausted the FSB links. With this approach, participants had to sift through long lists of irrelevant results to find useful information. As described in Section 2.6.3.2, some participants iteratively refined their search terms to filter out as many irrelevant results as possible. Others simply tested link after link until they found something useful or gave up.
The two approaches for accessing web resources described so far (using tool links and performing a search) fail in complementary ways. The tool’s list of links only cover a narrow set of topics, with each link highly relevant to the defect reported. Conversely, search engines over-approximate the set of relevant resources, but cover far more topics of interest.

We envision a new way for developers to access web resources while resolving defects with static analysis, context-informed web search. This approach improves on existing approaches by combining the breadth of web search with the relevance afforded by a tool’s contextual awareness. Users would perform a context-informed web search just as they would any other web search, but enter their query into a specialized search engine, rather than a general-purpose search engine. The specialized search engine would work in tandem with the developers’ static analysis tool and would start by fetching results just like a general-purpose search engine. Before returning those results to the user, the specialized search engine would query the static analysis tool for details about the developers context — information the tool has already computed in order to detect defects. Finally, the specialized search engine would only display context-relevant results.

Consider, for example, a developer who searches for ‘secure random number.’ A general-purpose search engine would return results targeted at Ruby, Javascript, Android, and Java developers. The specialized search engine could conclude from the static analysis tool that the developer was using Java and importing java.util.Random and filter out irrelevant results pertaining to other languages.

This information searching approach draws inspiration from several prior efforts. The Mica tool [Sty06] analyzes search results and presents users with information about which results might be most relevant, compared with our approach, which proposes taking information from the source code and a static analysis tool. Rahman and colleagues [Rah14] and Goldman and Miller [Gol09] similarly describe adapted search engines, theirs based in the
IDE. These previous works create search queries using information from the source code and from stack traces. Our proposed approach differs in that we also consider contextual information from a static analysis tool.

2.8 Threats to Validity

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.

Our mitigation strategy of recruiting participants who were familiar with the iTrust code base introduced its own threat. This design decision limited our population to students and developers who studied at North Carolina State University at some point in time. The participants we studied limit generalization and may not represent the range of developers who would use security tools. For instance, we likely cannot completely generalize these results to security experts — none of our participants self-identified as such.

The fact that this study was conducted in a controlled environment rather than an industrial development setting raises a threat to the external validity of our results. 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 questions might differ based on experience. 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.

Another reason we cannot claim our categorization is comprehensive is because the questions and strategies may have been dependent on the four FSB notifications we chose and the order they were presented in. We did not study all types of security defects. In fact, there are many defects that are not even detected by FSB. Although we chose the four notifications to span different topics as categorized by FSB, there may be information needs and strategies these topics did not expose.

As we discussed in Section 2.5, participants likely made many assumptions without explicitly stating them. Participants might have also felt obliged to actively pursue strategies while being observed in a lab setting. As a result, we likely identified fewer assumptions than participants actually made. Therefore, relative to the results for RQ2 where strategies were visible, there are fewer assumptions to discuss. To mitigate this threat, we used stemming and lemmatisation to capture as many explicitly stated assumptions as possible, specifically those where participants did not just use the word “assume.” Despite the scarcity of assumptions in our results, we believe they are worthwhile to include because they give us insight into how developers sometimes choose to satisfy information needs without actively executing a strategy.

Participants also may have spent an unrealistic amount of time (either too much or too little) on each task due to working outside their normal environment. To counteract this threat, we did not restrict the amount of time allotted 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 (s)he would in her normal work environment.

An additional threat is that the think aloud protocol may have influenced participants actions. Consequently, participants may have had different information needs or strategies than they would in the wild. For example, developers may have reflected more on the vulnerabilities than they would have in their normal working environments, causing them to approach them more carefully. This is perhaps evidenced by the existence of the Developer Planning and Self-Reflection category (Section 2.6.3.1).

### 2.9 Conclusion

To answer RQ1 and RQ2, this study explored how developers resolve security defects while using static analysis. To that end, we asked ten software developers to describe their thoughts as they assessed potential security vulnerabilities in iTrust, a security-critical web application. The results of this study are a categorization of questions and a catalog of strategies for answering those questions. This work advocates for tools that not only detect vulnerabilities, but also help developers actually resolve those vulnerabilities. Our findings have several implications for the design of static analysis tools. Most broadly, tools should support effective strategies and provide information that aligns with the information needs we have identified. In particular, these results suggest that tools should help developers, among other things, search for relevant web resources.
In Chapter 2 we observed ten software developers as they diagnosed security vulnerabilities using a static analysis tool. One limitation of that study is that none of those participants self-identified as security experts (see Table 2.1). In this chapter, we answer RQ3 (How do specialized developers on security red teams use static analysis?), which helps us generalize our thesis beyond the narrow population of ten students and developers we previously studied.

3.1 Introduction

In this chapter, we explore the emerging role of offensive security engineering teams, or red teams within organizations—comprised of full-time software security engineers that emulate malicious attackers. These red teams spar against their defensive blue team counterparts, who conduct routine incident response and monitoring. Large software

I’d like to thank Titus Barik for contributing to the study presented in this chapter.
companies, like Amazon, Dropbox, Facebook, and Microsoft, have embedded full-time red team operations alongside their blue team activities to challenge their security readiness and improve their defensive security posture.\(^1\) Although software security engineers are in some respects software engineers (like those we studied in the previous chapter), they also have several consequential differences in how they write, maintain, and distribute software.

We conducted 17 interviews across five red teams within a large organization. Through these interviews, we find that software security engineers have substantial impact in the organization as they harden security practices, drawing from their diverse backgrounds. Software security engineers are both agile yet specialized in their activities, and closely emulate malicious adversaries—subject to some reasonable constraints.

This work contributes an explanation of the role of software security engineers in red teams through first-hand accounts of professional software security engineers. The results of this work are useful to practitioners, researchers, and toolsmiths who wish to understand how offensive security teams operate, situate, and collaborate with partner teams in their organization.

### 3.2 Methodology

**Research context.** We conducted the study at Microsoft, a large software company with both red team and blue team software security operations.

**Recruitment.** We used a combination of random sampling and snowball sampling to recruit potential informants for our study. We initially randomly sampled informants from the company’s address book by searching for full-time engineers with the title of

\(^1\)We used publicly-posted LinkedIn job descriptions to identify other organizations with full-time red teams.
“Software Security Engineer,” having at least three years of experience within the company. We used snowball sampling and asked current informants to connect us with individuals who engaged in very different activities from themselves. Informants were compensated for their time with meal vouchers. Appendix B includes the initial script for email recruitment.

To reduce interviewer bias and obtain data from different perspectives during the interviews, each interview was conducted by two researchers. It is important to note that not all topics were discussed at the same level of detail with all informants, due to the nature of semi-structured interviews.

**Informants.** We continued interviewing informants until obtaining *theoretical saturation*, that is, the point at which we were obtaining interchangeable stories. This occurred at 17 informants, across five primary software security teams with offensive security engagements (Table 3.1). Informants had a mean experience of $\mu = 5$ years at Microsoft, and a mean experience of $\mu = 8$ years total for security-related engagements. Incidentally, recommendations by Crouch & McKenzie [Cro06] suggest that sample sizes for interview studies be under 20 informants in order to “facilitate the researcher’s close association with the respondents, and enhance the validity of fine-grained, in-depth inquiry in naturalistic settings.”

**Interview protocol.** We conducted semi-structured interviews that focused on the informants’ recent experiences, responsibilities, workflow, collaboration, challenges, motivations, tools, processes, and learning. Using guidelines from Hove & Anda [Hov05], we conducted interviews in pairs, with one interviewer taking the lead while the other takes notes and asks additional questions. We interviewed informants in their own offices, where they often showed us relevant artifacts, such as reports that they had prepared for development teams, custom tools they had built, and their security environment as configured for day-to-day work. We recorded all interviews for later transcription, and interviews typically
Table 3.1 Informants

<table>
<thead>
<tr>
<th>Informant</th>
<th>Security Role</th>
<th>Experience (Yrs)²</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Organization</td>
</tr>
<tr>
<td><strong>Cloud Computing Group</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chase</td>
<td>Cloud Applications</td>
<td>6</td>
</tr>
<tr>
<td>Cliff</td>
<td>Cloud Services</td>
<td>3</td>
</tr>
<tr>
<td>Clara</td>
<td>Cloud Services</td>
<td>2</td>
</tr>
<tr>
<td><strong>Databases Group</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bill</td>
<td>Applications</td>
<td>5</td>
</tr>
<tr>
<td>Brian</td>
<td>Networks</td>
<td>3</td>
</tr>
<tr>
<td><strong>Devices and Gaming Group</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dan</td>
<td>Browser</td>
<td>4</td>
</tr>
<tr>
<td>David</td>
<td>Browser</td>
<td>8</td>
</tr>
<tr>
<td>Dean</td>
<td>Cloud Services</td>
<td>3</td>
</tr>
<tr>
<td>Derek</td>
<td>Operating Systems</td>
<td>6</td>
</tr>
<tr>
<td>Gary</td>
<td>Threat Protection</td>
<td>5</td>
</tr>
<tr>
<td>George</td>
<td>Threat Protection</td>
<td>5</td>
</tr>
<tr>
<td><strong>Enterprise Group</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eric</td>
<td>Enterprise Ecosystem</td>
<td>4</td>
</tr>
<tr>
<td>Evan</td>
<td>Enterprise Ecosystem</td>
<td>3</td>
</tr>
<tr>
<td><strong>Productivity Software Group</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pam</td>
<td>Build Systems</td>
<td>5</td>
</tr>
<tr>
<td>Pat</td>
<td>Client Applications</td>
<td>8</td>
</tr>
<tr>
<td>Paul</td>
<td>Client Applications</td>
<td>4</td>
</tr>
<tr>
<td>Peter</td>
<td>Compliance and Auditing</td>
<td>14</td>
</tr>
</tbody>
</table>

¹ All informants’ titles include “Security Software Engineer.”
² Years of software security experience within the current organization and total experience across all professional security engagements.
lasted just under an hour. We obtained informed consent using Microsoft’s IRB protocol. Appendix B includes the semi-structured interview script.

**Analysis.** We used the thematic analysis procedure described by Braun & Clarke [Bra06], which in summary consists of six phases: 1) familiarizing yourself with your data, 2) generating initial codes, 3) searching for themes, 4) reviewing themes, 5) defining and naming themes, and 6) producing the report. After transcribing the interviews, we used the ATLAS.ti [Sci19] data analysis software for qualitatively coding the data. We performed coding over multiple iterations to search for, review, and define, and name themes. The results of this analysis are found in Section 3.3.

**Validity.** We used several methods to mitigate against common validity threats in qualitative research [Max92], namely: contextual inquiry, disconfirming evidence, prolonged engagements, and thick description. The details of these methods are described in Section 3.4 alongside the corresponding threats to validity.

**Data availability.** Informants provided consent only for the principal researchers to record and have access to the full interview transcripts. We provided assurances that the full interviews would remain confidential and that any information revealed in public reports would be anonymous.

### 3.3 Findings

We identified three top-level themes relating to workflow, security culture, and differences between software engineering and software security engineering. Because the interviews were semi-structured, we covered themes depending on the direction of the conversation. Table 3.2 summarizes those themes and sub-themes from our analysis, with check marks indicating coverage of the theme.
Table 3.2 Coverage of Informant Themes and Subcategories

<table>
<thead>
<tr>
<th>Theme</th>
<th>Informant Interview</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workflow</strong></td>
<td></td>
</tr>
<tr>
<td>Campaigns</td>
<td>9</td>
</tr>
<tr>
<td>Tactics</td>
<td>13</td>
</tr>
<tr>
<td>Collaboration</td>
<td>13</td>
</tr>
<tr>
<td><strong>Security Culture</strong></td>
<td></td>
</tr>
<tr>
<td>Backstory</td>
<td>14</td>
</tr>
<tr>
<td>Realism and Ethics</td>
<td>8</td>
</tr>
<tr>
<td><strong>Software vs. Security</strong></td>
<td></td>
</tr>
<tr>
<td>Organizational Impact</td>
<td>13</td>
</tr>
<tr>
<td>Security Mindset</td>
<td>8</td>
</tr>
</tbody>
</table>
Through our analysis, we found that software security engineers conduct long-term campaigns to exploit specific goals and targets, that team members have unique specializations that they apply to their tactics, and that their work is context-sensitive and requires on-demand tools. Red teams collaborate with partner teams and communicate vulnerabilities to them (Section 3.3.1). Informants were diverse in their backgrounds and told us about their motivations for becoming a software security engineer. We found that red teams closely emulate the activities of malicious adversaries, but differ in their ethical responsibility to prevent harm to the organization (Section 3.3.2). Informants also reported that while software engineers and software security engineers often use the same tools, there are substantial differences in how they make impact in their organization, and through their security mindset (Section 3.3.3).

3.3.1 Workflow

We describe the structure of red team security campaigns with partner teams (Section 3.3.1.1), the tactics used to execute these campaigns (Section 3.3.1.2), and how red teams communicate their results to developers in partner teams (Section 3.3.1.3).

3.3.1.1 Campaigns

A textbook campaign. Informants’ workflows comprise a series of campaigns. Here we characterize a typical campaign to provide a framework for understanding software security engineers activities. Dean describes a recent red team campaign that more-or-less mirrors a textbook offensive kill chain—kill chains describe the different phases of an intrusion [Hut11]. The campaign begins with “customer engagement and scoping, and ends
with reporting. End-to-end, it’s normally six to eight weeks. Everything else in between is testing, and it’s usually based on the environment.”

As Dean explained, campaigns begin with *scoping,* which includes “buy-off,” “meeting with the customer,” “deciding the target,” obtaining “permission to own their stuff and to go attack their things,” and following legal protocol, or “getting legal cover so no one comes and yells at me.” Scoping takes about a week. The next stage is *initial access,* or “cracking the perimeter”; it is “the longest portion of the assessment because we have to find a flaw to get inside the perimeter.” This activity usually involves “a little bit of reverse engineering, using virtual machines, and using the source code to figure out what is what.” Dean adds, “there’s something called ‘assume breach’ where for this campaign we assume one adversary already has credentials within the network. And the reason why that exists is because it’s true.” The next two weeks are *lateral movement,* essentially, “moving around the network to get to wherever we need to go.” The final step is *command and control* where the red teamer installs a remote access tool in order to establish a persistent, interactive channel for exfiltration of data or other intellectual property. At this point, Dean notes, “I am you as far as everything else in the network is concerned.”

**Every campaign is different.** Other campaigns are more open-ended. As Cliff (and Clara) observe, red teams are an emerging role, and “all the teams seem to have different styles of operations: sometimes they go fast and they aren’t expected to maintain persistence, and sometimes they do the full shebang.” Cliff adds, “in some of these cases, it’s months or years. They’re ‘low and slow,’ absolutely, but I’d be scared of what that person was able to do.”

A campaign’s scope is also “very variable and could be anything” (Eric). For example, sometimes the campaign is exploit hunting, “which essentially ends when we have enough bugs to make a chain of exploits that are required to do whatever we want” (Eric). This type
of campaign ends with mitigation proposal to the software engineers. Other campaigns, as Paul describes, “try and figure out where to go by things that either have had very little or a lot of public interest lately. In the middle I don’t care so much. When there is a lot of public interest you feel like it’s likely that someone external is going to discover something pretty critical.” But regardless of the type of campaign, they always have a specific objective, “no matter how you get that to happen” (Chase).

3.3.1.2 Tactics

“It’s definitely a challenge to try and put structure over onto this high-flying, edge-of-your-seat, break-stuff-however-you-can, everything-is-different sort of thing. Boy, is it tough.” (Cliff).

Software security engineers are agile in their abilities, but also specialized, chaining together a sequence of conventional and domain-specific tactics to execute red team activities. Much of the work is exploratory, requiring creativity and insight, and sometimes a bit of luck. As Paul says, “there are definitely some anxiety inducing aspects. I think that’s just the nature of anything super exploratory like that. You have no idea if there is anything to find.” Evan comments, “that’s just the job. You are looking for something that might not exist.” Cliff adds, “you have to be willing to pivot really fast.”

**Specializations within red teams.** Red teams assemble individual software security engineers with specialized skills, allowing team members to offer “different perspectives” (Dan) and make the team overall more effective. They specialize by software domain, such as “desktop applications, browsers, networks, kernels, or virtualization. There are a lot of options” (Derek). Security engineers primarily focus on “where they’re more interested in, such as network security, infrastructure, or deployment. We divide our responsibilities” (Brian). Pat notes, “there’s also quite a lot of specialized knowledge around different types of
vulnerabilities: the sorts of mistakes developers make, or potential problems in processes, for instance.”

But specialization has its disadvantages, particularly if a team’s composition becomes unbalanced: “the vulnerabilities we find are skewed towards the expertise of the people on our team, based on our past experience, and not necessarily how our customers are getting owned in the wild” (Pam). If security engineers overspecialize, it means they are less likely to have a “broader view or understanding all of the modules and what the modules are responsible for” (Paul).

**Context-sensitive work and on-demand tools.** The context-sensitive nature of their work often necessitates that software security engineers create on-demand tools, “because every product is unique” (Chase). As a result, “a lot of our tooling is developed on our own” (Dean, Pam, Pat, Brian, Bill). Using off-the-shelf security tools to identify vulnerabilities is often ineffective, because “it’s really low hanging fruit and rarely yields much. Developers are already using those to filter out most of those bad behaviors.” (Dan, Evan). Dan adds that they do use fuzzing tools, which are developed internally: “We write our own. They are constantly changing.” Though some red teams report using playbooks (Dean)—collections of reusable procedures—others like Bill argue that the “situations are much too custom or unique to provide blanket tactics.” Instead, Cliff argues, “you build stuff for single purposes, especially when you’re writing offensive tools. It’s more about making it work.”

### 3.3.1.3 Collaboration

**Team rotations.** Red teams rotate amongst different engagements, depending on organizational priorities (Dean, Evan, Peter, Eric), and collaborate with project managers and software engineers in the partner teams.
**Communicating vulnerabilities.** Finding vulnerabilities, as Pam says, isn’t just “throwing vulnerabilities over the fence; it’s following up with the team to make sure that they actually fix them.” To support developers, Cliff says, “there’s a science to explaining things to software engineers and project managers.” Bug reports filed against developers, “start with an introduction about the problem, explain step-by-step what is going on and why it’s a bug” (David); “it’s not just explaining, but giving some suggestions on how to fix the issue and that’s an important part of what you’re communicating back to them” (Clara). Dean explains, “we only file bugs if we can prove an exploit. Even if it’s super small.”

Not all vulnerabilities require detailed reports. Developers are generally responsive to fixing issues, when the organization has a strong security culture: “it’s usually in people’s best interest to fix it” (Cliff). For example, say Pat, “some vulnerabilities, like XSS, are more well-known than others. But other vulnerabilities are more complex and less well-known. Eventually you have to work through questions people have and just get them on the same page as you, building up a foundation of tribal knowledge.”

**Developer push back.** Occasionally, developers push back against the red team’s recommendations because they don’t recognize the security implications, or because they must “balance their time between shipping features and fixing bugs” (Chase). There are sometimes challenges in “getting developers to resolve or even triage their bugs” (Peter). In these situations, Paul explains that he will occasionally develop proof-of-concepts or more elaborate “weaponized exploits” when “the developers doesn’t believe that this is a serious thing. You can prove it is by actually weaponizing it.” As Chase says, “but at the end of the day it’s not about being unhackable, it’s about how hard it is to get hacked.”

**Engaging with blue teams.** Red teams “serve as a sparring partners for blue teams” (Dean). The blue teams are “the incident response teams, who run attack investigations in the event of a potential threat, evict the hacker, and basically drive the service back to normal
operations” (Chase). Many of the blue team activities rely on “machine learning, analytics, and combining telemetry obtained from host-based, network, service, and application monitoring” (Gary). When red teams discover an exploit, “they hand it over the blue team for them to signature” (George). These signatures allow the blue teams to identify if the same threat appears from an outside adversary in the future.

<table>
<thead>
<tr>
<th>Workflow Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Like external adversaries, red teams undertake campaigns with goals and targets defined a priori. To tactically achieve these goals, software security engineers must both develop specializations and maintain the ability to pivot quickly to new contexts. Red teams aren’t just responsible for reporting the vulnerabilities they find during campaigns; they must also motivate and enable developers to fix them.</td>
</tr>
</tbody>
</table>

### 3.3.2 Security Culture

We describe the security culture of red teams, which includes how security engineers become interested in software security (Section 3.3.2.1) and the set of shared attitudes, values, and goals that characterize this role (Section 3.3.2.2).

#### 3.3.2.1 Backstory

**Breaking stereotypes.** Do all hackers wear hoodies? “There are definitely stereotypes. Everybody thinks it’s just like this high school hacker in his black hoodie in his mom’s basement. And you look around and it’s a diverse group. We don’t look like your typical hacker. I don’t look like your typical hacker. I think people think of hackers always as a bad person. It’s not necessarily bad. They’re not doing things maliciously to hurt somebody else, they’re doing
it to make it better.” (Clara). Cliff reflects on malicious hacking: “I wouldn't say it's worth it. The black mark is not work it. That’s why I’m white hat in the first place.”

**Motivations.** Software security engineers arrived at their positions via a variety of interests or hobbies, such as “pirating games”, and “hacking on video games” (Bill, Derek), “writing emulators“ (Eric), and “hacking on game consoles in my free time, which led to studying more about crypto, memory corruption issues, how to exploit them, and how they work” (David). Brian adds, “games are where you start. For example, with games, you try to understand how the thing works and then you start to do some small hacking. From there, you start learning about security. I started doing some practical stuff before college.”

Others were introduced to security more formally. For example, Paul “started doing capture-the-flags at University. I’ve always sort of been interested in security before I started formally studying computer science. Once I got to take University I took a couple of courses in security. They were very practical—exploitation based. Very offensive security-based. I met up with some like-minded people there.” Cliff also attributes his success to formal education, adding, “that's funny, without school I probably wouldn't have been in security: we wrote worms, we did cross-site scripting and buffer overflows—stuff that is dated, but this is how it all works. I had never thought about making a program do something that I totally didn't intend for it to do as a developer. I thought, this is really cool.”

**From motivations to careers.** Our informants’ motivations and interests from casual hacking ultimately developed into careers in security. Before joining the organization as a red teamer, some previously worked in national security, military, or the defense industry (Dean, Cliff, Clara). Others, including Brian and Bill, worked previously in security consulting firms as external penetration testers. Two informants came from a prior software engineering role, such as software testing (Pam, Chase), where “through testing, they found security bugs in products” (Chase). Practices, such as creating “playbooks” (Dean) and
using “reverse engineering tools” (David), from these prior experiences were brought into their current red team activities.

Clara observes that, despite these divergent backstories, “the security industry is a pretty tight-knit group,” sharing their knowledge and experiences through “conferences,” “meetups,” and “slack-chat.” “It’s a small world. We chose security because we truly care about it and we want to do something good,” Clara says: “I feel security is the way I can leave an imprint and leave the world a better place.”

3.3.2.2 Realism and Ethics

Emulating the adversary. “We usually set goals at the beginning of our red team campaigns because we want them to be more like real life,” says Chase. Software security engineers are obligated to accurately model malicious actors, yet stop short of actually causing harm, such as data loss. For example, “if the goal is just to own a host, that’s what we’ll do. If the goal is to emulate a persistent threat, we’ll install a remote access tool to maintain access” (Chase, Evan). When using a black box approach that models an outside attacker, “you can say with some kind of certainty that the attacker would be more likely to find what we found, because we followed the same kind of steps. We can say [to our engineers] that you should fix these versus other kinds of bugs because they might be found sooner” (Bill).

Practical shortcuts. Nevertheless, there are also practical constraints that malicious adversaries don’t have. In particular, “it’s usually true that an attacker has no deadline. They don’t have the same boundaries that we have” (Brian). “Because of time constraints, our activities are loud, and they’re more detectable, which means you’re going to get caught more quickly than a sophisticated attacker might. When you’re going against real life attackers you need every advantage you get that’s reasonable” (Cliff).
Consequently, red teams employ principled shortcuts—or “gray boxing” (Dean)—when it makes sense to do so. Cliff explains, “the biggest advantage I get is inside network access. As an employee of [the organization], I get to see whatever is available on the corporate network and I have insider knowledge, both shared by other red teams and from presentations, meetings, and talking to the people who actually wrote the code. That way every project doesn't turn into a 6-month brigade.”

Cliff continues, “of course, the more shortcuts you have, the weaker your argument is when you want to identify something as a problem. But if you think the corporate network is a boundary, we are already at a problem.” This might seem like an unrealistic assumption, but as Pam notes, “there have been some high-profile incidents at other companies where attackers have gotten into the network somehow. We don't want to be the next one.”

**Ethics.** Software security engineers both acknowledge and take seriously the ethical responsibility of white hat hacking, voluntary choosing to give up certain expectations of privacy. Red teams have a “Rules of Engagement document that states what we can and can't do. We've had that spelled out for us so we don't get into trouble” (Clara). Dean adds that these rules “require them to undergo additional background checks. There are additional security controls in place and they tell you this before coming on board. If you are going to do this kind of work, you need to be okay with no privacy.” For instance, the activities of software security engineers are subject to additional logging and monitoring.

In contrast to malicious actors, red teams emulating “root and loot” (Chase) scenarios are “not going to touch customer data” (Cliff). Instead, “they’ll explore and think about what the solutions might be to protect that data” (Cliff). Clara notes, “sometimes we'll work the service owner and we'll show them how we can get into an exploit situation. Then we've proven the point that it could be done. We don't actually have to push the detonator button and blow up something.”
Software security engineers come from diverse backgrounds, and their interest and motivation in security engineering is equally varied. The effectiveness of red teams can be attributed to their ability to realistically emulate a malicious adversary—subject to some reasonable constraints that depend on the specific campaign. Red teams take seriously the ethical responsibilities of white hat hacking, and apply their skills to improve security in their organizations.

### 3.3.3 Software Engineering vs. Security Software Engineering

To characterize the role of security software engineers and better understand how they relate to the software engineers (like those we studied in Chapter 2) we describe the similarities and differences between these two groups. We compare how the impact of their work is measured (Section 3.3.3.1) and discuss a defining characteristic of red teams, the security mindset (Section 3.3.3.2).

#### 3.3.3.1 Organizational Impact

**Similarities.** On the surface, software engineering and software security engineering, as Cliff says, are “both development in one way or another.” Brian explains, “you need to have a background as a software engineer to understand what is involved doing in security processes and assessments.” Paul adds, “even as a software security engineer, I do some software engineering, especially around improving [security features in client applications.]” and Pam explains that both roles “still need to know how to dive deep into things and poke around.”
Differences. But when looking more closely at the two roles, there are substantial differences between software security engineers and engineers. In security software development, you might write tools that have an offensive purpose but it’s usually a quick script and you hack it together. You really don’t have the same practices, formality, foundation, and stability to build on [as in software engineering].” Derek explains, “as a software engineer, you have to have good coding standards, and stick to specific timelines. If something is delayed, the product won’t ship, and that attracts attention.”

In contrast, tools developed by software security red teams “aren’t shared broadly, because mostly these tools start as a collection of scripts. You don’t really care if you run the software and it crashes on you. The tools work well for our team, but not for everyone else. Since the kinds of tools that we write can be dangerous, we try to limit the audience of who can access those tools.”

Making an impact. As a result of these differences, it can be tricky to measure impact if software security engineers are evaluated directly against software engineers. For instance, for software engineers “you could measure how many check-in developers do or how many features they developed. But if you are a software security engineer reviewing a very secure piece of software, you might not produce anything. Sometimes when we don’t find any vulnerabilities, we explain what we tried to do—how we tested this feature in this particular way or tried to reach this network. Across the industry, it’s difficult to measure our output if you don’t know the person who is doing the job” (Brian). Paul explains two other approaches to quantifying impact. First, one way is to “find critical vulnerabilities that would have been serious if discovered externally.” The second approach is to “take a deep dive into a feature—you may not necessarily find any critical bugs, but you instead make a bunch of recommendations to actively improve the security.”
Although software security engineers “are an umbrella term for people who probably aren’t doing a lot of development—especially on features or products that you’d give to other people—I’d say the security moniker gives you leg room to make impact in other ways” (Eric). Bill concludes that “it’s incorrect for me to call myself a software engineer when I haven’t specialized in software engineering. These need to be distinct roles.”

### 3.3.3.2 Security Mindset

**Defining the mindset.** Security is a mindset and it’s a way of thinking about the world that is necessary for effective security engineering [Sev16]. But, as Pat explains, “I wouldn’t say there’s any one process. There’s several different ones, and it depends on what I’m trying to achieve.” For instance, Pat sometimes thinks about the security mindset as “recognizing how something can be used maliciously or broken” and that “it tends to be a blind spot among many people.” Pat explains, “It’s the difference between seeing something as a tool versus a weapon I suppose. There is a mantra among some people in the security space that everything is broken.”

**Recognizing risk.** Another aspect of the security mindset “is recognizing when it is the case that is someone is using something that they don’t understand how it works. The average developer has no understanding of how dangerous it is. There a lot of things out there that are very dangerous to do that are not advertised as this. People don’t have any awareness of the risk their undertaking by using it. keeping aware of that sort of risk is part of being a software security engineer” (Pat). Chase clarifies, “what that means is that there are always ways to break into or exploit software. So to be successful you need to have an exploitation mindset and having a broad understanding of the world and looking at possible vectors.” As David points out, “when you are a software security engineer, the question you always have in mind is, what if? I’m not saying developers write bad code,
because when you develop something you're goal is to make it work. You don't think about the possible edge cases.”

Thinking like an adversary. One way of thinking about possible vectors is through “have the willingness and motivation just to keep looking for new things. Just because something isn't an issue today, doesn't mean it's not going to further down the line. Some security engineers are so ingenious with all the things they do it surprises me. I would have thought to do it but it's really cool how they thought to poke a hole in something.”

When comparing the mindset between software security engineers and software engineering, Paul explains that the “the main difference is going to be having the right mindset for security—having an adversary's mindset—and always thinking about how things are going to break instead of how things are going to perform correctly.” Pat reflects, “there's a way of thinking and looking at things that's important for it that not everyone quite manages.”

<table>
<thead>
<tr>
<th>Software vs. Software Security Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Although software security engineers conduct software development activities, they have several consequential differences in how they write, maintain, and distribute software. For example, software security engineers limit the distribution of their tools because of their exploitation capabilities. Software security engineers have unique specializations such that they should not be evaluated directly as software engineers. A notable difference between software security engineers and software engineers is the security mindset.</td>
</tr>
</tbody>
</table>

3.4 Threats to Validity

Using Maxwell [Max92], we discuss three practical dimensions of validity that routinely arise in qualitative research: descriptive validity, interpretive validity, and generalizability.
**Descriptive validity.** It is possible that informants may misremember or unintentionally distort their account of a particular theme. We adopted a contextual inquiry approach from Lutters & Seaman [Lut07], and asked informants to provide specific accounts of recent experiences to mitigate against informants’ natural tendencies to generalize. We asked multiple informants about the same theme and searched for disconfirming evidence where statements from one informant would contradict the statements from another. We also had *prolonged engagements* over a period of six months, during which the researchers built trust with various red teams by collaborating with team leads (“gatekeepers”), and establishing rapport with the informants so that they were comfortable disclosing information. Finally, we provided assurances that the informant feedback would remain anonymous and confidential information would be scrubbed from any final reports.

**Interpretative validity.** This dimension concerns a misinterpretation in how the researchers understand informants’ statements during thematic analysis, and that the resulting themes are accurate. To reduce misinterpretation, we used *thick description* [Pon06], that is, grounding our findings by relying on the informants’ own words whenever possible. We also conducted interviews in pairs, and immediately after each discussed whether our interpretations about the interview were consistent. In cases of remaining uncertainty, we simply went back to the informant using online chat and asked for clarification—a form of lightweight member checking [Koe13].

**Generalizability.** Semi-structured interviews do not offer external validity in the traditional sense of nomothetic, sample-to-population, or statistical generalization. In place of statistical generalization, our qualitative findings support an idiographic means of satisfying external validity: *analytic generalization* [Pol10]. In analytic generalization, we generalize from informant conversations to themes.
We conducted our study at a single company, and thus the results may not generalize to other organizations. However, Kotulic & Clark [Kot04] found security research to be one of the most intrusive types of organization research and that “there is undoubtedly a general mistrust of any ‘outsider’ attempting to gain data” about an organization’s security practices. This makes it difficult to obtain access to informants. We did recruit from five different red teams within Microsoft, who operate autonomously. Several informants also had prior red team or security experience before joining Microsoft, and could contrast their prior experiences with their current role.

3.5 Discussion

We discuss the significance of our findings on software security engineers in red teams, our interpretations of these findings, and the contributions of this study within the broader context of software security research and practices.

3.5.1 Organizationally Supporting Red Teams

Red teams thrive in a security-aware organizational culture. For example, for our informants, static and dynamic analysis are already part of the continuous integration pipeline, and “software engineers are are not allowed to build unless you have these security tools in their build definitions” (Peter). Thus, most of the issues detected by these approaches are already addressed by software developers before getting to the software security engineers. This allows red teams to investigate more sophisticated attack scenarios. This approach works because developers are willing to take on additional security responsibilities [Hil17].

Ultimately, as Da Veiga & Eloff [Da 10] finds, an “organization’s success or failure effectively depends on the things that its employees do or fail to do.” When red teams function
within a security-aware organization, software security engineers are able to expedite their security campaigns. For instance, red teams at Microsoft typically skip initial access tactics such as spearphishing, a tactic often used by malicious adversaries to gain a foothold within a network. This step can be reasonably skipped in a campaign because employees already understand the importance of maintaining their credentials securely.

In addition to conducting offensive campaigns (Section 3.3.1.1), red teams at Microsoft offer a number of pathways (Section 3.3.1.3) for software engineers to engage more closely with the red teams. One pathway is design reviews, where “a red team will meet with a partner before implementation to identify early design problems in the architecture and connections between different components” (Bill). Doing design reviews helps catch potential security problems earlier in the process, when the issue can be more easily corrected. A second pathway is ride-alongs, where one or more software engineers joins the red team to participate in a security campaign against their own product or service. These pathways propagate security knowledge throughout the organization, and enable software engineers to take a more active security role.

### 3.5.2 Designing Tools for Software Security Engineering

Informants expressed both a desire for scripting capabilities in tools to enable automation, and for tools that could synthesize source code and binary artifacts to construct explanations.

**Recommendation I—Tools should support scripting or extensions to allow for automation.** As we show in Section 3.3.1.2, the context-sensitive nature of software security requires software security engineers to write their own ad-hoc tools and scripts while conducting security campaigns. In addition, many informants \((n = 11)\) expressed a desire to
initially explore manually with their tools, but automate their strategies if that manual ex-
ploration was productive. Informants felt some tools supported extensibility well, including
Wireshark [Wir] and IDA Pro [Ida]; however, one criticism of many scripting extensions was
that they required learning a proprietary language specific to that tool (Pat, Cliff, Eric). This
suggestions that tools support extensibility, but also use a common language for extension,
such as Python.

**Recommendation II—Tools should support both comprehension of code and expla-
nation of code.** The tools that software security engineers use are similar to the tools
software engineers use, but they use them in different ways. Rather than using integrated
development environments and debugging tools for writing code, our informants primarily
used these tools to help understand code written by others. Informants reported that code
comprehension was mostly a manual process, and had available to them very few tools
to assist, other than basic navigation. Instead, informants had to write ad-hoc scripts to
overcome these limitations. For example, Evan built a plugin that improved highlighting by
allowing him to highlight multiple variables at once.

As we discuss in Section 3.3.1.3, red teams have to communicate their findings to
software engineers, often through reports. To generate these reports, informants manually
collected code from multiple different projects and then stitched the relevant code snippets
together to construct logical explanations, interleaving their own commentary between
the code snippets. In some cases, they would rely on text or ASCII diagrams as part of
their explanation. For red teams, program comprehension and program explanation are
first-class activities, and proper tools need to be developed to support them.
3.5.3 How Practitioners Perceive Security Research

Several informants (Clara, Dean, Dan) described how they or their team learn new skills and share knowledge with other security practitioners through attending industry conferences, reading online blogs and news sources, and attending meetups (Section 3.3.2). While previous work suggests that software practitioners find academic research papers to be essential or worthwhile [Lo15], and a replication study in empirical software engineering found similarly results [Car16], few informants reported using academic research as something that they apply to their security practices. Dan explains that, “I’ve read some academic work and they’ve given me a lot of ideas—but I haven’t actually been able to apply them in the real world.” Some of the barriers, Dan observes that “the papers are taking in a lot of new technologies that I’m not super familiar with, and frankly, the academic literature is always written with these weird Greek symbols that make no sense to me.”

Practitioners don’t seem to be benefiting, at least directly, from the academic literature. However, the opposite is also true: industry typically does not provide researchers with access to sensitive materials such as incident-response data, post-infection actions taken by malicious adversaries, or historical samples that are often necessary to conduct threat protection research [Lem18]. We hope that studies such as this one will help to strengthen the relationship between industry and research.

3.5.4 Hacking Tools as Offensive Weapons and Defensive Tools

Software security tools have a somewhat complicated relationship with engineers, due to their dual nature: they can be used as both offensive weapons, such as by an adversary—a sword, or a defensive tool to harden security—a shield [Rad]. As we found in Section 3.3.3.1,
this causes subtle but important difference in how software security engineers are able to write and publish tools.

Unlike traditional software products, software security engineers are mindful about how their tools might be used by others. In particular, some people are not warm to the idea of having another person literally hacking their systems [Lav15]. For these reasons, security tools are often not shared broadly, or made visible to other teams even within the organization. As a result, the tools that software security engineers use don’t receive as much investment, and are therefore less mature in terms of features and stability.

Previous work by Witschey et al. [Wit15] identified reasons for using or not using security tools. We found an additional barrier to tool adoption not reported by Witschey et al. [Wit15] for software security engineers. Specifically, our informants reported not using third-party tools because security engineers “aren’t confident there isn’t a backdoor unless the tool is verified. We don’t trust security tools from the internet unless you control the environment and have reviewed the code” (Brian) This makes it difficult and expensive to introduce new security tools into the team. Paradoxically, software security engineers see the introduction of software security tools as potentially reducing the security of their systems.

3.5.5 Recognizing the Impact of Software Security Engineering

Although software security engineers within red teams play a critical role in hardening software applications and services within their organization, their impact is often unacknowledged due to the clandestine nature of their activities (Section 3.3.3.1). For example, this makes it difficult for engineers in red teams to get visibility outside of their own organizations, because they can’t publicize their efforts—either through open source projects or conferences.
Red team engagements are particularly important in cloud service organizations, because these environments host not only the organizations’ services but also those of third-party tenants. Cloud computing services have become an alluring target for these adversaries because they are readily available, offer inexpensive but substantial compute power, provide anonymity via blocks of third-party IP addresses, and allow lateral movement across concentrated assets [Sin16].

Even organizations without red teams can benefit from the positive externalities of red team activities. For example, open source projects benefit because security vulnerabilities found by Microsoft red teams are responsibly disclosed to the open source project maintainers. Red teams will sometimes “engage in part of the development process with other security teams, even if they’re part of a different organization” (Dan).

Although software security engineers have some overlapping responsibilities with security engineers, the activities (Section 3.3.1.2) they conduct—such as reverse engineering and exploit hunting—are disparate and specialized such that that organizations need a distinct security discipline to fully appreciate and recognize their impact.

### 3.6 Conclusion

To understand the emerging role of red teams, we conducted an interview study with 17 informants to learn about their workflow, their security culture, and clarify how software security engineers are distinct from software engineers. Through these interviews, we found that red teams undertake campaigns with goals and targets that emulate the activities of sophisticated actors. To achieve their goals, software security engineers must specialize and maintain the ability to pivot quickly. We found that software security engineers challenge the stereotypes often applied to hackers; the software security engineers we interviewed come
from diverse backgrounds. In contrast to malicious adversaries, red teams take seriously the ethical responsibilities of white hat hacking. Although software security engineers have some overlapping responsibilities with software security engineers, a key difference is in how they apply a security mindset—thinking like an adversary.
4.1 Introduction

In Chapter 2, we observed developers using a single security-oriented static analysis tool, FSB. Here, we broaden in scope to examine tools other than FSB. Additionally, this chapter demonstrates how we can use the findings from Chapter 2 to identify usability issues in practice. In this chapter, we answer RQ4 (How do existing static analysis tools support developers' information needs and strategies?) by examining how characteristics of existing analysis tools contribute to and detract from the vulnerability resolution process.

To that end, we evaluate the usability of four security-oriented static analysis tools, Find Security Bugs [Fsb], RIPS [Rip], Flawfinder [Fla], and a commercial tool. (Our license

I’d like to thank Lisa Nguyen Quang Do for contributing to the study presented in this chapter.)
agreement with the tool vendor stipulates that we anonymize the commercial tool; we refer to it as CTool.) Similar evaluations have been successfully conducted in the adjacent field of end-user security [Whi99; Goo03; Mar02; Cla07; Gal17]. For instance, in their foundational work, Whitten and Tygar conducted a cognitive walkthrough evaluation to identify usability issues in an end user security tool (PGP) [Whi99]. To study usability issues in developers’ security-oriented static analysis tools, we borrowed a similar technique, namely heuristic walkthroughs [Sea97],

As a result, we have identified several usability issues, ranging from missing affordances to interfaces that scale poorly. Each of these usability issues represents a concrete opportunity to improve developers’ security tools. Alongside these usability issues, we contribute our visions for how toolsmiths and security researchers can improve the usability of static analysis tools for security. Ultimately, by improving the usability of security-oriented static analysis tools, we can enable developers to create secure software by resolving vulnerabilities more accurately and efficiently.

We make the following contributions in this chapter:

- A categorized list of usability issues that serves both as a list of known pitfalls and as a list of opportunities to improve existing tools. (Section 4.4)

- Design suggestions and discussion that demonstrates the actionability of this list. (Under Discussion throughout Section 4.4)

- Specifications for a heuristic walkthrough approach that researchers and practitioners can use to improve additional tools. (Section 4.3.3)
4.2 Description of Selected Tools

4.2.1 Tool Selection Process

We evaluated four security-oriented static analysis tools, Find Security Bugs (FSB), RIPS, Flawfinder, and CTool. In this section we justify our choice of those four tools and describe those tools, focusing particularly on how they present information to developers.

Organizations and researchers have compiled lists containing dozens of different security-oriented static analysis tools [Sec; Owab; Codb; Whe]. Combining these lists, we considered 61 candidate tools. Because there are so many unique tools, it would be intractable for us to evaluate all types of tool interfaces. To narrow the selection of tools to use for our evaluation, we followed two criteria.

The first, and most straightforward, selection criteria was availability. We only considered tools we could access and run. This criteria limited our selection of commercial tools, because their license agreements often explicitly forbid the publication of evaluations or comparisons. (For example, Coverity’s license agreement states, “Customer will not disclose to any third party any comparison of the results of operation of Synopsys’ Licensed Products with other products.”) Therefore, via professional connections, we requested explicit permission to evaluate six candidate commercial tools. Although most tool vendors were interested in the study in principle, only one agreed to participate. We refer to this tool as CTool. The vendors who decided not to participate in our study cited lengthy approval processes or legal restrictions as reasons not to have their tools evaluated. After three months of negotiation, the makers of CTool agreed to provide us with an evaluation license of their tool. That license allowed us to conduct this evaluation, but stipulated that we

---

anonymize the tool by not using screenshots, names, trademarks, or other distinguishing features in any publication.

Second, we faced the potential threat that the tools we selected would not represent the real-world diversity of tool interfaces. For instance, Lint, Jlint [Jli], and Pylint [Pyl] are three distinct tools, but they share very similar command-line interfaces. To increase the generalizability of our results, we chose four tools that cover different aspects of the tool interface design space. This primarily translates to selecting tools with four different modes of interaction: command-line interface, IDE integration, and two standalone tools. Table 4.1 summarizes how FSB, RIPS, Flawfinder, and CTool vary along some of the interface dimensions we considered. (Note: Table 4.1 reflects the interfaces of the versions of the tools we chose to evaluate. For instance, FSB can also be run as a command-line tool.) The full table of tools and interface dimensions is available online [Usd].

Though definitive usage statistics are hard to find for these tools, it's fair to characterize all four of these tools as widely-used. According statistics published by Sourceforge,² FindBugs,³ RIPS, and Flawfinder have approximate download counts of: 1,410,000, 143,000, and 19,000 respectively. CTool has been adopted by government agencies and hundreds of companies across different industries.

In the remainder of this section we describe each of the four tools we evaluated.

### 4.2.2 Find Security Bugs

Find Security Bugs (FSB) is an extension of the SpotBugs [Spo] static analysis tool that detects bugs in Java software maintained by Philippe Arteau [Fsb]. Find Security Bugs focuses on 125 different security vulnerability types described in the CWE [Cwe] and OWASP top 10 [Owac]

---
²https://sourceforge.net/
³Download statistics aren't available specifically for FSB, which is a plugin for FindBugs
Table 4.1 Evaluated tool interface dimensions

<table>
<thead>
<tr>
<th>Dimension</th>
<th>FSB</th>
<th>RIPS</th>
<th>Flawfinder</th>
<th>CTool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remediation Information(^1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Trace Navigation(^2)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Quick Fixes(^3)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphical Rep. of Traces(^4)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Command-Line Interface</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDE Integration</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standalone Interface</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

\(^1\) Information that helps developers fix a vulnerability;  
\(^2\) Affordances that allow developers to trace dataflow;  
\(^3\) Features for applying patches automatically;  
\(^4\) Graphical representations of dataflow traces;

by detecting dangerous API signatures in Java programs. Like SpotBugs, Find Security Bugs is an open-source IDE plugin. We used the Eclipse [Ecl] plugin, version 1.7.1. Although no data on the users of Find Security Bugs is available, the GitHub repository has been forked 221 times, and it is listed in recommended reputable lists of static analysis tools such as the OWASP's [Owab].

Figure 4.1 presents the GUI of the Find Security Bugs plugin, running on the source code of Apache POI version 3.9 [Apa]. The list of vulnerabilities found by the tool is presented on the left view, under categories that indicate the severity of the errors: “Scary”, “Troubling” and “Of Concern”. In each category, the errors are grouped in sub-categories of confidence, indicating how certain the tool is that the error is a true positive. Finally, in each sub-category, the errors are grouped by vulnerability type (e.g., “Cipher with no integrity”). When double-clicking on an error, the tool highlights the line of code that causes the error in the editor. The left gutter of the editor shows bug icons to help users locate the errors. Tooltips for those icons provide information on all errors occurring at that particular line.
The bottom view provides more information about the vulnerability that is being examined. It contains a bug description, examples of similarly vulnerable code and how to fix it, links to useful information on this vulnerability, and tool-specific information. It also provides a “Navigation” panel that contains a step-by-step trace of the vulnerability. For example for an SQL injection, it reports a witness from the source to the sink.

The tool can be manually re-run by right-clicking on the project. It re-scans the entire project every time it is re-run. Aside from the list of errors shown in the IDE, it is possible to export a report of all bugs found in XML format.

With the configuration settings shown in Figure 4.2, it is possible to customize how the list of results is shown in the left view. Bug patterns and categories can be toggled in and out, in which case, errors matching the category will no longer show in the list. The different types of vulnerabilities can also be reclassified in different severity categories. It is also possible to include and exclude particular source files from the scan, and to choose
which detectors are run by Find Security Bugs (e.g., the AndroidSqlInjectionDetector, or the AtomicityProblem detector).

### 4.2.2.1 RIPS

RIPS [Rip] is a security static analysis tool maintained by RIPS Technologies. It is applicable to PHP code and can find more than 80 vulnerabilities described in the CWE/SANS top 25 and the OWASP top 10. It provides a standalone web interface from which the user can configure and launch scans, and consult the results. We used version 0.55 of RIPS. It is
Figure 4.3 RIPS’ exploit creator view.

recommended on the OWASP list of static analysis tools [Owab]. The open-source version is also used by CERN and recommended on its list of analysis tools [Cer].

Figure 4.4 presents the main screen of RIPS when run on version 2.0 of the Word-press [Wor] source code. Upon finishing a scan, a pop-up presents a summary of the results with statistical data on the files scanned and the numbers of errors found. The errors are grouped by files, and in each file, ordered by vulnerability type. Each individual error is documented with a short description of the vulnerability and a code snippet in which it occurs. An icon on the left (not pictured) opens a code viewer view showing a non-editable version of the file containing the error. Sometimes, a “help” icon and a “generate exploit” icon are also shown on the left. The help icon opens a view in which the vulnerability is explained in more detail, with a code example and its fix. The generate exploit icon opens the view
in Figure 4.3 in which the user can generate an example exploit for this vulnerability. The
description box also contains information on other files in which the same error occurs.

On the top menu of the page, as shown in Figure 4.4, the user has access to additional
views: The “stats” button opens the scan summary. The “user input” button shows the list
of variables that may contain user-influenced inputs (e.g., $_GET[error]). The “functions”
button opens a view in which the user can see the list of functions contained in the scanned
files, and a call graph illustrating which functions call each other. The user can edit certain
parts of the graph: rearrange the node layout, modify the function names, and edit the
edges between the different functions. The “files” button provides the same functionalities
as the “functions” button, on the level of the files.

The tool can be re-run by clicking on the “scan” button. RIPS scans the files for which
the path in the file system is provided. The user can thus restrict which files have to be
scanned. The user can set the verbosity level of the tool, which will make the scan return
more or fewer results, and they can select what kind of vulnerabilities RIPS should search
for. It is also possible to filter which results are shown by using the “hide all” buttons which
hide all errors found in a particular file, or searching for a particular regex pattern. The style
of the errors (font, color, background color, etc.) can also be customized.

4.2.3 Flawfinder

Flawfinder [Fla] is a static analysis tool maintained by David Wheeler that detects uses of
dangerous functions in C/C++ code as described in the CWE/SANS top 25 [San]. This open-
source tool is run using the command-line. We used version 2.0.4. Flawfinder was used for
multiple assessments of analysis tools [Che04; You04; Cow03]. The tool also appears in the
OWASP [Owab] and CERN [Cer] lists of recommended static analysis tools.
Figure 4.4 The Graphical User Interface of RIPS.

Figure 4.5 illustrates an HTML report produced by Flawfinder on the OpenSSL source code, version 1.0.1e [Ope]. On the top of the report, Flawfinder provides information on the tool and the ruleset it uses to find the vulnerabilities. It then lists all files that were scanned, and all errors it found, ordered by severity. For each error, it provides the location of the error (file name and line number), the severity score, the vulnerability type, and the dangerous function that is used in the code. A short description of the vulnerability is also given, to explain why the function is dangerous, along with a link to the CWE page of the vulnerability. A fix is then proposed, which is often the name of a safe function that can be used instead of the vulnerable one. The bottom of the report shows the analysis summary, which contains statistical data about the scan, such as with the number of files scanned, the number of errors reported, etc. The tool either prints its report in the command-line or produces reports in HTML or CSV format.
Flawfinder Results

Here are the security scan results from Flawfinder version 2.0.4. (C) 2001-2017 David A. Wheeler. Number of rules (primarily dangerous function names) in C/C++ ruleset: 223
Examing openssl-1.0.1e/crypto/dso/dso_null.c
Examing openssl-1.0.1e/crypto/dso/dso_beos.c
Examing openssl-1.0.1e/crypto/dso/dso.h
...

Final Results

* openssl-1.0.1e/crypto/rand/randfile.c:86: [3] (race) chmod: This accepts filename arguments; if an attacker can move those files, a race condition results. (CWE-362). Use fchmod() instead.
  #define chmod _chmod

...

Analysis Summary

Hits = 110
Lines analyzed = 403801 in approximately 11.93 seconds (33845 lines/second)
Physical Source Lines of Code (SLOC) = 278715
Symlinks skipped = 108 (–allowlink overrides but see doc for security issue)
Minimum risk level = 4
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See ‘Secure Programming HOWTO’ (https://www.dwheeler.com/secure-programs) for more information.

Figure 4.5 The Graphical User Interface of Flawfinder.

It is possible to restrict which files are scanned by Flawfinder using command-line options to include or exclude symbolic links. It can also only consider patch files, e.g., the diffs between two git commits. Errors can also be filtered out of the output based on regex patterns, risk level, likelihood of being a false positive, and whether or not the dangerous function gets input data from outside of the program.

4.2.4 CTool

CTool is a commercial tool that is largely used in industry to scan C, C++, and Java code and bytecode. It is able to detect a large range of software defects ranging from simple bugs to complex security vulnerabilities. The tool can be run from the command-line or through a full application. It also provides different interfaces such as an IDE plugin, or a web page.
In this study, we use the web interface to scan the binaries of Apache POI version 3.9 [Apa]. The creators of CTool state it is used by hundreds of companies around the world.

The web interface of CTool has two main views, which we detail on a high level to keep the tool’s anonymity. The first view is an overview of the warnings found in the project. Each warning is reported along with a priority score, the warning type, the code location, and information on its severity. The second view details one warning in particular. This second view includes: warning details; the source code snippets; information about how particular lines of the source code contribute to the warning; and sometimes fix suggestions. In this view the user can comment on the warning, and edit its priority for example. The two main views of CTool are supported by a large number of visuals. In particular diverse charts and graphs, and a complex navigation system are used to help the user get an overview of all warnings in the code, or one particular warning. The GUI cannot be customized, since it is a web page, but the diversity of the visuals and the navigation capabilities cover a large number of potential use cases developers would run into.

The tool provides the ability to export a report in xml, html, or pdf format, and to customize the report. It also allows the users to customize the analysis on multiple levels: different checkers can be activated or deactivated, custom checkers can be plugged-in and run by the tool, and code annotations allow developers to guide the analysis at runtime. Users can also annotate warnings and then track scores throughout the development lifecycle across builds.
4.3 Methodology

4.3.1 Study Environment

To ensure the evaluators could exercise the tools in a variety of situations, we chose subject applications that contained multiple types of vulnerability patterns. Since each of the four tools can detect many different types of defects, we selected test projects that covered a variety of these vulnerability patterns. Synthetic suites, like the Juliet test suite [Bo12], could have helped us ensure this coverage. However, they do not necessarily represent how vulnerabilities appear to developers in the wild. For instance, vulnerabilities in the Juliet suite are pre-labeled as false positives or true positives and generally strip away any code not relevant to triggering the vulnerability.

Therefore, while selecting test suites, we were also concerned with the ecological validity of our findings. We selected test suites from the “Applications” section of the Software Assurance Reference Dataset [Nis], because those test suites are derived from production code bases containing known vulnerabilities.

Since these tools scan code written in different languages, we selected three test suites that satisfied the constraints described above: RIPS scanned WordPress version 2.0 (PHP); FSB and CTool scanned Apache POI version 3.9 (Java); and Flawfinder scanned OpenSSL version 1.0.1e (C). All three open-source tools and their associated applications were configured in a single virtual machine image for evaluation. The virtual machine is available online [Usv].
4.3.2 Heuristic Walkthroughs

We identified usability issues in the security tools using heuristic walkthroughs [Sea97], a two-phase method that combines the strengths of two usability evaluation techniques: cognitive walkthroughs [Pol92] and heuristic evaluations [Nie90]. In a cognitive walkthrough, evaluators simulate the tasks that real users would perform with a system. In a heuristic evaluation, evaluators systematically examine a system following a set of heuristics (as opposed to the task-driven approach in a cognitive walkthrough). In a heuristic walkthrough, evaluators first perform a cognitive walkthrough and then perform a heuristic evaluation. Combining the strengths of these two techniques in this way, heuristic walkthroughs have been shown to be more thorough than cognitive walkthroughs and more valid than heuristic evaluations on their own [Sea97].

We chose this evaluation technique, because heuristic walkthrough evaluations have successfully identified usability issues in other domains, such as electronic health records systems [Edw08], music transcription tools [Bur17], and lifelong learning systems [Gu11]. Compared with traditional laboratory and field studies, this technique does not require the onerous recruitment of participants with security expertise. Researchers have noted that recruiting sufficiently many participants for security-focused studies can be prohibitively difficult [Aca17]. One reason, for instance, is that security research is one of the most intrusive types of organizational research [Kot04]. Individuals and organizations may be reluctant to openly discuss their security practices, for fear of revealing weaknesses in their security practices.

Addressing this problem, heuristic walkthroughs enable relatively few external evaluators to identify usability issues by providing evaluators with a structured process and requiring the evaluators to be double experts (experts in the application domain and usabil-
ity principles). In compliance with these requirements, the two evaluators who conducted the heuristic walkthroughs (myself and another researcher) for all four tools were familiar with security tools and usability principles. Our evaluation draws on our experience building advanced static analysis tools, conducting laboratory experiments, and graduate-level courses in security, software design, and user experience. This experience provided us familiarity with the application domain (security tools) and familiarity with the evaluation technique (the heuristic walkthrough method). Not including the time required to configure each tool, each evaluator spent approximately one workday with each tool. The study protocol is available in Appendix C and the detailed list of usability issues [Usi] are available online.

4.3.2.1 Phase 1: Task-Oriented Evaluation

Phase 1 of a heuristic walkthrough resembles a cognitive walkthrough, where evaluators approach a system with a list of predefined tasks. The goals of this phase are for evaluators to familiarize themselves with the system and complete tasks similar to those actual users would try to complete. In Phase 1 of our study, we used the tools with a particular task in mind: fixing as many errors as possible in a limited time, as a security auditor at a software company. To do so, we used the following guidelines:

- Choose a vulnerability that you’d be most likely to inspect first.

- Determine whether the reported vulnerability is actually a vulnerability.

- Propose a fix to the vulnerability.

- Assess the quality of your fix.
To help us think critically about each tool, we used the list of guiding questions presented by Sears for heuristic walkthroughs [Sea97]. These questions ask evaluators to consider whether users will: know what to do next; notice the correct controls; know how to use the controls; see progress being made. During Phase 1, we recorded the vulnerabilities we inspected, our impressions of the tool, and any usability issues we encountered.

4.3.2.2 Phase 2: Free-Form Evaluation

Phase 2 of a heuristic walkthrough resembles a heuristic evaluation, where evaluators freely explore an entire system using a set of usability heuristics to identify issues. After completing Phase 1, evaluators have familiarized themselves with the system and, thus, are prepared to conduct a heuristic evaluation.

As opposed to using generic usability heuristics, like Nielsen's ten heuristics [Nie95] or the cognitive dimensions framework [Gre89], we used the 17 categories of information needs from Chapter 2 as the basis for our heuristics. Prior studies have shown that domain-specific heuristics can be more effective [Jaf14; Dyk93; Pat12].

The two evaluators considered each of the 17 information needs and recorded any usability issues that related to those information needs. Of course, during this phase, the evaluators also recorded additional usability issues that did not precisely fit any of the provided heuristics.

Finally, because similar issues were identified across tools, heuristic categories, and evaluators, we performed an informal thematic analysis to group usability issues into themes and subthemes. This analysis is only intended to reduce repetition and clarify the presentation of the results. Section 4.4 is organized according to these themes; each subsection describes one theme.
4.3.3 Replication Package

To support the replication and continuation of this work, we have made our materials available, including the virtual machine containing the static analysis tools (we exclude CTool for legal reasons) and the code bases they were used on [Usv], and the full list of usability issues we detail in the following section [Usi]. The evaluation guide is available in Appendix C.

Our replication package includes the virtual machine image used during our heuristic evaluation. This image contains FSB, RIPS, and Flawfinder configured to run on their respective codebases, and instructions describing how to get initial scan results. CTool is not included, because our license agreement does not permit us to disclose information that would deanonymize the tool.

Because heuristic walkthrough evaluations of security tools are beneficial beyond the scope of what was feasible during our study, we also provide the heuristic evaluation guide we developed. With this guide, a qualified evaluator with expertise in static analysis tools and usability principles could extend our work to any additional static analysis tool for security.

4.4 Results

Our heuristic walkthrough evaluation identified 194 distinct usability issues. We do not intend for the presence or quantity of these issues to be a statement about the overall quality of the tools we evaluated. Instead, each of these issues represents a potential opportunity for tools to be improved.
In each section, we will give a general description of the usability issues in that theme, explain how instances of those issues impact developers, and sketch how our insights could be used by tool designers and researchers to improve security-oriented static analysis tools. Next to the title of each theme, we report the number of usability issues in parenthesis (X) that we identified in that theme. This number simply characterizes our findings and should not be interpreted as the ranking or severity of the issues in that theme. Also note that these counts sum to slightly more than 194, because some usability issues span multiple themes.

To further organize the results, we have bolded short titles that describe subthemes of issues within each theme. Next to each subtheme title are the tools, in {braces}, that issues in that subtheme apply to. For instance, “Immutable Code {RIPS, Flawfinder, CTool}” denotes that Immutable Code usability issues apply to RIPS, Flawfinder, and CTool, but not FSB. In addition, Table 4.2 provides an overview of the themes and subthemes.

### 4.4.1 Missing Affordances (39)

Beyond presenting static information about code defects, analysis tools include affordances for performing actions, such as navigating code, organizing results, and applying fixes. Issues in this category arose when tools failed to provide affordances.

**Managing Vulnerabilities {FSB, RIPS, Flawfinder}:** After scanning the source code, tools must report the identified vulnerabilities to developers. We found that FSB, RIPS, and Flawfinder did not provide adequate affordances for helping developers navigate and manage the list of reported vulnerabilities. Managing the list of reported vulnerabilities is important, because it allows developers to quickly find the vulnerabilities they would like to inspect and fix. For instance, some developers might only be interested in a subset of the code, or have expertise fixing particular types of vulnerabilities. These three tools simply
Table 4.2 Usability issues, grouped by theme

<table>
<thead>
<tr>
<th>Theme and Subtheme</th>
<th>FSB</th>
<th>RIPS</th>
<th>FlawFinder</th>
<th>CTool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 4.4.1 Missing Affordances</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Managing Vulnerabilities</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Applying Fixes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Section 4.4.2 Missing or Buried Information</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vulnerability Prioritization</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fix Information</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Section 4.4.3 Scalability of Interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vulnerability Sorting</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Overlapping Vulnerabilities</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scalable Visualizations</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Section 4.4.4 Inaccuracy of Analysis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Section 4.4.5 Code Disconnect</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mismatched Examples</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Immutable Code</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Section 4.4.6 Workflow Continuity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracking Progress</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Batch Processing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

show a list of all vulnerabilities they found; the options to manage the list of all potential vulnerabilities were limited.

Flawfinder, for example can generate a single text file, csv, or HTML page containing all the scan results. As Figure 4.5 depicts, in Flawfinder these “Final Results” are presented as a list that cannot be reorganized, searched, or sorted. To find a vulnerability to inspect in detail, a developer must linearly search through the full list of results. Consequently, it is difficult for developers to quickly find and fix vulnerabilities they are interested in.

**Applying Fixes [FSB, RIPS, Flawfinder, CTool]:** The tools we evaluated did not fully support *quick-fixes* (semi-automated code patches for common vulnerabilities) and did not otherwise offer assistance applying changes. Instead, developers must manually fix the issues reported by these tools. Only FSB included some quick-fixes, but this feature
was available for just three out of the 21 defect patterns present in our test suite. Without these affordances for applying fixes, developers must exert extra effort to resolve the defects presented by their tools.

**Discussion:** Many of the affordances that we noted as missing from these tools do not represent revolutionary breakthroughs in user interface design. In fact, features like sorting and filtering lists are commonplace in many applications. Integrating these well-known affordances into static analysis tools for security could be one low-cost way to improve the usability of these tools. On the other hand, some affordances will require more effort to incorporate into analysis tools. For example, affording developers the ability to accurately apply automated patches remains an open research area. We are encouraged by systems like FixBugs [Bar16], which assists developers in applying quick-fixes with FindBugs. Our results suggest that security-oriented static analysis tools would benefit from advances in this area.

### 4.4.2 Missing or Buried Information (96)

Static analysis tools can provide developers with a wide range of information about the defects they detect. For example, all four tools we studied give information about the location and defect-type of the vulnerabilities detected. The issues in this theme correspond to instances where tools failed to provide information that would be used to resolve defects. In this theme we discuss both missing information and buried information. These two issues are intertwined, because buried information that a developer never unearths is effectively missing.

**Vulnerability Prioritization [FSB, RIPS, Flawfinder, CTool]:** Since tools can generate many alerts from a single scan, before fixing a vulnerability, developers must decide which
alert to inspect first. To varying extents, all four tools failed to provide information that, had the information been present, would have helped developers decide which vulnerabilities to inspect. Many of these issues arose as we considered the “Vulnerability Severity and Rank” heuristic during Phase 2. We noted several different types of missing information, such as information about: which files contained clusters of vulnerabilities (Flawfinder); a vulnerability’s severity (RIPS); and how to interpret severity scales (FSB, Flawfinder CTool).

For example, unlike RIPS, FSB provides information about the severity of each vulnerability, typically in the following form:

```
Rank: Of Concern (18), confidence: Normal
```

However, even FSB does not provide information about how to interpret this report. A developer might be left wondering whether 18 is high or low, or what other confidence values are possible. This issue may disproportionately affect users who are using a tool for the first time and still learning to interpret the scales. Nonetheless, lacking information about how to prioritize vulnerabilities, developers might misallocate their limited time by fixing low-severity vulnerabilities.

**Fix Information {FSB, RIPS, Flawfinder, CTool}:** The tools we evaluated also failed to provide some information that developers would need to accurately fix vulnerabilities. The types of missing information spanned many different categories. To name a few, the tools were missing code examples, fix suggestions, definitions of unfamiliar terms, and explanations of how vulnerabilities could be exploited. Furthermore, some types of information that were present were not detailed enough, such as when the tools provided terse issue
descriptions or when the tools listed possible fixes, but did not articulate the tradeoffs between those solutions.

**Discussion:** One solution to these types of issues would be to simply add more information to tool notifications. This simple solution would ensure all the information needed to select and fix vulnerabilities is present for the developer. However, overstuffing notifications with too much information might bury the most pertinent information at a given time. Instead, the challenge for static analysis tools is to discern when developers need a particular piece of information and deliver that information.

### 4.4.3 Scalability of Interface (11)

As static analysis tools scale to find more defects in larger codebases, so too must their interfaces for presenting those defects. The issues in this section arose when tools struggled to present large amounts of information about vulnerabilities. Here we distinguish between scalable interfaces and scalable tools because we are interested in the usability of these tools’ interfaces, not their technical capabilities, which have already been explored elsewhere [Zit04; Rea16; Aus13]. Each of the four tools we examined exhibited an interface scalability issue.

**Vulnerability Sorting [Flawfinder]:** As we previously discussed in Section 4.4.1, Flawfinder does not provide affordances for managing the list of vulnerabilities it detects. This issue is magnified as Flawfinder scales to identify more vulnerabilities in a project. Lacking the ability to manage this list, developers must resort to sub-optimal task selection strategies, such as searching linearly through the list for a vulnerability they would like to inspect.

**Overlapping Vulnerabilities [FSB, CTool]:** Like the other tools we evaluated, FSB and CTool can detect multiple different patterns of vulnerabilities. When multiple vulnerability
patterns are detected on the same line, these tools do not provide clear indications that multiple problems are occurring in the same location. FSB, for example draws multiple bug icons directly on top of each other, which appears just the same as a single bug icon. In fact, Line 117 in Figure 4.1 contains multiple overlapping vulnerabilities, however this is not perceptible without hovering over the bug icon (Figure 4.6). CTool includes a feature for displaying overlapping vulnerabilities, but this feature is turned off by default and is located in a somewhat hidden location.

![Figure 4.6 Instance of Overlapping Vulnerabilities](image)

Scaling up the number of defect detectors increases the likelihood that these instances of overlap will occur. The decision to display vulnerability markers in this way deprives developers of information and forces FSB users to manually inspect each line for duplicates. As a consequence, developers might overlook severe vulnerabilities if their indicators are hidden beneath minor vulnerabilities.

**Scalable Visualizations [RIPS]:** Finally, the clearest scalability issue we encountered was the call graph visualization in RIPS. An example of this visualization is depicted in Figure 4.7. The graph is intended to show the flow of tainted data from sources to sensitive sinks. However, when functions along the flow are called from many sites, all the call sites are added to the graph, resulting in a crowded visualization. Furthermore, when these call sites span more than 50 files, RIPS will not generate any visualization (Figure 4.8).

**Discussion:** We propose two potential design changes that would improve FSB and CTool's overlapping vulnerabilities issues. One possibility is that these tools could use the
Figure 4.7 Scalability of RIPS’ function view.

stack metaphor to present the co-located vulnerabilities. Rather than draw the icons directly on top of each other, the tool could offset each subsequent bug by a few pixels. Alternatively, the tools could annotate a single bug icon with the number of vulnerabilities present on that line (e.g., ![3](image)).

RIPS provides a call graph visualization, whereas the two open-source tools we evaluated provided no similar feature. Such a feature could help visually oriented developers trace vulnerabilities and reason about the flow of tainted data through their system. However, if tools are to implement successful graph visualizations to support developers, the scalability of the visualization must be considered. Tool designers could consider implementing
features such as those in Reacher [Lat], which enable developers to expand and highlight only the relevant paths through the graph.

4.4.4 Inaccuracy of Analysis (17)

The issues in this category arose when we encountered implementation bugs or unexpected behaviors. These issues do not necessarily represent deep design flaws of FSB, RIPS, Flawfinder, and CTool. However, these bugs do have an impact on usability, because implementation bugs may affect developers’ confidence in tools and their abilities to complete tasks. We encountered several issues in this category spanning all four tools; to illustrate the types of issues in this category, here we describe in detail one of these issues affecting FSB.

One set of issues in this category affected FSB, specifically its code navigation features. For instance, when we used FSB’s quick-fix feature, the IDE unexpectedly navigated away from the current file. This behavior was disorienting and could easily cause a developer to lose track of the code they were trying to fix. We also observed issues with FSB’s navigation pane in the bug info window. This pane often either contained duplicated entries, was
missing entries, or contained entries that, when clicked, didn't navigate the user anywhere.

Figure 4.9 depicts an instance of the duplicated entries issue—both entries labeled “Sink method java/io/File.<init> (Ljava/lang/String;)V” refer to the same location in the code.

- **Navigation**

<table>
<thead>
<tr>
<th>java/io/File.&lt;init&gt;(Ljava/lang/String;)V reads a file whose location might be specified by user</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sink method java/io/File.&lt;init&gt;(Ljava/lang/String;)V</td>
</tr>
<tr>
<td>Sink method java/io/File.&lt;init&gt;(Ljava/lang/String;)V</td>
</tr>
<tr>
<td>Method usage with tainted arguments detected</td>
</tr>
<tr>
<td>At RecordGenerator.java:[line 62]</td>
</tr>
</tbody>
</table>

**Figure 4.9** Duplicate entries in FSB's navigation feature

### 4.4.5 Code Disconnect (14)

Static analysis tools generate reports based on the code they scan. However, we identified usability issues when the content of those reports were disconnected from the source code.

**Mismatched Examples [FSB, RIPS, Flawfinder, CTool]:** The first issue in this category relates to the code examples used by all four tools. Many FSB notifications, for instance, contain hard-coded examples of vulnerable code and also examples of suggested code patches. Providing any code example is certainly more helpful than giving no information. Nonetheless, because the examples are hard-coded for each pattern, the burden of figuring out how to adapt and apply that example to the current context falls on the developer. Even if the example is written in the same programming language, this translation can be non-trivial, especially if the example is drawn from a source using different libraries or frameworks. Figure 4.10 depicts one instance where FSB's examples are mismatched with the vulnerable code. In this case, FSB's “solution” example (Figure 4.10(b)) differs
substantially from the original problematic code: the variable names are different; the ciphers are initialized in different modes (encrypt vs. decrypt); and the ciphers are using different encryption algorithms (ECB vs. GCM).

![Private Cipher](image)

(a) Vulnerable code

**Solution:**

```java
private Cipher getCipher() throws GeneralSecurityException{
    byte[] key = generateKey(0);
    Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
    SecretKey skey = new SecretKeySpec(key, "AES");
    cipher.init(Cipher.DECRYPT_MODE, skey);
    return cipher;
}
```

(b) FSB's example “solution”

Figure 4.10 Instance of **Mismatched Examples**

**Immutable Code [RIPS, Flawfinder, CTool]:** We encountered usability issues while trying to apply changes. RIPS, Flawfinder, and CTool’s web UI do not allow developers to directly modify the code they scan. These three tools display a projects’ source code (CTool) or snippets (RIPS, Flawfinder) containing potential vulnerabilities, notify the developers that changes might be required, but do not enable developers to make those changes. Instead, developers must view scan results in one window and edit the code using a separate text editor. This disconnect forces developers to transfer their findings from where the tool presents its results to where they need to make changes. Furthermore, without being able to access the complete original source code while using a static analysis tool, as is the case
with RIPS and Flawfinder, developers cannot use navigation tools to browse code related to a vulnerability. In sum, this workflow is problematic, because developers are burdened with manually maintaining a mapping between the tool’s findings and the code editor. For example, developers must mentally track details about the vulnerable code—like which line, module, and version it is contained in—so that they can apply the fix in the appropriate place.

**Discussion:** Presenting results within an IDE, like FSB and other versions of CTool do, helps developers maintain connections between a vulnerability and the code surrounding that vulnerability. Developers also have the added benefit of being able to use code navigation tools while inspecting a vulnerability within this environment. However, our findings reveal opportunities to establish deeper connections between vulnerable code and how tools present those vulnerabilities to developers. Considering the mismatched examples usability issue, we imagine that analysis tools’ code examples could be parameterized to use similar variable names and methods to those in the scanned code. Achieving a closer alignment between source code and examples will hopefully reduce developers’ *cognitive load* while translating between the two, freeing them up to concentrate on resolving the vulnerability.

**4.4.6 Workflow Continuity (29)**

Developers do not use static analysis tools in isolation. Tools must synergize with other tasks in a developer’s workflow, such as editing code, testing changes, and reviewing patches. Issues in this category arose when tools dictated workflows that were not compatible with a developer’s natural way of working.
Tracking Progress (FSB, RIPS, Flawfinder): Developers work with static analysis to reduce the number of vulnerabilities in their code, making code changes to progressively resolve warnings. However, the way some tools present their results does not support developers in tracking their progress. Instead of reporting which vulnerabilities were added and removed between scans, tools only provide snapshots of all the current vulnerabilities in a project at a given time. This is only somewhat problematic when a developer wants to consider the effects of their changes on a single vulnerability notification. For example, if the tool at first reports 450 vulnerabilities, and then reports 449 after the developer applies a patch, then they can assume their patch fixed the one vulnerability. However, when a developer or his/her team makes sweeping changes to address many several vulnerabilities simultaneously, it becomes much more difficult to determine which issues were added and removed between scans based only on two snapshots.

Batch Processing (FSB, RIPS, Flawfinder, CTool): Secondly, all four tools we evaluated dictate that developers address notifications one at a time. This is problematic because projects can contain many repeated occurrences of the same vulnerability patterns. For instance, CTool detected 30 occurrences of the same potential package protection vulnerability across POI. Serially fixing these vulnerabilities can be tedious and error-prone—developers have to make sure they consistently apply the right fix to each of these occurrences. Since tools are technically capable of scanning many more files than developers could manually scan, they must also enable developers to fix similar vulnerabilities in batches. Otherwise, the tool far outpaces the developer, adding vulnerabilities to his/her work queue much faster than they can dismiss them.

Discussion: Static analysis tools for security can be improved to better support developers’ workflows. By keeping track of the vulnerabilities added and removed in each scan, tools could provide the option of presenting developers with scan diffs. These diffs could
help developers identify changes that add many potential vulnerabilities to the code as well as draw developers’ attention to changes that remove vulnerabilities.

Tools could also support developers’ workflows by enabling them to process similar vulnerabilities in batches. One way tools could accomplish this is by integrating with automated refactoring tools. Rather than fixing the individual occurrences of an issue one at a time, developers could describe to the static analysis tool how they would like to fix the issue, then the static analysis tool would apply a refactoring to fix all occurrences of the issue.

4.5 Limitations

We only evaluated four tools and certainly expect that evaluating additional security-oriented static analysis tools with unique interface features could yield new usability issues. To mitigate these threats to generalizability, we selected analysis tools that each covered distinct interface features, and included both commercial and open-source tools in our evaluation. We also make our evaluation guide available so that our results can easily be extended to include additional static analysis tools for security. Still, we do not claim that any of the usability issues we identified necessarily generalize to other static analysis tools.

We also acknowledge that the evaluators’ individual tool usage styles might have influenced the issues they identified. For instance, the evaluators’ ad hoc approach to reading documentation might not align with some users who read all available documentation before using a tool. However, in this example, many issues we identified, like scalability issues, would not be influenced by documentation reading. Nonetheless, to bolster the ecological validity of our study, we selected real-world applications and instructed evaluators to perform realistic tasks in Phase 1 of their evaluation.
Our choice of usability evaluation technique also influences our results. We chose to use heuristic walkthroughs, because they enable relatively few evaluators to identify usability issues. Compared with laboratory studies, where qualified participants must be recruited and might typically spend only an hour using a tool, evaluators conducting heuristic walkthroughs have more time to find deeper issues.

Relatedly, our choice of heuristics might have influenced the usability issues we identified. We chose to use these heuristics because prior studies have shown that domain-specific heuristics can be more effective. However, it’s possible that using different heuristics would reveal different usability issues.

Finally, some of the issues we identified might be mitigated by experience with a tool or additional formal training. As evaluators, we familiarized ourselves with each of the analysis tools and used documentation published online to learn to use the tools. However, we did not participate in any formal training courses specific to these tools. It is possible more training would have reduced the impact of some issues, for example, by teaching us how to interpret the vulnerability prioritization scales. Nonetheless, while training might help with some of these issues, it is an individual cost to each developer using these tools, and only improves the experience of developers who participate in training.

4.6 Conclusion

To answer RQ4, we evaluated the usability of four security-oriented static analysis tools, Find Security Bugs, RIPS, Flawfinder, and a commercial tool. The heuristics we used to evaluate these tools were derived directly from the categories of information needs in Chapter 2 (i.e., the information needs we reference in the thesis statement). This study
revealed usability issues that detract from all four tools. For instance, we noted that the
interfaces of these tools do not scale as more vulnerabilities are detected in a project.

This chapter supports our thesis statement in two ways: 1) by examining three additional
tools beyond Find Security Bugs (which we already studied in Chapter 2), we support the
generalizability of the thesis statement; 2) by identifying usability issues in widely-used
static analysis tools and creating a protocol for conducting additional usability evaluations,
we demonstrate the practical applicability of our thesis statement.
CHAPTER

FIVE

RELATED WORK

We have organized the related work into four sections. Section 5.1 summarizes prior studies that have evaluated security tools. Section 5.2 describes related research in the adjacent field of end user security. Section 5.3 covers work in the area of usable static analysis. Finally, Section 5.4 discusses studies that characterize developers’ various security activities.

5.1 Evaluating Security Tools

Developers’ security tools are constantly being improved, enabling them to scan more code and detect new types of defects. As the state of the art improves, various studies have assessed the effectiveness of the security tools developers use to find and remove vulnerabilities from their code [Mar05; Aus11; Liv05]. However, these evaluations of developer security tools have primarily focused on the technical aspects of the tools, such as the defects they detect and their false positive rates, rather than focusing on the usability of those tools. In this section we will summarize related work that has evaluated developer security tools.
Several studies have conducted comparative evaluations of developer security tools. For instance, Zitser and colleagues evaluated five static analysis tools that detect buffer overflow vulnerabilities, comparing them based on their vulnerability detection and false alarm rates [Zit04]. Comparing a wide range of Android security tools, Reaves and colleagues categorize tools according to the vulnerabilities they detect and techniques they use [Rea16]. Their results do also include some usability experiences, such as how long it took evaluators to configure the tool and whether output was human-readable. Austin and colleagues compare four vulnerability selection techniques: systematic manual penetration testing, exploratory manual penetration testing, static analysis, and automated penetration testing [Aus13]. Considering the number of vulnerabilities found, false positive rates, and tool efficiency, they conclude that multiple techniques should be combined to achieve the best performance. Emanuelsson and Nilsson compare three static analysis tools, Coverity Prevent, Klocwork K7, and PolySpace Verifier, in an industrial setting [Ema08].

Jovanovic and colleagues evaluate their tool, PIXY, a static analysis tool that detects cross-site scripting vulnerabilities in PHP web applications [Jov06]. 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 [Liv05]. They also found their tool to be effective because it had a low false positive rate. As yet another example, Dukes and colleagues conducted a case study comparing static analysis and manual testing vulnerability-finding techniques [Duk13]. 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, this dissertation examines the usage of static analysis tools through the lens of developers’ information needs and, accordingly, provides a novel framework for evaluating security tools.

There have been a limited number of studies that account for usability in their evaluations of developer security tools. Assal and colleagues conducted a cognitive walkthrough evaluation of Findbugs to determine how well it helped developers count the number of vulnerabilities in a codebase [Ass16]. Based on the usability issues identified in this study, the authors created a tool, Cesar, designed to be more usable. Assal and colleagues used a similar approach to our heuristic walkthroughs (Chapter 4), however they focus on a single tool and the very specific task of counting vulnerabilities. Nevertheless, this work is a promising example of how usability evaluations can be used to improve state-of-the-art developer security tools. Similarly, Nguyen and colleagues motivate the design of their tool, FixDroid, by describing usability issues that affect Android lint tools. FixDroid uses data flow analysis to help secure Android apps [Ngu17]. Again, this work by Nguyen and colleagues serves as an example of how the types of usability issues we systematically identify in Chapter 4 could be used to design more effective tools.

In summary, static analysis tools have traditionally been evaluated based on their technical capabilities, often overlooking the usability of these tools. Researchers have recently begun evaluating the usability of developer security tools and, as a result, have gained new insights leading to the improvement of these tools. This dissertation advances this research area by advocating for more usability evaluations of security tools and providing a framework for such evaluations.
5.2 Improving the Usability of End User Security Tools

Several studies have usability into consideration in the adjacent domain of end user security tools. Through these evaluations, researchers identified usability issues in various end user security tools, ranging from encryption tools to Tor browsers. Collectively, these studies have improved the usability of end user security tools by contributing a better understanding of how users interact with these tools.

In their foundational work, Whitten and Tygar studied the usability of PGP, an end user encryption tool [Whi99]. They evaluated PGP using a combination of a cognitive walkthrough and a laboratory user test to identify aspects of the tool’s interface that failed to meet a usability standard. The issues revealed by their study covered topics such as irreversible actions and inconsistent terminology.

Since Whitten and Tygar’s study, others have successfully applied similar approaches to study the usability of additional end user tools. For instance, Good and Krekelberg studied the usability of the Kazaa P2P file sharing system [Goo03]. Their findings suggest that usability issues led to confusion and privacy concerns for users. Similarly, Gerd tom Markotten studied the usability of an identity management tool using heuristic evaluations and cognitive walkthrough [Mar02]. Additionally, Reynolds and colleagues conducted two studies to understand multiple aspects of YubiKey usability [Rey18].

Clark and colleagues conducted cognitive walkthroughs to examine the usability of four methods for deploying Tor clients [Cla07]. Based on their evaluation, they make recommendations for facilitating the configuration of these systems. Also studying the usability of Tor systems —through a user study instead of a cognitive walkthrough—Gallagher and colleagues conducted a drawing study to elicit users’ understanding, and misunderstandings, of the underlying system [Gal17].
Like these previous studies, we are concerned with the usability of security tools and strive to better understand usability issues by conducting empirical studies. We are encouraged by these studies’ successful applications of evaluation techniques like cognitive walkthroughs and heuristic evaluations to end user tools. In contrast to these prior studies, we evaluate static analysis tools to identify usability issues in the domain of developer security tools.

5.3 Usable Static Analysis Tools

Outside the domain of security, relatively few researchers have studied the human aspects of using static analysis tools to identify and resolve defects. Muske and Serebrenik survey 79 studies that describe approaches for handling static analysis alarms [Mus16]. They organize existing approaches into seven categories, which include “Static-dynamic analysis combinations” and “Clustering.” According to their categorization, these studies, including this dissertation, would best be categorized as “Simplifying manual inspections.”

Johnson and colleagues interviewed 20 developers to identify reasons why developers did not use static analysis [Joh13]. One relevant reason that participants reported is that tools may report the presence of errors, but do not enable developers to fix those errors. As a result of these findings, Johnson and colleagues developed the notion of “Bespoke Tools,” which present adapted notifications to developers depending on the concepts they know [Joh15].

Sadowski and colleagues [Sad15] report on the usability of their static analysis ecosystem at Google, Tricorder. Their experiences suggest that warnings should be easy to understand and fixes should be clear, which motivates the work in this thesis. Similarly, Ayewah and colleagues describe their experiences running static analysis on large code bases. They
make suggestions for how tools should help developers triage the numerous warnings that initially might be reported [Aye07].

Prior studies on the usability of general-purpose static analysis tools provide a foundation for this dissertation, which focuses specifically on how developers resolve security vulnerabilities with static analysis. This dissertation reveals new insights about security-specific aspects of how developers use static analysis tools, such as how developers need information about potential attacks. Additionally, this dissertation confirms that the findings from these prior studies of general static analysis tool usage apply to security tools. For instance, prior studies report that false positives detract from the usability of static analysis tools [Sad15; Joh13]. As we observed in Chapter 2, developers are similarly concerned with whether vulnerabilities reported by security tools are false positives.

### 5.4 Software Security Activities

This dissertation focuses on one particular activity developers practice to help secure the software they create, namely using static analysis tools to resolve security vulnerabilities. Several empirical studies explore a range of other software security activities. Tahaei and Vaniea [Tah19] performed a systematic literature review, which provides an overview of many of these types of activities. Their literature review covers 49 publications with software developers as participants and identifies eight themes: organizations and context; structuring software development; privacy and data; third party updates; security tool adoption; application programming interfaces; programming languages; and testing assumptions. According to their categorization, the work in this dissertation falls under the security tool adoption theme.
One proximate study to this work was conducted by Thomas and colleagues [Tho18]. In this study, they interviewed application security experts who examine source code, or perform static or dynamic analysis, labeling them under the general umbrella of security auditors. Their findings suggest that security auditors are generalists, and able to conduct security activities interchangeably. In contrast, our findings in Chapter 3 show that even within red teams, software security engineers are agile in their abilities but also deeply specialize in a particular aspect of software security. Another difference is that their study also reports that security analysis used off-the-shelf static and dynamic analysis tools, such as Fortify, to find security vulnerabilities. Our findings (and Hafiz and Fang [Haf16]) suggest that is not the case with exploit hunting: the use of static and dynamic analysis tools are relatively uncommon and considered ineffective for finding exploits, though several red teams have adopted fuzzing techniques to identify vulnerabilities that lead to exploits. One explanation is differences in security maturity of the organizations and where these activities occur. The use of off-the-shelf analysis tools for red team activities may be useful in organizations where developers do not have a strong security culture [Xia14; Wit15; Pol17], and thus, do not use or underutilize security tools. Our informants in Chapter 3 reported that static and dynamic analysis are already part of the continuous integration pipeline, and security issues identified by these tools will break the build. Thus, these issues are already addressed by software developers before getting to the software security engineers. A trade-off, of course, is that this pushing additional security responsibilities on to the developer [Hil17].

Assal and Chiasson conducted a survey to explore the interplay between developers and their software security processes [Ass19]. While this dissertation focuses on tool usage, their work helps to capture the social and organization processes that also contribute to vulnerability resolution. Among other things, they report on the prevalence of different
strategies for handling software security. Most developers surveyed indicated that they rely on colleagues or experts for support.

Hafiz and Fang [Haf16] conducted an empirical study through reports of the three prominent security vulnerabilities, including buffer overflows, SQL injection, and cross-site scripting—with the goal of understanding the activities and tools used during vulnerability discovery. We credit Hafiz and Fang [Haf16] for their inventive approach to obtaining informants: the study looks at reported vulnerabilities at SecurityFocus\(^1\) and then contacts the authors of the reports to obtain first-hand information about activities relating to security vulnerability discovery. Like Hafiz and Fang’s study, this dissertation also empirically explores developers’ activities surrounding vulnerability resolution.

\(^1\)https://www.securityfocus.com/
In the introduction of this dissertation we first introduced the following thesis statement:

*To diagnose potential security vulnerabilities while using static analysis tools, developers must strategically synthesize four categories of information about: attacks, vulnerabilities, the software itself, and the social ecosystem that built it.*

To support this thesis statement, we introduced four research questions. Here we summarize the answers to these questions based on our findings from the previous chapters:

**Information Needs (RQ1):** What information do developers need while using static analysis tools to diagnose potential security vulnerabilities?

In Chapter 2, we observed developers while they were using a static analysis tool to diagnose potential security vulnerabilities. We analyzed the questions they asked, identifying 17 categories of information needs. Table 2.3 enumerates these categories and Section 2.6 describes them in more detail. Together, these 17 categories of information substantiate the following portion of the thesis statement, “...several categories of information pertaining to attacks, vulnerabilities, the software itself, and the social ecosystem that built it.”
Strategies (RQ2): What are developers’ strategies for acquiring the information they need?

To answer this question, we developed a new notation, *strategy trees*, for describing developers’ strategies (Section 2.4.1). Using this notation, we reanalyzed the data from RQ1, identifying developers’ strategies and annotating those strategies when they contributed to success and failure. As a result, we identified a categorization of strategies that parallels the categories of information needs. These strategies are presented alongside their respective information needs in Section 2.6. We found that developers employed complex strategies to acquire all the different types of information they needed. This research question substantiates the first half of the thesis statement, “To diagnose potential security vulnerabilities while using static analysis tools, developers must strategically synthesize several categories...”

Specialization (RQ3): How do specialized developers on security red teams use static analysis?

To answer this question, we interviewed developers who are members of security red teams (Chapter 3). We found that, like the developers we studied in RQ1 and RQ2, specialized developers need to: examine source code to obtain various types of information, such as the locations in the code where an attacker can enter the system (Section 3.3.1.1); collaborate with teams to get information about the social ecosystem that built the software (Section 3.3.1.3); and understand how attackers could exploit vulnerable code (Section 3.3.3.2). However, unlike traditional software engineers, security specialists often have to write their own ad-hoc tools rather than rely on out-of-the-box solutions (Section 3.5.2). This research question helps us extend our thesis statement to cover a broader set of developers.

Usable Tools (RQ4): How do existing static analysis tools support developers’ information needs and strategies?
To answer this question we conducted heuristic walkthrough evaluations of four security-oriented static analysis tools, using the information needs we identified in RQ1 as the basis for our heuristics (Chapter 4). We found that usability issues detracted from all four tools we analyzed. Section 4.4 summarizes these issues and also provides sketches for designs that mitigate several of these issues. Answering this research question helped us extend the thesis statement to cover more tools than just Find Security Bugs (which we studied exclusively in RQ1 and RQ2). Further, by answering this research question, we have demonstrated how the thesis statement can be applied to finding practical usability issues in tools.

6.1 Implications

The first, and most straightforward implication of this dissertation is that it reveals how developers use static analysis tools to resolve vulnerabilities. To that end, we have categorized developers’ information needs and strategies (Chapter 2), and characterized how security experts on red teams use static analysis tools to contribute to the resolution of vulnerabilities (Chapter 3). Along with the contributions of our heuristic walkthrough study (Chapter 4), these findings enables us, and future toolsmiths, to design systems and tools that help developers fix vulnerabilities more effectively. Essentially, by better understanding what developers need from static analysis tools for security, we can create tools that better satisfy their needs. As a result of these improvements, this dissertation will hopefully empower more people to contribute to software security.
6.2 Recommendations for Toolsmiths

Here we provide a succinct list of recommendations for toolsmiths, based on the findings of this dissertation:

- Static analysis tools should enable developers to fix the vulnerabilities they detect by providing the various categories of information developers need.

- Information alone is not enough. Help novices to behave like experts, and experts to work more efficiently, by encapsulating effective strategies in vulnerability notifications.

- Developers range in their security expertise and familiarity with particular vulnerabilities; consider your audience.

- Out-of-the-box solutions have limited utility; strive to support scripting and extensions.

- Evaluate the usability of your security tools, fix usability issues, repeat.

6.3 Future Work

Armed with a better understanding of how software developers resolve security vulnerabilities with static analysis, what should we do now? We conclude by discussing three potential areas of future work.

- In this dissertation, we have argued that usability evaluations of developer security tools are detrimentally uncommon (Section 5.1). We have also specified a low-cost
heuristic walkthrough technique for evaluating such tools (Section 4.3.3), and demonstrated its effectiveness by identifying several usability issues in popular static analysis tools (Section 4.4). We hope that this dissertation enables and inspires practitioners to conduct more usability evaluations.

- Throughout this dissertation we have argued that we can improve developers’ security practices by improving the usability of their tools. However, it is likely worthwhile to explore other avenues to achieve the same result. One particularly promising area is security education. For instance, understanding how to provide educational feedback to developers as they resolve security vulnerabilities remains a gap in the research literature [Tah19]. This dissertation helps us understand how developers use security tools; that understanding could be leveraged to understand how to provide better educational feedback. Relatedly, it might also be beneficial to explore how to teach developers the security mindset and effective strategies for resolving security vulnerabilities—Ko and colleagues have begun exploring how to teach effective debugging and code reuse strategies to novice programmers in high schools [Ko19].

- Tools! In Section 2.7 and throughout Section 4.4 we provide sketches for tools and interface modifications that could improve developers’ experience while resolving security vulnerabilities. For instance, we describe a prototype tool that has shown promise in helping developers navigate program flow more effectively (Section 2.7.1) and a vision for a tool that helps developers search for information more effectively (Section 2.7.3). While these ideas still need to be implemented and thoroughly evaluated, these designs only begin to scratch the surface of what is possible when we consider usability and account for developers’ strategic information needs when designing tools.
Defending software against attack, developers turn to static analysis tools to detect and fix security vulnerabilities early. Yet, the usability of these tools has been critically overlooked. This dissertation has advocated for the developers on the front lines of software security by building a better understanding of their needs while using static analysis. We have shown that this understanding can be used to find usability issues in existing tools and can inspire the design of entirely new tools. Ultimately, we hope that this dissertation has drawn attention to the usability of static analysis tools for security and energizes further efforts to improve those tools.


A.1 Recruitment Email

Hi [Name],

My name is Justin Smith. I am a PhD student working with Dr. Murphy-Hill on doing a study with iTrust. I am looking for students who have worked on this system in the past. Participation in this study will help improve the design of security related software engineering tools.

The study will take approximately 45 – 60 minutes to complete. Participants must perform the study at NC State University, Room [TBD]. Familiarity with iTrust is required.

If you have any questions about the study or would like to schedule a time to participate, please respond to me at this email address (jssmit11@ncsu.edu). Participants must be 18 years of age or older.

Thanks!

Justin Smith
A.2 Procedure

**Introductory Script:** This experiment is designed to evaluate how you approach security vulnerabilities. This session will be recorded using screen and audio capture software. I will present you with some code that FindBugs has analyzed and ask you to explain what is going on.

This experiment is not designed to test your ability as a developer. Instead, we are attempting to understand how you approach code vulnerabilities. During the experiment, you can use any web resource. I have opened a web browser, which you are welcome to use at any point.

For the purposes of this experiment, imagine that you are in charge of security for iTrust, which is nearing release. You must justify any proposed code changes that you make. Currently, the software may contain security vulnerabilities. FindBugs has been configured to direct you to 4 areas of concern. For each area, I’ll ask you about 10 questions.

Do you have any questions before we begin?

1) Navigate to first warning message

2) Read through the code/warning message

3) Try to think aloud. So, say any questions or thoughts that cross your mind regardless of how relevant you think they are.

**Experimental Questions:**

• Can you explain what this warning is trying to tell you?

• What are you exploring/trying to figure out right now?

• What information do you need to proceed?
• For the sake of time, explain to me what you would keep doing?

• Would you normally do this in your work?

• How does the wording effect the process?

• Based on your understanding of this error message, what is the percentage likelihood you would modify this section of the 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 this error message?

• On a scale of 1-5, how confident are you that you are making the correct judgment?

• What part of the message is most helpful?

• What information would you like to see added to the error message?

• Look at the error message one last time. Is there any information that you initially disregarded?

• Anything else I should know?

**Post-Experiment Questions:**

• What is your current job title (if student, indicate so here)

• How many years have you been programming?

• How many years of professional programming experience do you have?

• Over the last year, about how many hours per week would you say you spend programming, on average?
• When programming, do you typically use security tools? Y/N
  
  – If Y, Which tool(s) do you use? Why do you use these security tool?
  
  – If N, can you tell me the reasons why you don't use security tools?

• On a scale form 1 to 5, how familiar are you with security vulnerabilities? (1 = not at all, to 5 = very familiar)

• In general how effectively did the FindBugs communicate information to you? What did you or didn't you like?

• What security related questions could the tool answer better?

A.3 Categorized Questions

Here we present the full list of questions we identified and categorized. The #Y.X notation corresponds with the tags used in strategy and assumption analysis.

**Developer Planning and Self-Reflection #1.X**

What was I looking for? #1.1
What do I know now? #1.2
What should I do first? #1.3
Do I understand? #1.4
Have I seen this before? #1.5
What was that again? #1.6
What's the next thing I should be doing? #1.7
Where am I (in the code)? #1.8
Is this worth my time? #1.9
Is this my responsibility? #1.10
Why do I care? #1.11
Have I used this package before? #1.12
Am I making the right decision for my code? #1.13
How long have we been talking? #1.14

Vulnerability Severity and Rank #2.X
How serious is this vulnerability? #2.1
Are all these vulnerabilities the same severity? #2.2
How do the rankings compare? #2.3
What do the vulnerability rankings mean? #2.4

Data Storage and Flow #3.X
How is data put into this variable? #3.1
Does data from this method/code travel to the database? #3.2
Where does this information/data go? #3.3
How do I find where the information travels? #3.4
How does the information change as it travels through the program? #3.5
What does this variable contain? #3.6
Is any of the data malicious? #3.7
Where is the data coming from? #3.8
What are the contents of this SQL file? #3.9
Where along the data pipeline is this method? #3.10
Where along the pipeline do you want to make sure the data is secure? #3.11

Application Context/Usage #4.X
What is this method/variable used for in the program? #4.1
Are we handling secure data in this context? #4.2
What is the context of this vulnerability/code? #4.3
How does the system work? #4.4
Will usage of this method change? #4.5
Is this method/variable ever being used? #4.6
Is this code used to test the program/functionality? #4.7
Does test utils get sent to production code? #4.8
How many layers of code/security have we passed through before reaching this point? #4.9

**Control Flow and Call Information #5.X**
What is the call hierarchy? #5.1
How can I get calling information? #5.2
Who can call this? #5.3
Where is the method being called? #5.4
What causes this to be called? #5.5
Are all calls coming from the same class? #5.6
What gets called when this method gets called? #5.7
How often is this code called? #5.8
Is this called from another less secure API? #5.9
How frequently does this method get called? #5.10
How many calls does it take to reach the vulnerability? #5.11
What are the parameters called? #5.12
Is the entire set of possible parameters known in advance? #5.13

**Resources and Documentation #6.X**
What type of information does this resource link me to? #6.1
What is the documentation? #6.2
Can my team members/resources provide me with more information? #6.3
Where can I get more information? #6.4
What information is in the documentation? #6.5
Is this a reliable/trusted resource? #6.6
How do resources prevent or resolve this? #6.7
How should I word my search to get the right information? #6.8
Is this the OWASP site? #6.9
Does the method have a javadoc? #6.10

**Understanding Approaches and Fixes #7.X**

Why should I use this alternative method/approach to fix the vulnerability? #7.1
What are the alternatives for fixing this? #7.2
Does the alternative function the same as what I’m currently using? #7.3
When should I use the alternative? #7.4
Is the alternative slower? #7.5
Are there other considerations to make when using the alternative(s)? #7.6
How does my code compare to the alternative code in the example I found? #7.7
Is secure random good? #7.8
Is prepared statement a Java thing or a 3rd party library? #7.9
How secure is secure random? #7.10

**Code Background and Functionality #8.X**

Who wrote this code? #8.1
Why is this code needed? #8.2
Is this library code? #8.3
Are there tests for this code? #8.4
Why was this code written this way? #8.5
What does this code do? #8.6
Is this code doing anything? #8.7
How much effort was put into this code? #8.8
Why are we using this API in the code? #8.9
Is there a list of files we are going to iterate over? #8.10
Is rand static or final? #8.11
Why are we collecting this information in the first place? #8.12
Is there a variable called “dir” at the top? #8.13
How does the code accomplish design goals? #8.14
What is the name of the current class? #8.15
Could the pointer be null? #8.16
Does it throw an exception instead #8.17

**Locating Information #9.X**
Where is this used in the code? #9.1
Where are other similar pieces of code? #9.2
Where is this artifact? #9.3
Is this artifact located in this class? #9.4
Where is this method defined? #9.5
Where is this class? #9.6
Where is the next occurrence of this variable? #9.7
How do I track this information in the code? #9.8
How do I navigate to other open files? #9.9
Where is this class instantiated #9.10
Where was the class instantiated #9.11

**Assessing the Application of the Fix #10.X**
How hard is it to apply a fix to this code? #10.1
How do I use this fix in my code? #10.2
How do I fix this vulnerability? #10.3
Is there a quick fix for automatically applying a fix? #10.4
Will the code work the same after I apply the fix? #10.5
Can these fix suggestions be applied to my code? #10.6
Will the error go away when I apply this fix? #10.7
Does the code stand up to additional tests prior to/after applying the fix? #10.8
What other changes do I need to make to apply this fix? #10.9

Preventing and Understanding Potential Attacks #11.X
How can this vulnerability lead to an attack? #11.1
How can I replicate an attack that exploits this vulnerability? #11.2
Why is this a vulnerability? #11.3
What are the possible attacks that could occur? #11.4
How can I prevent this attack? #11.5
How should I address this problem? #11.6
What is the problem (potential attack)? #11.7
Is this a real vulnerability? #11.8
How do I find out if this is a real vulnerability? #11.9
How much value would you get from randomly injecting stuff? #11.10
Is releaseForms handling data securely? #11.11

Relationship Between Vulnerabilities #12.X
Does this other piece code have the same vulnerability as the code I’m working with? #12.1
Are all the vulnerabilities related in my code? #12.2
Are all of these notifications vulnerabilities? #12.3

End-User Interaction #13.X
Is there input coming from the user? #13.1
Does the user have access to this code? #13.2
Does user input get validated/sanitized? #13.3

**Notification Text #14.X**
What is the relationship between the error message and the code? #14.1
What code caused this error message to occur? #14.2
What does the error message say? #14.3

**Understanding Concepts #15.X**
What is the term for this concept? #15.1
Do these words have special meaning related to this concept/problem? #15.2
How does this concept work? #15.3
What is this concept? #15.4
Is Java's true/false considered enum? #15.5
What is the nomenclature for testing suites? #15.6

**Confirming Expectations #16.X**
Is this doing what I expect it to? #16.1

**Uncategorized #17.X**
Have I exhausted all references in this method? #17.1
Is this method deprecated? #17.2
What does the constructor and datatype look like? #17.3
Was the author of this code aware of security issues? #17.4
Has the package I'm using been tested? #17.5
Is it done the same way in Python? #17.6
Are there non-malicious ways the code could break? #17.7
How can I transfer text from the IDE to the browser? #17.8
What parts of the code do you trust with this data? #17.9

What do you want to trust? #17.10

**Understanding and Interacting with Tools #18.X**

Why is the tool complaining? #18.1

What is the tool output telling me? #18.2

Can I verify the information the tool provides? #18.3

What is the tool keybinding? #18.4

What is the tool’s confidence? #18.5

What tool do I need for this? #18.6

How is the information presented by the tool organized? #18.7

What is the related stack trace for this method? #18.8

How can I annotate that these strings have been escaped and the tool should ignore the warning? #18.9

**Discarded (5) #Discard**

### A.4 Assumptions

<table>
<thead>
<tr>
<th>Rephrased Assumption (+/- indicates correct/incorrect)</th>
<th>Task</th>
<th>Participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Assume the SQL file comes from the user</td>
<td>T1</td>
<td>P1</td>
</tr>
<tr>
<td>- The SQL file is coming from the user.</td>
<td>T1</td>
<td>P1</td>
</tr>
<tr>
<td>+ Assume the results from call hierarchy are in alphabetical order</td>
<td>T1</td>
<td>P1</td>
</tr>
<tr>
<td>+ Assume this statement returns a number between 0 and 26</td>
<td>T2</td>
<td>P1</td>
</tr>
<tr>
<td>+ The two instances of the bug are the same.</td>
<td>T3</td>
<td>P1</td>
</tr>
<tr>
<td>- Assume that redisplaying partial information on the web form is not dangerous *</td>
<td>T3</td>
<td>P1</td>
</tr>
<tr>
<td>Assumption</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>Assume that someone else on the team wrote the validation code correctly.</td>
<td>T3</td>
<td></td>
</tr>
<tr>
<td>Assume that someone else on the team wrote the validation code correctly.</td>
<td>T3</td>
<td></td>
</tr>
<tr>
<td>Assume the validation code is correct.</td>
<td>T3</td>
<td></td>
</tr>
<tr>
<td>Assume the validation method throws an error.</td>
<td>T3</td>
<td></td>
</tr>
<tr>
<td>Assume the validate method detects bad input.</td>
<td>T3</td>
<td></td>
</tr>
<tr>
<td>Assume the form input is not used anywhere else.</td>
<td>T3</td>
<td></td>
</tr>
<tr>
<td>Assume the data in the bean is valid.</td>
<td>T4</td>
<td></td>
</tr>
<tr>
<td>Assume bean data has been validated.</td>
<td>T4</td>
<td></td>
</tr>
<tr>
<td>Assume DAO data has been validated.</td>
<td>T4</td>
<td></td>
</tr>
<tr>
<td>Assume all forms are validated.</td>
<td>T4</td>
<td></td>
</tr>
<tr>
<td>This attack could be prevented by preventing ../ and other similar escape sequences.</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>Assume that Eclipses's call hierarchy for java and C++ are similar.</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>By changing the import statement, I could determine if random is a valid method in secure random.</td>
<td>T2</td>
<td></td>
</tr>
<tr>
<td>The SQL file contains queries to be run against the database.</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>Assume the calls in create tables and drop tables can be trusted.</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>Assume prepared statements prevent SQL injection</td>
<td>T4</td>
<td></td>
</tr>
<tr>
<td>Presumably the Django framework is safe. (45 vulnerabilities in CVE)</td>
<td>T4</td>
<td></td>
</tr>
<tr>
<td>TestDataGenerator code is only accessible by admins.</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>Assume Java.io.file is secure.</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>All external packages have been tested and work securely.</td>
<td>T1</td>
<td></td>
</tr>
</tbody>
</table>
+Assume the typo could be fixed by replacing “in” with “it”
+Assume string parameters correlate with session variables.
-True and false in Java are implemented as enum.
+Assume the file is opened in order.
+Assume the method is used to make first passwords or reset passwords.
+The secure random library handles correctly seeding the random number generator.
-Assume the code author didn’t know what he was doing.
+Assume there is nothing bad in the file.
+Assume it’s safe to execute the SQL file.
+I believe I checked the caller for all of the methods.
+The method is not vulnerable because it’s called only from the testdatagenerator.
+Secure random is sufficient for this application.
-Secure random doesn’t provide a nextInt method.
+More secure random numbers take longer to generate.
+Assume owasp.org is OWASP’s site.
+I’m looking at the official OWASP site.
-The warnings are false positives.
+The pattern that findbugs detects exists in other parts of the application as well.
-The values are only being used for SQL queries (not being used in other dangerous contexts)
+The code uses the statement class, not prepared statements.
- Assume all DAO’s use prepared statements.
- Assume the value gets selected from a drop down menu.  
- Assume there is a flaw in the file.io method.  
- The issue relates to using SQL files that directly access the file system.  
  + The code is used to release records.  
  + Assume that by this point the user has been authenticated.  
- Assume the problem is that there is no validation on anything.  
- There are not drawbacks to overly redundant security.  
  + Presume the tool doesn't know how the method is used.  
  + Presume the code is used to build a database.  
  + Assume the method creates passwords on first login.  
  + Assume iTrust uses jUnit testing.  
- Assume all test classes contains setup and teardown methods.  
  [beanbuildertest]  
  + Assume this class is responsible for generating random passwords.  
  + Presume the password is used for login purposes.  
- Assume the random seed is actually random.  
  + Assume the notification text is correct.  
- Presume the vulnerability pertains to the DAO objects.  
  + Assume the method handles a post request.  
  + Presume http servlet_request is an api object.  
- Presume the object wraps the get protocol.  
  + Assume the bean class doesn't perform any sanitization.  
  + Presume once in the form, the data gets checked with logic.  
  + Presume the method handles post requests.
- Presume that the form handles data securely.  
+ Presume the statement is a DAO object.
B.1 Interview Script

The interview session should take approximately 40 minutes.

B.1.1 Introductions

Thank you for taking the time to talk with us today. We really appreciate your help with this research and are looking forward to hearing about your experiences. To give a little background on us, we are researchers who work in software engineering and AI. We are in the early stages of identifying opportunities to help other teams in the domain of security.

Before we get started, is it ok if we record this conversation? The full interviews will be kept confidential to Microsoft and your identity and any personal information will remain anonymous. Portions may be used for reports made to leadership team and external academic community. Previous work on studying challenges with data scientists.
B.1.2 Background:

• What do you at Microsoft?

• How did you come about your current role?

• If you were previously at a different company, how did the practices there differ?

• In what ways was is better/worse there?

• How is security engineering different from software engineering?

• What makes someone an expert security engineer?

• How do you think security engineers are perceived by others in the company or outside the community?

B.1.3 Workflow:

• What’s a typical day like for you?

• How do you discover security vulnerabilities?

• How do you go about fixing vulnerabilities?

• What tools do you use?

• What do you like about those tools?

• What do you think could be improved?

• What are the biggest pain points or challenges you face?
• Are you involved in the process of publicly disclosing vulnerabilities? Making the decision to?

• Is your work mostly an individual effort or collaborative. If collaborative, how do you share information with your team members? How is work divided amongst your teammates?

• If you split up all of the activities that you have to do, how much of it is your “core security” work versus other activities?

B.1.4 Collaboration

• What are the different roles in the security space? What are we missing?

• What unique skills are required for each role?

• What other teams do you work with?

• What do the people above you care about in terms of security?

• Which projects and how do you decide?

• Who is the output for? Developers? Other PM?

• How do you communicate security vulnerabilities to others?

• How do security software engineers collaborate with software engineers in products teams? Are there any challenges to doing so?

• How do you measure success or improvement?
B.1.5 Learning:

- What sources?
- Could you tell us about a time when you adopted new security tool or practice?
- In what ways did this new practice make you more/less productive?
- Ask about strategies: how do you learn strategies, transfer strategies?
- What's the process for onboarding a new team member?

B.1.6 Wrapups:

- Ask for continued support: As this research project progresses are you open to us
  following up with you?
- Do you have any questions or comments you would like to add before we wrap up?
- Finally, do you think there's anyone else we should talk to?

B.2 Recruitment Email

Subject: Security tools chat?

Hi [name],

We’re researchers at Microsoft studying how security software engineers use security tools. We’d like to have conversation with you about how you use security tools during your day-to-day activities and obtain feedback you might have on how security tools could be
made better. We've sent you this invitation because of your organization, title, and tags as listed on [employee directory]. Your feedback will be used for research in improving the usability and effectiveness of security tools as used by practitioners at Microsoft.

I've added an hour block to your calendar at your office location, with the expectation that the interview will take about 40 minutes. Please feel free to accept or reject depending on your availability or suggest an alternate time that might work better for you. If you're not the right person to ask about this topic, we'd appreciate it if you could suggest others individuals in the organization who would be more appropriate.

Thanks so much!

Justin Smith, Research Intern

Titus Barik, Researcher
C.1 Heuristic Walkthrough Evaluation Guide

C.1.1 List of test suites

All tests from: https://samate.nist.gov/SRD/testsuite.php

- **C/C++**: OpenSSL 2015-10-28 1.0.1e. https://www.openssl.org/source/old/1.0.1/
- **PHP**: Wordpress version 2.0

C.1.2 Heuristic Walkthrough: Pass 1

You work at a software company and are responsible for the security of your product. Your team uses [toolname] to detect vulnerabilities early in the development process. The tool
detects several potential vulnerabilities in the code, but you only have a limited amount of
time to resolve them. Given these constraints, work through the following tasks:

C.1.2.1  Prioritized list of tasks

1. Choose a vulnerability that you’d be most likely to inspect first.
2. Determine whether the reported vulnerability is actually a vulnerability.
3. Propose a fix to the vulnerability.
4. Assess the quality of your fix.

Repeat these tasks until you feel satisfied with your assessments. Use the questions
below to guide your evaluation. Record any usability problems you encounter during this
phase

C.1.2.2  Guiding questions

1. Will users know what they need to do next? It is possible that they simply cannot
   figure out what to do next.
2. Will users notice that there is a control (e.g., button, menu) available that will allow
   them to accomplish the next part of their task? It is possible that the action is hidden
   or that the terminology does not match what users are looking for. In either case, the
   correct control exists but users cannot find it. The existence and quality of labels on
   controls and the number of controls on the screen influence the user's ability to find
   an appropriate control (Franzke, 1995).
3. Once users find the control, will they know how to use it (e.g., click on it, double click,
   pull-down menu)? For example, if the control is a pull-down menu but it looks like
   a normal button, users may not understand how to use it. Users may find the icon
that corresponds to the desired action, but if it requires a triple-click they may never figure out how to use it.

4. If users perform the correct action, will they see that progress is being made toward completing the task? Does the system provide appropriate feedback? If not, users may not be sure that the action they just performed was correct.

C.1.3 Heuristic Walkthrough: Pass 2

C.1.3.1 Heuristics

Guided by the knowledge you gained in Pass 1, you are now free to explore any part of the system. Evaluate the system using each of the following heuristics, which are derived from Chapter 2. For your convenience, short summaries of each heuristic are included here.

**Preventing and Understanding Potential Attacks**

Information about how an attacker would exploit this vulnerability or what types of attacks are possible in this scenario.

*Notes go here

Does the tool satisfy users’ needs in this category? *Likert Scale*

<table>
<thead>
<tr>
<th>Likert Scale</th>
</tr>
</thead>
</table>

**Understanding Approaches and Fixes**

Information about alternative ways to achieve the same functionality securely.

*Notes go here

Does the tool satisfy users’ needs in this category? *Likert Scale*

<table>
<thead>
<tr>
<th>Likert Scale</th>
</tr>
</thead>
</table>
Assessing the Application of the Fix

Once a fix has been selected and/or applied, information about the application of that fix or assessing the quality of the fix.

*Notes go here

Does the tool satisfy users’ needs in this category? Likert Scale

---------------------------------

Relationship Between Vulnerabilities

Information about how co-occurring vulnerabilities relate to each other.

*Notes go here

Does the tool satisfy users’ needs in this category? Likert Scale

---------------------------------

Locating Information

Information that satisfies “where” questions. Searching for information in the code.

*Notes go here

Does the tool satisfy users’ needs in this category? Likert Scale

---------------------------------

Control Flow and Call Information

Information about the callers and callees of potentially vulnerable methods.

*Notes go here

Does the tool satisfy users’ needs in this category? Likert Scale

---------------------------------
**Data Storage and Flow**

Information about data collection, storage, its origins, and its destinations.

*Notes go here

Does the tool satisfy users’ needs in this category? *Likert Scale*

_____________________________________________________________

**Code Background and Functionality**

Information about the history and the functionality of the potentially vulnerable code.

*Notes go here

Does the tool satisfy users’ needs in this category? *Likert Scale*

_____________________________________________________________

**Application Context/Usage**

Information about how a piece of potentially vulnerable code fits into the larger application context (e.g., test code).

*Notes go here

Does the tool satisfy users’ needs in this category? *Likert Scale*

_____________________________________________________________

**End-User Interaction**

Information about sanitization/validation and input coming from users.

*Notes go here

Does the tool satisfy users’ needs in this category? *Likert Scale*

_____________________________________________________________
Developer Planning and Self-Reflection
Information about the tool user reflecting on or organizing their work.
*Notes go here
Does the tool satisfy users’ needs in this category? Likert Scale

Understanding Concepts
Information about unfamiliar concepts that appear in the code or in the tool.
*Notes go here
Does the tool satisfy users’ needs in this category? Likert Scale

Confirming Expectations
Does the tool behave as expected?
*Notes go here
Does the tool satisfy users’ needs in this category? Likert Scale

Resources and Documentation
Additional information about help resources and documentation.
*Notes go here
Does the tool satisfy users’ needs in this category? Likert Scale
Understanding and Interacting with Tools

Information about accessing and making sense of tools available. Including, but not limited to the defect detection tool.

*Notes go here

Does the tool satisfy users’ needs in this category? Likert Scale

Vulnerability Severity and Rank

Information about the potential impact of vulnerabilities, including which vulnerabilities are potentially most impactful.

*Notes go here

Does the tool satisfy users’ needs in this category? Likert Scale

Notification Text

Textual information that an analysis tool provides and how that text relates to the potentially vulnerable code.

*Notes go here

Does the tool satisfy users’ needs in this category? Likert Scale

Other Usability Problems / Notes