Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security

Justin Smith
Lafayette College
smithjus@lafayette.edu

Lisa Nguyen Quang Do
Google
lisanqd@google.com

Emerson Murphy-Hill
Google
emersonm@google.com

Abstract

Static analysis tools can help prevent security incidents, but to do so, they must enable developers to resolve the defects they detect. Unfortunately, developers often struggle to interact with the interfaces of these tools, leading to tool abandonment, and consequently the proliferation of preventable vulnerabilities. Simply put, the usability of static analysis tools is crucial. The usable security community has successfully identified and remedied usability issues in end user security applications, like PGP and Tor browsers, by conducting usability evaluations. Inspired by the success of these studies, we conducted a heuristic walkthrough evaluation and user study focused on four security-oriented static analysis tools. Through the lens of these evaluations, we identify several issues that detract from the usability of static analysis tools. The issues we identified range from workflows that do not support developers to interface features that do not scale. We make these findings actionable by outlining how our results can be used to improve the state-of-the-art in static analysis tool interfaces.

1 Introduction

Security-oriented static analysis tools, like Spotbugs [12], Checkmarx [2], and CodeSonar [3] enable developers to detect issues early in the development process. Among several types of code quality issues, developers rank security issues as the highest priority for these tools to detect [22].

Evaluating the efficacy of these security-oriented static analysis tools has been a popular topic for researchers [17, 29, 51, 63]. 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. Chess and McGraw argue that 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" [21].

Unfortunately, developers continue to make mistakes and need help resolving security vulnerabilities due to the poor usability of security tools [33]. Recently, Acar and colleagues set forth a research agenda for remedying usability issues in developer security tools, explaining that "Usable security for developers has been a critically under-investigated area" [14]. As part of that research agenda, they call for usability evaluations of developer security tools. This approach has been successfully applied in the adjacent field of end-user security tools [25, 30, 31, 58, 60]. For instance, Whitten and Tygar conducted cognitive walkthroughs to identify usability issues in PGP, an end user security tool [60]. We use a similar evaluation technique, namely heuristic walkthroughs [55] in combination with a user study, to identify usability issues in developers' security-oriented static analysis tools.

We evaluated four security-oriented static analysis tools, Find Security Bugs [13], RIPS [10], Flawfinder [7], and a commercial tool. (Our license agreement with the tool vendor stipulates that we anonymize the commercial tool; we refer to it as CTool throughout the paper.) To our knowledge, this study is the first to identify usability issues across multiple developer security tools using heuristic walkthroughs. As a result of this study, we identified several usability issues, ranging from missing affordances to interfaces that scale poorly. Each of these usability issues represents an 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 these

Copyright is held by the author/owner. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee.
USENIX Symposium on Usable Privacy and Security (SOUPS) 2020.
August 9–11, 2020, Virtual Conference.

tools, we can enable developers to create secure software by resolving vulnerabilities more accurately and efficiently. To support replication, we make our study setting available in a virtual machine [6], along with the study protocol [5] and the detailed list of usability issues we identified for the four tools [8].

The contributions of this paper are:

- A heuristic walkthrough and a user study evaluating the usability of the interfaces of four static analysis tools.
- A categorization of usability issues that serves both as a list of known pitfalls and as a list of opportunities to improve existing tools.
- Design guidelines and discussions that illustrate the actionability of these issues.
- Specifications for a low-cost heuristic walkthrough approach that researchers and practitioners can use to improve additional tools.

2 Related Work

We have organized the related work into two categories. First, we will discuss relevant work concerning usability evaluations of end user security tools. Next, we will discuss prior evaluations of developer security tools.

2.1 Usability Testing End-User Security Tools

Several studies have evaluated the usability of end user security tools. Through these evaluations, researchers identified usability issues in various end user security tools, ranging from encryption tools [49] to Tor browsers [25]. 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, 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 [60]. Their study revealed issues such as irreversible actions and inconsistent terminology.

Since the 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 [31]. 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 a heuristic evaluation and cognitive walkthrough [58]. Reynolds and colleagues conducted two studies to understand multiple aspects of YubiKey usability [52].

Clark and colleagues conducted cognitive walkthroughs to examine the usability of four methods for deploying Tor clients [25]. 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 [30].

Like these previous studies, we are concerned with the usability of security tools and strive to better understand usability issues by conducting an empirical evaluation. 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.

2.2 Evaluating 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 [63]. Comparing a wide range of Android security tools, Reaves and colleagues categorize tools according to the vulnerabilities they detect and techniques they use [51]. 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 detection techniques, including static analysis, with respect to the number of vulnerabilities found, false positive rates, and technique efficiency [17]. 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 [29].

There have been a limited number of studies that account for usability in their evaluations of developer security tools. Imtiaz and colleagues [38] study developer actions on Coverity warnings to determine how it helps fix bugs. They show that despite the quick fixes and the low complexity of the warnings, developers still take a disproportionately large amount of time to fix them. Assal and colleagues conducted a cognitive walkthrough evaluation of Findbugs to determine how well it helped developers determine the number of vulnerabilities in a codebase [16]. Based on the usability issues identified in this study, the authors created a tool, Cesar, designed to be more usable. Gorski and colleagues conducted a participatory design study of security warnings generated for cryptographic APIs [32]. They find that design guidelines for end-user warnings are insufficient in this context. Nguyen and colleagues describe some usability issues that affect Android lint tools to motivate the design of their tool, FixDroid, which uses data flow analysis to help secure Android apps [44]. However, the descriptions of Lint's usability issues are not based on a formal evaluation. Smith and colleagues conducted a user study which identified 17 categories of developers' information needs while using a security-oriented static analysis

tool [56]. Thomas and colleagues leveraged Smith and colleagues' framework to evaluate the usability of ASIDE, an interactive static analysis tool [57]. Our work differs from these prior studies because we study the usability (rather than the technical capability) of static analysis tools.

The studies closest to our own focus on the usage and usability of different analysis tools. Sadowski and colleagues describe the Tricorder static analysis ecosystem at Google [54]. They also provide guiding principles based on their experience with Tricorder. Some of these guidelines emphasize the importance of usable static analysis. For instance, they argue that analysis tools should fix bugs, not just find them. Johnson and colleagues [40] interview 20 developers on their experience with the static analysis tools they use at work. Christakis and colleagues [23] survey the developers at Microsoft about their usage of the tools, and report on live-site incidents to complete the survey. Lewis and colleagues [43] interview developers on two analysis tools at Google. Nguyen Quang Do [45] surveys 87 developers and analyzes the logs of static analysis tools at Software AG. Through those studies, the authors find common usability issues such as workflow integration, waiting times, bad warning explainability, and bad tool design. This is related to the findings of Imtiaz and colleagues. [37], who mined StackOverflow posts to discover that filtering and verifying false positives are major concerns of developers when using static analysis tools. Similarly, studies of static analysis tools, such as Parfait [24], focus on scalability and developer workflow and not on the user interface. While those studies report on general usability issues, we focus on tool design, and in particular, on the tool's user interface (which includes the Graphical User Interface, but also all functionalities provided to the user, e.g., generating reports).

3 Methodology

In this section we first justify our choice of tools and then describe the interfaces of those tools. Next, we describe the study environment, including the projects that each of the tools scanned. We also outline our approach toward conducting the heuristic walkthroughs and the user study. Finally, we provide a replication package.

3.1 Tools

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

We considered 61 candidate tools from lists of static analysis tools compiled by organizations and reserchers [64–67]. To narrow the selection of tools to use for our evaluation, we followed two criteria.

Table 1: Evaluated tool interface dimensions

Dimension	FSB	RIPS	Flaw-finder	CTool
Remediation Information[1]	✓	✓	✓	✓
Trace Navigation[2]	✓	✓		✓
Quick Fixes[3]	✓			
Graphical Rep. of Traces[4]		✓		✓
Command-Line Interface		✓	✓	
IDE Integration	✓			
Standalone Interface		✓		✓

[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;

The first one 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."[1]) Although most tool vendors we contacted were interested in the study in principle, only one agreed to participate in our study, under the condition that we anonymize the tool by not using screenshots, names, trademarks, or other distinguishing features in any publication. We thus refer to this tool as CTool (for Commercial Tool).

Second, 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 1 summarizes how FSB, RIPS, Flawfinder, and CTool vary along some of the interface dimensions we considered. Note that Table 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 [11].

Though definitive usage statistics are hard to find for these tools, it is 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.

Due to the *availability* constraint, our sample only includes one commercial tool. This introduces a potential threat that

[1] https://www.synopsys.com/company/legal/software-integrity/coverity-product-license-agreement.html
[2] https://sourceforge.net/
[3] Download statistics are not available specifically for FSB, which is a plugin for FindBugs

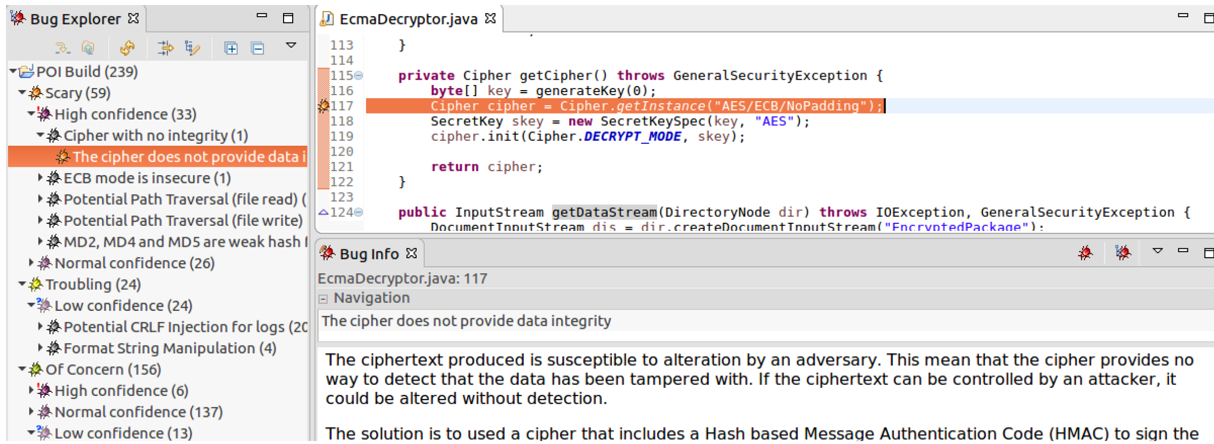


Figure 1: The Graphical User Interface of Find Security Bugs.

our sample may not represent tools used most frequently in the real-world. However, an empirical study examined the 20 most-popular Java projects on GitHub that use static analysis as part of their continuous integration pipeline [62]. They report that those projects commonly use open-source tools, like CheckStyle, FindBugs, and PMD. Reportedly, none use commercial analysis tools. Vassallo and colleagues' findings support this general trend in different contexts [59].

3.1.1 Find Security Bugs

Find Security Bugs (FSB) is an extension of the FindBugs/SpotBugs [12] static analysis tool for Java programs [13]. FSB detects 125 types of security vulnerabilities. We used the open-source Eclipse [4] plugin of the tool, version 1.7.1.

Figure 1 depicts FSB's GUI. The "Bug Explorer" pane on the left lists the potential vulnerabilities found by the most recent scan. Categories indicate the severity of the errors. In each category, errors are grouped according to FSB's certainty. Finally, vulnerabilities are grouped by type (e.g., "Cipher with no integrity"). When a user double clicks an error, the tool highlights the relevant lines of code in the editor. Tooltips for those icons provide information on all errors occurring at that particular line. In addition, the "Bug info" pane 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 trace of the vulnerability.

FSB also allows users 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 exclude particular source files from the scan, and to choose which analyses to run.

3.1.2 RIPS

RIPS [10] is a security static analysis tool for PHP code that detects more than 80 vulnerabilities. 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.

Figure 2 presents the main screen of RIPS. RIPS summarizes its results in the "Result" popup. Vulnerabilities are grouped by files and ordered by vulnerability type. RIPS provides a short description of each vulnerability alongside the problematic code. An icon on the left (not pictured) opens 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. When available, the help view explains vulnerabilities in more detail and sometimes suggest fixes. The generate exploit icon opens a view in which the user can generate an example exploit for this vulnerability.

The top menu of the page gives access to additional views that include a summary of the scan, a list of the program's functions, and a call graph illustrating which functions call each other. The user can rearrange the layout of the graph.

3.1.3 Flawfinder

Flawfinder [7] is a command-line tool that detects uses of dangerous functions in C/C++. We used version 2.0.4 of this open-source tool.

Figure 3 depicts Flawfinder's HTML report. The report lists all files that were scanned and all the vulnerabilities Flawfinder found, ordered by severity. For each error, Flawfinder provides the location of the error, the severity score, the vulnerability type, the dangerous function, a short description of the vulnerability, a link to the CWE page of the vulnerability, and a proposed fix—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

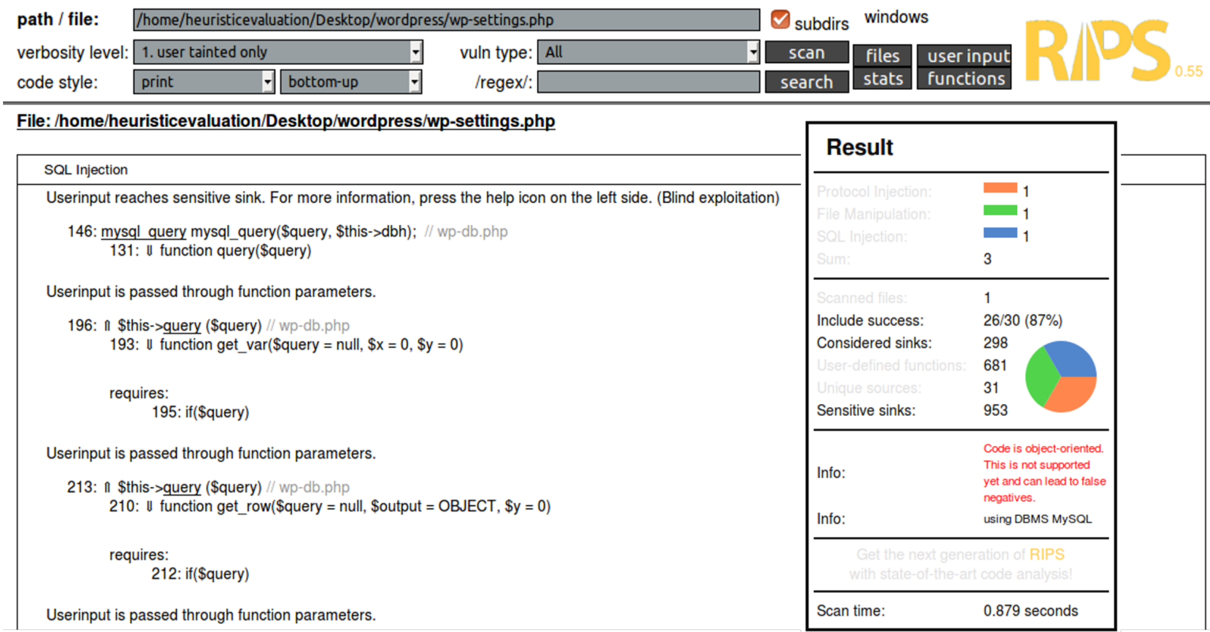


Figure 2: The Graphical User Interface of RIPS.

Flawfinder Results

Here are the security scan results from Flawfinder version 2.0.4, (C) 2001-2017 David A. Wheeler.
names) in C/C++ ruleset: 223

Examining openssl-1.0.1e/crypto/dso/dso_null.c
Examining openssl-1.0.1e/crypto/dso/dso_beos.c
Examining openssl-1.0.1e/crypto/dso/dso.h

...

Final Results

- openssl-1.0.1e/crypto/rand/randfile.c:86: **[5]** (race) *chmod: This accepts filename arguments; 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
Hits@level = [0] 2176 [1] 535 [2] 2245 [3] 59 [4] 108 [5] 2
Hits@level+ = [0+] 5125 [1+] 2949 [2+] 2414 [3+] 169 [4+] 110 [5+] 2

Figure 3: The Graphical User Interface of Flawfinder.

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 can be customized with command-line options to exclude files, or run on patchfiles, i.e. the diffs between two git commits. Errors can also be filtered out of the output with regex patterns.

3.1.4 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, 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. The creators of CTool provided us with the default web interface, which we used to scan Apache POI version 3.9 [1].

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 the selected warning. It shows its details in the source code, and sometimes suggests fixes. In this view, the user can comment on the warning and manage it (e.g., edit its priority). 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. 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 by choosing which checkers to run, and set code annotations that guide the analysis at runtime. Users can also annotate warnings and track scores throughout the development lifecycle across different runs.

3.2 Analyzed Applications

To ensure the evaluators could exercise the tools in a variety of situations, we chose subject applications that contained multiple types of vulnerability patterns. Synthetic suites, like

the Juliet test suite [19], could have helped us ensure this coverage. However, those hand-crafted tests do not represent how vulnerabilities appear to developers in the wild.

We selected test suites from the "Applications" section of the Software Assurance Reference Dataset [9], originating from production code bases containing known vulnerabilities.

We selected three test suites: 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 [6].

3.3 Heuristic Walkthroughs

We first identified usability issues by using a *heuristic walkthrough* [55], a two-phase method that combines the strengths of two usability evaluation techniques: cognitive walkthroughs [50] and heuristic evaluations [47]. 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 [55]. We chose this evaluation technique because heuristic walkthrough evaluations have successfully identified usability issues in other domains, such as electronic health records systems [28], music transcription tools [20], and lifelong learning systems [35]. Heuristic walkthroughs enable relatively few external evaluators [15] to identify usability issues by providing them with a structured process and requiring them to be double experts (experts in the application domain and usability principles). In compliance with these requirements, the two authors who conducted the heuristic walkthroughs 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. Not including the time required to configure each tool, each evaluator spent approximately one workday with each tool. The study protocol [5] and the detailed list of usability issues [8] are available online.

3.3.1 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. To do so, we used the following guidelines:

- Choose a vulnerability to inspect first.
- Determine whether it is a true positive or a false positive.
- Propose a fix to the vulnerability.
- Assess the quality of the fix.

To help us think critically about each tool, we used Sears' list of guiding questions [55]. 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.

3.3.2 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.

Among the many sets of available heuristics, we chose to use Smith and colleagues' 17 information needs [56] as the basis for our heuristics. We made this choice because the information needs are specific to security-oriented static analysis tools, in contrast with Nielsen's ten heuristics [46] or the cognitive dimensions framework [34], which are both more generic. These 17 information needs pertain specifically to security-oriented static analysis tools and cover a range of relevant topics such as: vulnerabilities, attacks, and fixes; code and the application; individuals; and problem solving support. Prior studies have shown that domain-specific heuristics can be more effective [27, 39, 48] and these particular heuristics have been used previously to evaluate a security static analysis tool [57]. Table 4 summarizes these heuristics.

The two evaluators considered each of the 17 information needs and recorded any usability issues that related to those information needs. 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 is organized according to these themes; each subsection describes one theme.

3.4 User Study

In the second part of our evaluation, we conducted a user study on the four static analysis tools, with the goal of triangulating the observations made in the heuristic evaluation. To this end, we recruited 12 participants, who we refer to as **P01–P12**,

with various degrees of professional experience as software developers. Participants answered questions on a Likert scale from 1 (novice) to 5 (expert) about their experience. Table 3 reports on their responses. In summary, participants self-reported familiarity with software security (median 3/5), Java (median 4/5), C++ (median 2/5), and PHP (median 1/5).

We presented each participant with two of the four static analysis tools, and asked them to fix warnings reported by the tools, using the same code bases as for the heuristic evaluation. Participants thought aloud while working with each tool for approximately 20 minutes. Following each tool, we conducted semi-structured post-task interviews. At the end of the session we collected demographic data. During the study, we allowed participants to ignore warnings they were uncomfortable with. This choice helped to account for differences in programming language skill. It also simulated real-world developers' strong propensity to selectively ignore most warnings [53, 61]. For all tools, all participants managed to find warnings they were comfortable with. In the post-task interviews, participants described their experience with the tools, focusing on how the tool helped them understand and fix warnings. Appendix C lists the questions we used to guide this discussion.

To reduce fatigue, we only asked participants to interact with two tools. Still, the mean session duration was 52 minutes, 58 seconds. To avoid learning effects between the two tools, we applied a latin-square design [26]. We distributed the tools evenly between participants—each tool was evaluated by six participants.

3.4.1 Data Extraction

We captured screen, audio, and questionnaire responses. Afterwards, we asked two independent researchers to review the audio recordings and extract the usability issues encountered by the participants. This yielded a total of 562 individual usability incidents. We kept the intersection of both reviewers' reported incidents, thus reducing the number of total usability incidents to 140. Two authors then classified the incidents into distinct usability categories using the open card sort methodology [36]. The classification yielded a Cohen Kappa of $\kappa = 0.93$, indicating an almost perfect agreement [41]. Afterwards, the two raters discussed and agreed on a final classification, which we present in Section 4.

3.5 Replication Package

To support the replication and continuation of this work, we have made our materials available, including the virtual machine image used during our heuristic evaluation. It contains the static analysis tools (we exclude CTool for legal reasons) and the code bases they were used on [6], the study protocol [5] and the list of usability issues we detail in the following section [8].

The user study protocols and heuristic walkthrough guide are also available in the Appendix.

Because usability 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 Results

Through our heuristic walkthrough evaluations and user study, we identified 194 and 140 usability issues, respectively. 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. For completeness, we provide the full list of usability issues in the supplemental materials [8].

In each section, we will give a general description of the usability issues relating to 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 during the heuristic walkthrough phase. 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 2 provides an overview of the themes and subthemes.

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

Table 2: Usability issues, grouped by theme

Theme	Subtheme
4.1 Missing Affordances	Managing Vulnerabilities Applying Fixes
4.2 Missing or Buried Information	Vulnerability Prioritization Fix Information
4.3 Scalability of Interface	Vulnerability Sorting Overlapping Vulnerabilities Scalable Visualizations
4.4 Inaccuracy of Analysis	
4.5 Code Disconnect	Mismatched Examples Immutable Code
4.6 Workflow Continuity	Tracking Progress Batch Processing

code, or have expertise fixing particular types of vulnerabilities. These three tools simply 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 3 depicts, in Flawfinder these "Final Results" are presented as a list that cannot be reorganized 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.

User Study: Most issues in this category impacted Flawfinder, where participants found the presentation of vulnerabilities (3) "irritating" (**P09**) and "poor" (**P11**). **P01** and **P12** were also confused by RIPS's presentation of vulnerabilities—**P12** suggested that it would be easier to make sense of the error messages if they were presented in a table.

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.

User Study: We did not identify any participants who faced these issues during our user study. One explanation for this may be that issues in this category only become apparent after working with a tool for an extended period of time, trying to apply several fixes.

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 [18], 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.

4.2 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.

User Study: Participants in the user study encountered similar issues to those we identified. **P04** complained that RIPS did not provide any severity or priority scores. Further, several participants (**P01**, **P04**, **P05**, **P08**, and **P10**) were confused by FSB and CTool's scales.

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.

User Study: Nearly all participants experienced issues as a result of insufficient fix information across all four tools. For instance, participants described the information that was provided as, "not helpful and too complicated" (**P05**), "generic" (**P06**), "unclear and very irritating" (**P09**), and "short" (**P11**).

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.

4.3 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 [17, 51, 63]. Each of the four tools we examined exhibited an interface scalability issue.

Vulnerability Sorting {Flawfinder}: As we previously discussed in Section 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.

User Study: Unsurprisingly, participants requested the ability to "sort the warnings by type, class, and significance."—**P11**

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 1 contains multiple overlapping vulnerabilities, however this is not perceptible without hovering over the bug icon (Figure 4). CTool includes a feature for displaying overlapping vulnerabilities, but this feature is turned off by default and is located in a somewhat hidden location.

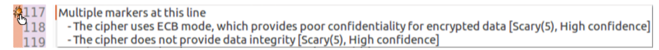


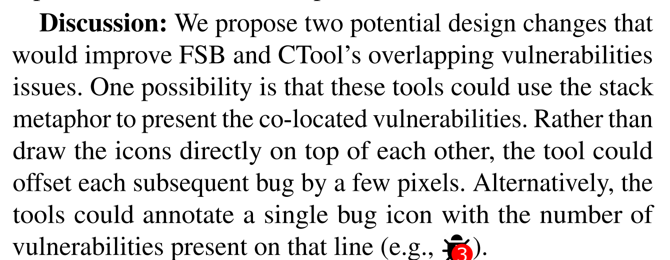
Figure 4: Instance of **Overlapping Vulnerabilities**

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.

User Study: Participants did not explicitly mention this issue during the user study, either because they did not inspect any overlapping notifications, or because they didn't notice that they were overlapping.

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 5. 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 6).

User Study: Only one participant (**P04**) experienced this issue with RIPS during the study, suggesting that the graph representation, "could be improved."

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 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., ).

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 [42], which enable developers to expand and highlight only the relevant paths through the graph.

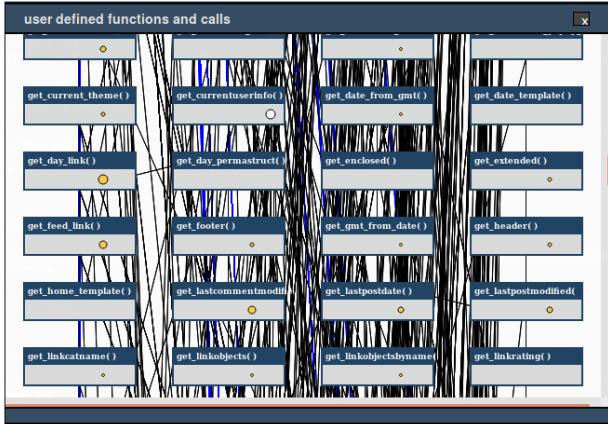


Figure 5: Scalability of RIPS' function view.

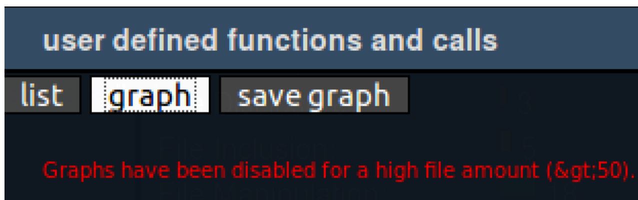


Figure 6: RIPS call graph visualization for more than 50 files.

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 7 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.

User Study: While using RIPS, participants (**P01**, **P06**, **P12**) encountered a similar unexpected behavior. RIPS summarizes results in a popup window (Figure 2). Frustratingly, this popup window obscures other information about the results and cannot be closed.

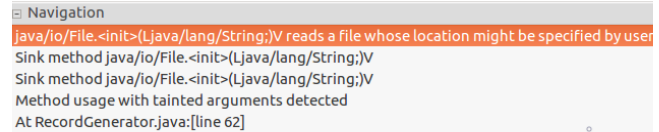


Figure 7: Duplicate entries in FSB's navigation feature

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 8 depicts one instance where FSB's examples are mismatched with the vulnerable code. In this case, FSB's "solution" example (Figure 8b) 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).

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

(a) Vulnerable code

```
Solution:
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
Byte[] cipherText = c.doFinal(plainText);
```

(b) FSB's example "solution"

Figure 8: Instance of **Mismatched Examples**

User Study: Several participants encountered similar issues with mismatched examples while using RIPS (**P04**, **P06**, **P10**, **P12**). Figure 9 illustrates a common issue. Here RIPS suggests that functions-compat.php contains an error on line 304, however, this file is only 155 lines long. The mismatched line numbers confused participants: "I am confused because the line number doesn't correspond to line in file."—**P10**

Userinput is passed through function parameters.

304: get_settings ($option); // functions-compat.php

Figure 9: RIPS line numbers mismatched with code.

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, but do not enable developers to directly 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. 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 vulnerability—like which line, module, and version it is contained in—so that they can fix the appropriate line.

User Study: Several participants in the user study verified this issue. They described switching between a code editor and the static analysis tools as, "disorienting" (**P10**), "cumbersome" (**P03**), and "time-consuming" (**P09**).

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.

4.6 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. These issues 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, some tools present their results in such a way that does not allow developers to track 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.

User Study: **P01** encountered this issue while using FSB; they were unable to verify that their change fixed the bug, "How do you check whether it has been fixed?" **P12** echoed **P01**'s frustration while using Flawfinder, "It would be nice if the user interface could track which errors have been added and removed. Now, I need to do all this work manually."

Batch Processing {FSB, RIPS, Flawfinder, CTool}: Secondly, all four tools we evaluated dictate that developers address notifications individually. This is problematic because projects can contain many occurrences of the same vulnerability patterns. For instance, CTool detected 30 occurrences of a package protection vulnerability in POI. Serially fixing these vulnerabilities is error-prone—developers must consistently apply the right fix to each occurrence. 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 developers, adding vulnerabilities to their work queues much faster than they dismiss them.

User Study: **P09** described this issue during the user study, "If the bugs are similar, you don't have option to fix them at the same time... I would like to fix them all at once."

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 provide developers with scan diffs. These would 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.

5 Design Guidelines

To help make our findings more actionable for toolsmiths, we provide a set of design guidelines:

- **Communicate what and how to fix.** Just locating potential vulnerabilities is insufficient. Static analysis tools should enable developers to diagnose (and, when it's technically feasible, automatically fix) the vulnerabilities they detect. Tools should either provide semi-automated quick-fixes or provide enough information for developers to manually fix detected vulnerabilities. We are inspired by FixBugs [18], which shows how human-in-the-loop fixes can improve automation beyond what's possible with quick-fixes. For even more complex problems, better explanations would be preferable. (Section 4.2)
- **Situate alerts within editable code.** Switching between a code editor and a tool's results page is disorienting and time-consuming. Tools should present their alerts where developers can directly modify problematic code. (Section 4.5)
- **Integrate with existing workflows.** Tools should integrate with developers' workflows. After developers fix potential vulnerabilities, tools should clearly communicate which problems have been fixed and which new problems have been introduced. (Section 4.6)
- **Generate contextualized notifications.** Tools should infer contextual information about the code they scan and use that contextual information to generate alerts and examples that match the current context. (Section 4.5) For instance, FSB's example in Figure 8 could be contextualized by parsing the vulnerable code and generating an alert with matching variable names and cyphers.

6 Limitations

First, we only evaluated four tools. To mitigate this threat to generalizability, we selected analysis tools that are each representative of distinct interface features (Table 1) and we included both a commercial tool and open-source tools. We do not claim that all of the usability issues we identified necessarily generalize to other analysis tools. For example, not all tools use ambiguous vulnerability severity scales like FSB does (Section 4.2). However, the themes and subthemes we report (Table 2) are more generalizable—all but two subthemes describe usability issues that span multiple tools. We expect that these *types* of issues detract from the usability of other tools. To further mitigate this threat, we also made our evaluation guide available. If researchers or toolsmiths are interested in understanding how these categories manifest in a particular tool, our approach can easily be applied to additional tools.

We also acknowledge that the evaluators' individual tool usage styles might have influenced the issues they identified. To bolster the ecological validity of our study, we selected real-world applications and instructed evaluators to perform realistic tasks. Additionally, we triangulate our findings through a user study with professional developers.

Another limitation of our study is that we examined datasets containing known vulnerabilities (Section 3.2). This choice guaranteed that scans would yield results, thus enabling us to examine how each tool presented vulnerabilities. However, static analysis users may face additional usability issues when searching for unknown vulnerability patterns.

Our choice of usability evaluation technique also influences our results. Compared with user studies, where qualified participants might spend less than an hour using a tool, evaluators conducting heuristic walkthroughs have more time to find deeper issues. On the other hand, 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 [27, 39, 48]. To mitigate this issue, we performed a user study with independent participants and independent reviewers, resulting in a classification that is different from the heuristics we chose. The results of the user study confirm most of our observations from the heuristic walkthrough. The user study was performed in a closed setting, within time limits. While it would be interesting to conduct a complementary study over several months in an industry setting, we note that the heuristic walkthrough already mitigates this limitation by allowing evaluators to spend more time and investigate the tools in depth. To avoid further external threats to validity, we recruited participants with an industry background.

7 Conclusion

This paper responds to a call for usability evaluations of *developer* security tools. In this work, we conducted a heuristic walkthrough and a user study to evaluate the usability of four static analysis tools: Find Security Bugs, RIPS, Flawfinder, and CTool. To enable similar evaluations to be conducted even more broadly, we have made our heuristics, study protocols and study setting available in a replication package.

This work reveals usability issues that detract from all four of the tools we examined. We discuss potential mitigations for those types of issues. To help toolsmiths design more usable static analysis tools for security, we present a set of design recommendations. We hope that our work enables practitioners to improve the usability of their tools and inspires researchers to evaluate the usability of developers' security tools.

Acknowledgments

We would like to thank our study participants for their time. We would also like to thank Maria Riaz, Caitlin Sadowski, Sarah Heckman, William Enck, Amy Ko, Adam Meade, and Kathryn Stolee for their feedback.

References

[1] Apache poi source code. https://poi.apache.org/subversion.html, 2018.

[2] Checkmarx home page. https://www.checkmarx.com/, 2018.

[3] Codesonar home page. https://www.grammatech.com/products/codesonar, 2018.

[4] Eclipse home page. http://www.eclipse.org/, 2018.

[5] Evaluation guide. https://figshare.com/s/087f103905189f1a7ca0, 2018.

[6] Evaluation vm. https://figshare.com/s/bf91e24a3df3c4fff77c, 2018.

[7] Flawfinder home page. https://www.dwheeler.com/flawfinder/, 2018.

[8] List of issues. https://figshare.com/s/71d97832ae3b04e0ff1a, 2018.

[9] Nist test suites. https://samate.nist.gov/SRD/testsuite.php, 2018.

[10] Rips home page. http://rips-scanner.sourceforge.net/, 2018.

[11] Security tool interface dimensions. https://figshare.com/s/5255acbe659d3097d8a2, 2018.

[12] Spotbugs home page. https://spotbugs.github.io/, 2018.

[13] Find security bugs home page. http://find-sec-bugs.github.io/, 2019.

[14] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *Cybersecurity Development (SecDev), IEEE*, pages 3–8. IEEE, 2016.

[15] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. Security developer studies with github users: Exploring a convenience sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 81–95, Santa Clara, CA, 2017. USENIX Association.

[16] Hala Assal, Sonia Chiasson, and Robert Biddle. Cesar: Visual representation of source code vulnerabilities. In *Visualization for Cyber Security (VizSec), 2016 IEEE Symposium on*, pages 1–8. IEEE, 2016.

[17] Andrew Austin, Casper Holmgreen, and Laurie Williams. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology*, 55(7):1279–1288, 2013.

[18] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 211–221, Oct 2016.

[19] T. Boland and P. E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 45(10):88–90, Oct 2012.

[20] Manuel Burghardt and Sebastian Spanner. Allegro: User-centered design of a tool for the crowdsourced transcription of handwritten music scores. In *Proceedings of the 2Nd International Conference on Digital Access to Textual Cultural Heritage*, DATeCH2017, pages 15–20, New York, NY, USA, 2017. ACM.

[21] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, November 2004.

[22] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 332–343, New York, NY, USA, 2016. ACM.

[23] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 332–343, New York, NY, USA, 2016. ACM.

[24] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, Manuel Valdiviezo, Andrew Browne, Jacob Zimmermann, Andrew Craik, Douglas Teoh, and Christian Hoermann. Static deep error checking in large system applications using parfait. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 432–435, New York, NY, USA, 2011. ACM.

[25] Jeremy Clark, Paul C. Van Oorschot, and Carlisle Adams. Usability of anonymous web browsing: an

examination of tor interfaces and deployability. In *Proceedings of the 3rd symposium on Usable privacy and security*, pages 41–51. ACM, 2007.

[26] Yadolah Dodge. *Latin Square Designs*, pages 297–297. Springer New York, New York, NY, 2008.

[27] Dean Julian Dykstra. *A Comparison of Heuristic Evaluation and Usability Testing: The Efficacy of a Domain-Specific Heuristic Checklist*. A & M University, Texas, 1993.

[28] Paula J. Edwards, Kevin P. Moloney, Julie A. Jacko, and François Sainfort. Evaluating usability of a commercial electronic health record: A case study. *International Journal of Human-Computer Studies*, 66(10):718 – 728, 2008.

[29] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.*, 217:5–21, July 2008.

[30] Kevin Gallagher, Sameer Patil, and Nasir Memon. New me: Understanding expert and non-expert perceptions and usage of the tor anonymity network. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 385–398, Santa Clara, CA, 2017. USENIX Association.

[31] Nathaniel S Good and Aaron Krekelberg. Usability and privacy: a study of kazaa p2p file-sharing. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 137–144. ACM, 2003.

[32] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. Listen to developers! a participatory design study on security warnings for cryptographic apis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[33] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[34] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[35] Xiaoqing Gu, Fengjia Gu, and James M Laffey. Designing a mobile system for lifelong learning on the move. *Journal of Computer Assisted Learning*, 27(3):204–215, 2011.

[36] William Hudson. The encyclopedia of human-computer interaction, 22. card sorting. https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/card-sorting, 2014.

[37] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 245–249, May 2019.

[38] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333. IEEE, 2019.

[39] Pooya Jaferian, Kirstie Hawkey, Andreas Sotirakopoulos, Maria Velez-Rojas, and Konstantin Beznosov. Heuristics for evaluating it security management tools. *Human Computer Interaction*, 29(4):311–350, 2014.

[40] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[41] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. page 159–174, 1977.

[42] Thomas D. Latoza and Brad A. Myers. Visualizing call graphs. In *in VL/HCC'2011: IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 18–22.

[43] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead. Does bug prediction support human developers? findings from a google case study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 372–381, May 2013.

[44] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writing secure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1065–1077. ACM, 2017.

[45] Lisa Nguyen Quang Do. *User-Centered Tool Design for Data-Flow Analysis*. PhD thesis, Paderborn University, 2019.

[46] Jakob Nielsen. 10 usability heuristics for user interface design. *Nielsen Norman Group*, 1(1), 1995.

[47] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, NY, USA, 1990. ACM.

[48] Tulsidas Patil, Ganesh Bhutkar, and Noshir Tarapore. Usability evaluation using specialized heuristics with qualitative indicators for intrusion detection system. In *Advances in Computing and Information Technology*, pages 317–328. Springer, 2012.

[49] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability smells: An analysis of developers' struggle with crypto libraries. In *Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*, SOUPS'19, page 245–257, USA, 2019. USENIX Association.

[50] Peter G. Polson, Clayton Lewis, John Rieman, and Cathleen Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741 – 773, 1992.

[51] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, et al. * droid: Assessment and evaluation of android application analysis tools. *ACM Computing Surveys (CSUR)*, 49(3):55, 2016.

[52] Joshua Reynolds, Trevor Smith, Ken Reese, Luke Dickinson, Scott Ruoti, and Kent Seamons. A tale of two studies: The best and worst of yubikey usability. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 872–888. IEEE, 2018.

[53] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

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

[55] Andrew Sears. Heuristic walkthroughs: Finding the problems without the noise. *International Journal of Human-Computer Interaction*, 9(3):213–234, 1997.

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

[57] Tyler Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson R Murphy-Hill. What questions remain? an examination of how developers understand an interactive static analysis tool. In *WSIW@ SOUPS*, 2016.

[58] Daniela G. tom Markotten. User-centered security engineering. In *Proceedings of the 4th EurOpen/USENIX Conference–NordU2002*, 2002.

[59] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, pages 1–39, 2019.

[60] Alma Whitten and J Doug Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *USENIX Security Symposium*, volume 348, 1999.

[61] Glenn Wurster and Paul C Van Oorschot. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, pages 89–97, 2008.

[62] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017.

[63] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.

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

[65] Owasp source code analysis tools. `http://owasp.org/index.php/Source_Code_Analysis_Tools`.

[66] Web application security consortium static code analysis tools. `http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList`.

[67] Static analysis tools for security. `https://www.dwheeler.com/essays/static-analysis-tools.html`.

A User Study Briefing

Thank you so much for agreeing to participate in this study. As I mentioned in the email, we are researchers interested in improving the usability of different security-oriented static analysis tools. This study will provide valuable information on how to design and develop better support for the use of static analysis tools.

Today I'll have you to use two static analysis tools and ask a few questions about your experience using them. During this session I'll be recording the computer screen and audio from our conversation. You can let me know at any point if you'd like to stop or pause the recording. If all that sounds ok to you, could you please sign this form for me...

Consent form: <obtain consent from participant>

B User Study Task Briefing

In this scenario, you have been tasked with evaluating the security of two applications. These applications have been analyzed by a static analysis tool that detects potential security vulnerabilities. I'll have you use the first tool for about 15-20 minutes, then I'll ask you a few questions. Then I'll have you use the second tool for about 15-20 minutes and I'll ask you a few more questions.

As you are working with the tools, try to think aloud. So, say any questions or thoughts that cross your mind regardless of how relevant you think they are. If you are silent for longer than 30 seconds or so, I'll gently remind you to KEEP TALKING.

(If participants are silent for more than 30 seconds raise a "KEEP TALKING" sign [Sugirin 1999])

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.

Do you have any questions before we begin?

(TURN ON SCREEN RECORDER, AUDIO RECORDER, AND BACKUP RECORDER!)

C Post-Study Questions

1. Which issues did you encounter when using the tool?
2. Which functionalities of the tool did you like most?
3. Which functionalities of the tool did you dislike most?
4. Which functionalities of the tool did you find useful?
5. Were there moments when you were confused?
6. Would you use this tool in your development work?

D Heuristic Walkthrough Guide

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:

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.

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.

Pass 2

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 Smith and colleagues' 17 information needs [56]. 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.
- Understanding Approaches and Fixes
  Information about alternative ways to achieve the same functionality securely.
- 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.
- Relationship Between Vulnerabilities
  Information about how co-occurring vulnerabilities relate to each other.
- Locating Information
  Information that satisfies "where" questions. Searching for information in the code.
- Control Flow and Call Information
  Information about the callers and callees of potentially vulnerable methods.
- Data Storage and Flow
  Information about data collection, storage, its origins, and its destinations.
- Code Background and Functionality
  Information about the history and the functionality of the potentially vulnerable code.
- Application Context/Usage
  Information about how a piece of potentially vulnerable code fits into the larger application context (e.g., test code).
- End-User Interaction
  Information about sanitization/validation and input coming from users.
- Developer Planning and Self-Reflection
  Information about the tool user reflecting on or organizing their work.
- Understanding Concepts
  Information about unfamiliar concepts that appear in the code or in the tool.
- Confirming Expectations
  Does the tool behave as expected?
- Resources and Documentation
  Additional information about help resources and documentation.
- Understanding and Interacting with Tools
  Information about accessing and making sense of tools available. Including, but not limited to the defect detection tool.
- Vulnerability Severity and Rank
  Information about the potential impact of vulnerabilities, including which vulnerabilities are potentially most impactful.
- Notification Text
  Textual information that an analysis tool provides and how that text relates to the potentially vulnerable code.
- Other Usability Problems / Notes

E Summary of Participants' Experience

Table 3: Summary of Participant Experience

ID	Tool 1	Tool 2	Security Familiarity	Java Familiarity	C++ Familiarity	PHP Familiarity	Professional Exp. (Years)
P00	FSB	Rips	●●●●●	●●●●●	●●○○○	●○○○○	7
P01	FSB	Rips	●●●○○	●●●●○	●●●○○	●○○○○	3
P02	FSB	Flawfinder	●●●○○	●●●○○	●●○○○	●○○○○	2.5
P03	FSB	CTool	●●○○○	●●○○○	●○○○○	●○○○○	4
P04	RIPS	CTool	●●●○○	●●●○○	●●○○○	●○○○○	3
P05	Flawfinder	CTool	●●●●○	●●●●○	●●○○○	●●○○○	7
P06	RIPS	FSB	●●●●○	●●●●●	●●●●○	●●○○○	1
P07	Flawfinder	FSB	●○○○○	●●●○○	●●○○○	●○○○○	0.5
P08	CTool	FSB	●●●○○	●●●●○	●●●●○	●●○○○	3
P09	Flawfinder	Rips	●●○○○	●●●○○	●●○○○	●○○○○	6
P10	CTool	Rips	●○○○○	●●○○○	●●○○○	●●○○○	3
P11	CTool	Flawfinder	●●●○○	●●●●○	●●○○○	●○○○○	0
P12	RIPS	Flawfinder	●●●○○	●●●●○	●●●●○	●●●●○	8

F Heuristics Summarized

Table 4: Summary of heuristics from Smith and colleagues [56]

Heuristic	**Description**
Preventing & Understanding Potential Attacks	Information about how an attack would exploit this vulnerability or what types of attacks are possible in this scenario.
Understanding Alternative Fixes & Approaches	Information about alternative ways to achieve the same functionality securely.
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.
Relationship Between Vulnerabilities	Information about how co-occurring vulnerabilities relate to each other.
Locating Information	Information that satisfies "where" questions. Searching for information in the code.
Control Flow & Call Information	Information about the callers and callees of potentially vulnerable methods.
Data Storage & Flow	Information about data collection, storage, its origins, and its destinations.
Code Background & Functionality	Information about the history and the functionality of the potentially vulnerable code.
Application Context / Usage	Information about how a piece of potentially vulnerable code fits into the larger application context (e.g., test code).
End-User Interaction	Information about sanitization/validation and input coming from users. Does the tool help show where input to the application is coming from?
Developer Planning & Self-Reflection	Information about the tool user reflecting on or organizing their work.
Understanding Concepts	Information about unfamiliar concepts that appear in the code or in the tool.
Confirming Expectations	Does the tool behave as expected?
Resources & Documentation	Additional information about help resources and documentation.
Understanding & Interacting with Tools	Information about accessing and making sense of tools available. Including, but not limited to the defect detection tool.
Vulnerability Severity & Rank	Information about the potential impact of vulnerabilities, including which vulnerabilities are potentially most impactful.
Notification Text	Textual information that an analysis tool provides and how that text relates to the potentially vulnerable code.