Discovering Buffer Overflow Vulnerabilities In The Wild: An Empirical Study

Ming Fang
Auburn University
Auburn, AL, USA
mzf0018@auburn.edu

Munawar Hafiz
Auburn University
Auburn, AL, USA
munawar@auburn.edu

ABSTRACT

We performed an empirical study on reporters of buffer overflow vulnerabilities to understand the methods and tools used during the discovery. The participants were reporters featured in the SecurityFocus repository during two six-month periods; we collected 58 responses. We found that in spite of many apparent choices, reporters follow similar approaches. Most reporters typically use fuzzing, but their fuzzing tools are created ad hoc; they use a few debugging tools to analyze the crash introduced by a fuzzer; and static analysis tools are rarely used. We also found a serious problem in the vulnerability reporting process. Most reporters, especially the experienced ones, favor full-disclosure and do not collaborate with the vendors of vulnerable software. They think that the public disclosure, sometimes supported by a detailed exploit, will put pressure on vendors to fix the vulnerabilities. But, in practice, the vulnerabilities not reported to vendors are less likely to be fixed. Ours is the first study on vulnerability repositories that attempts to collect information from the people involved in the process; previous works have overlooked this rich information source. The results are valuable for beginners exploring how to detect and report buffer overflows and for tool vendors and researchers exploring how to automate and fix the process.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; A.1 [**Introductory and Survey**]; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Experimentation, Measurement, Security.

Keywords

Secure Software Engineering, Vulnerability, Empirical Study.

1. INTRODUCTION

When Beth, a beginner, wants to find information about how to detect buffer overflow vulnerabilities, she can pursue many sources of information. She can search the Internet, ask her peers, read books and research papers on detecting overflows, or even train to become an ethical hacker. She will be easily overwhelmed by the number of "best practices" suggested and the number of suggesters claiming that their methods work best without backing the claims with empirical data. A researcher who wants to improve the security engineering process to fix the reported vulnerabilities more efficiently will also need empirical data to understand which problems to target. A similar information gap confuses tool builders who want to build popular, widely-adopted tools.

Researchers who have studied security engineering process focused on finding statistics of vulnerability trends, lifecycle of vulnerabilities, time to fix a vulnerability, etc [14–16, 26, 27]. These studies have not focused on the approaches of the people who report buffer overflow vulnerabilities, i.e., the *tools they use and methodologies they follow.* Also, previous studies explore the tip of an iceberg; they miss a lot of activities going on behind vulnerability detection and reporting. *The people involved in these activities carry first-hand information about corresponding security approaches.* The knowledge that they possess is not stored in vulnerability repositories or any other repositories. No attempts have been made to tap into this rich information source.

This paper describes a study performed on *reporters* of buffer overflow vulnerabilities at SecurityFocus repository [29]. These participants reported buffer overflow vulnerabilities that were featured in the repository during two six-month periods (Dec 2011 - May 2012, Dec 2012 - May 2013). We ran the study in the form of an email questionnaire. Our analysis is based on the responses of 58 reporters (out of 229 with contact information, 25.33% response rate). The group of responders included reporters who have discovered a vulnerability for the first time as well as reporters who have reported hundreds of vulnerabilities, reporters who have a few months experience as well as reporters with over ten years of experience, reporters working on source code as well as reporters working on binaries, and reporters working on a Windows platform as well as reporters working on a Unix platform. We asked them about the *methods* they followed to detect and report buffer overflow vulnerabilities and the *tools* they used. The questions were open-ended. We analyzed the responses by applying a structural coding approach [25] using Text Analysis Markup System Analyzer (TAMS Analyzer) [32], an open source tool for coding.

We analyzed the responses in the two rounds of the study separately and aggregated the results for common trends. Our study focused on the approach taken by the developers to explore for buffer overflows in software, the tools that they use, the way they report vulnerabilities, and the effort they put on creating exploits after detecting vulnerabilities. We found that there is a common method that reporters follow to detect buffer overflow vulnerabilities. Despite the extensive research on static and dynamic analysis tools to detect buffer overflows, almost none of these tools are used. Instead, reporters use fuzzing tools to find vulnerabilities and then use a few sophisticated debuggers to find root causes. Also, our study reveals the needs and problems of the vulnerability discovery process. We found that experienced reporters are uninterested in reporting vulnerabilities to vendors or creating detailed exploits. The lack of interest in reporting vulnerabilities to vendors indicates a communication gap. Experts report publicly because they think that it will pressure the vendors into fixing the vulnerabilities; sometimes they spend additional time to develop detailed exploits to add to the pressure. But, we found that vulnerabilities that were reported publicly (even those with detailed exploit code) were less likely to be fixed than vulnerabilities reported to vendors. The vulnerabilities publicly reported by our study participants along with detailed exploit code have remained unfixed for an average of 774 days; only 1 of 14 such vulnerabilities were fixed.

Our goal was to study what people do to detect buffer overflows, find the common trends, and capture their essence. This paper makes the following contributions.

- To the best of our knowledge, it describes the first study to collect information about practices from the people involved in discovering vulnerabilities: the most reliable, yet previously ignored, source of information. The data is collected from 58 reporters of buffer overflow vulnerabilities that were featured in the SecurityFocus repository during two six-month periods.
- It describes a research methodology (Section 3) that can be reused by researchers who want to understand the discovery process of other security vulnerabilities.
- It presents information about what actually happens during the detection, analysis, and reporting of buffer overflow vulnerabilities (Section 4). It suggests which approaches are used, which work, which do not, and what needs to change (Section 5).

2. BACKGROUND

This section describes previous work on studying vulnerability repositories, and studies on human subjects that focus on understanding security engineering tools and practices.

2.1 Studying Vulnerability Repositories

Studies on vulnerability repositories focus on harvesting statistical trends [4, 16, 26, 27, 30] or creating vulnerability models and using them for prediction [1, 2, 10]. Our study focuses on the reporters who possess the most important information; no previous works have explored this source.

Schneier [26] first focused on the entire window of exposure to a vulnerability and proposed a life cycle model for vulnerabilities; later Arbaugh and colleagues [4] proposed a modified model. Most of the studies of vulnerability repositories consider this life cycle model (or an adapted version) and reach statistical conclusion on various aspects, e.g., time to create a patch. Arora and colleagues [5] studied 308 vulnerabilities to find out how efficiently vendors respond to vulnerability reports. Frei and colleagues [15] studied Microsoft and Apple patches from CERT, Secunia, SecurityFocus, and some other repositories to measure the time to produce a typical patch. Frei et al. [14] later proposed a modified vulnerability life cycle model. They introduced definitions of important time periods, e.g., time of patch availability, time of patch installation, etc.

Other researchers concentrated on the trends in vulnerabilities. Gopalakrishna and Spafford [16] studied CVE reports of five well-known software in order to find a relationship of vulnerability types reported on the same software. Their results suggest that the discovery of a vulnerability in a software may influence the discovery of the same type of vulnerabilities in that software. Scholte and colleagues [27] studied the National Vulnerability Database (NVD) to analyze how cross site scripting and SQL injection vulnerabilities evolved over time in web applications. Wu and colleagues [34] studied how semantic templates can be overlaid on vulnerability repositories to harvest more systematic information; this can be useful for further trend analysis. Shahzad and colleagues [30] studied 46,310 vulnerability reports reported between 1998 and 2011 collected from three sources (including NVD and Open Source Vulnerability Database, OSVDB) and identified trends such as the prominence of buffer overflow and DoS vulnerabilities, the increase of remotely exploitable vulnerabilities, etc.

Many researchers have focused on creating vulnerability models and using them for prediction. Browne and colleagues [10] first studied CERT database for vulnerabilities reported on some well-known software in search of a trend of exploitations. Anbalagan and Vouk [2] proposed a model to describe the relationship between security vulnerabilities and security exploits. Alhazmi and colleagues [1] present a vulnerability discovery model and predict the number of vulnerabilities for several operating systems. In contrast, Zhang et al. [36] applied data mining and machine learning technologies to unsuccessfully find a trend in vulnerabilities reported in the NVD database. They reported that data in NVD generally have poor prediction capability, except for a few vendors and software applications.

2.2 Understanding Security Engineering Tools and Practices

A few researchers have studied human factors of security engineering. Schryen [28] surveyed a collection of client and server side software to analyze the patching behavior of developers. Okhravi and Nicol [22] studied browser vulnerabilities to understand how much pre-deployment testing is optimal. Aranda and Venolia [3] did not study security vulnerabilities. They studied collaboration behavior of Microsoft developers to fix bugs. Layman and colleagues [19] conducted a user study to design better fault detection tools.

There have been a few studies comparing tools and approaches to detect vulnerabilities. Wilander and Kamkar [33] studied four compiler based approaches that detect and prevent buffer overflow vulnerabilities and reported that these tools only provide a partial solution. Studies on automated penetration testing tools on web vulnerabilities have also reported that existing tools missed many vulnerabilities [6,12, 31]. Among these works, Austin and colleagues [6] reported that a combination of different approaches (manual analysis, static analysis, and automated penetration testing) is necessary to discover vulnerabilities. These studies imply that some approaches are better than others in detecting vulnerabilities; they do not focus on whether the tools and approaches are used by people in practice.

Our study found that reporters rarely use static analysis tools when they search for vulnerabilities. This justifies more work on understanding the challenges of using static analysis in practice. Works by Johnson and colleagues [17], Baca and colleagues [7], and Rutar and colleagues [24], explored the usability challenges of static analysis tools. These works focused on the needs of secure software developers, not on the specific needs of reporters of security vulnerabilities.

3. RESEARCH METHODOLOGY

This section narrates our research methodology in detail. We describe the scope of the study, how the participants were selected, which questions were asked, how we collected data, and how we analyzed the responses.

3.1 Scope of the Study

We conducted our study on all reporters of buffer overflow vulnerabilities that were featured in the SecurityFocus repository during two six-month periods (December 1, 2011 to May 31, 2012, and December 1, 2012 to May 31, 2013). We selected all vulnerabilities with 'buffer overflow' in its name. Not all of these vulnerabilities were new: SecurityFocus lists vulnerabilities reported earlier if some new information is added to the listing.

Choosing the right repository is important for the success of the study. Massacci and Nguyen [20] studied the quality of a vulnerability database by measuring how often it is used by other security researchers.

We chose SecurityFocus [29] repository because it is well-known (Section 4.4.4), it is highly ranked in Massacci and Nguyen's [20] study, and we have used this repository in previous research. Massacci and Nguyen also report that it is one of the most widely used repositories in empirical studies only behind the vulnerability list kept by National Vulnerability Database (NVD). We chose SecurityFocus over NVD because it contains more vulnerabilities than NVD. For example, SecurityFocus listed 516 vulnerabilities with at least one reporter during the study period, some new and some updates to old vulnerabilities. Only 390 (75.58%) had a Common Vulnerability Enumeration (CVE) identifier; these were in NVD. Choosing SecurityFocus allowed us to contact with more people and potentially collect more responses.

3.2 Participants

There were 623 buffer overflows featured in SecurityFocus during the study period. 29 do not have any reporter associated, while 78 vulnerabilities are reported by vendors of the corresponding software. We could not find any contact person for these. The remaining 516 vulnerabilities attribute at least one reporter; some have two or three reporters.

There are 332 unique reporters who reported the 516 vulnerabilities. Among them, 238 reported a single vulnerability during the study period. The remaining 94 reporters were associated with more than one vulnerability.

The vulnerability reports attribute reporters, but do not store their contact information. We searched for the contact information from various sources, e.g., multiple vulnerability repositories, security advisories, emails in vendor webpage where bugs were reported, etc. In many cases, we searched for the reporters on the Internet using their names.

We could not identify the email address of 36 reporters. 42 others had Twitter or LinkedIn addresses only, but no email address. We removed these 78 reporters and sent email questionnaire to the remaining 254 reporters. 25 emails were bounced; the contact information that we found was incorrect. Among the remaining 229 reporters, 58 responded to our study (58/229 ≈ 25.33% response rate).

Out of the 58 reporters who responded, 37 reported only one vulnerability during our study; 21 reported more than one vulnerability. Three reporters reported more than fifteen vulnerabilities during the study period.

Table 1: Study Questions For Participants

Questionnaire (Round 2)
[Round 1 had questions 1 and 2, and questions 8,9,10 merged as 2 questions]
1. Is this the first security vulnerability that you have reported? If not, how many previous vulnerabilities have you reported?
2. How long have you been interested in security vulnerabilities? Do you do this as a hobby, or is it your professional responsibility?
3. Please comment on your familiarity with the application. Were you affiliated with the development process of the application, or were you an end user?
4. Why did you select this application? Was it a random selection? Do you have some methods that help you select a target application?
5. Please describe the circumstances under which this vulnerability was detected. Was the detection of this vulnerability a coincidence or were you specifically searching for vulnerabilities?
6. Did you have access to the source code when you discovered the vulnerability? Or were you working on binaries only?
7. Can you please describe the steps you followed while detecting the vulnerability? Do you follow these same steps for other buffer overflow (or related) vulnerabilities you may have detected?
8. Were any tools used to discover this vulnerability? Please name them and describe how these tools were used to detect this vulnerability.
9. If you used tools, why did you choose these tools?
10. How did you first find out about these tools? Research paper? Web Search? Suggestion from a peer?
11. Have you used other tools for detecting other instances of buffer overflow (or related) vulnerabilities? Please name them. Why were they not used here?
12. Did you prepare an exploit demonstrating the vulnerability? If yes, can you provide a general description of the process? What tool(s) did you use? How much time did it take?
13. Which forum did you use to report the vulnerability and why did you select this particular forum?
14. Did you search for additional vulnerabilities in the same application once this vulnerability was detected? Did you find/report any more?

3.3 Questionnaire

The first round of the study had 11 open-ended questions. During the second round, we refactored the questions, i.e., we asked the same questions but broke some into separate questions. This round had 14 questions.

Table 1 lists the questions in the second round. Questions 1 and 2 asks about a participant's background. Questions 3–7 and 14 focus on the methodology followed by a reporter, while question 8–11 focus on the tools used. Questions 12 and 13 explores reporting and exploit generation tasks.

3.4 Process

We conducted the first round of the study between May 2012 and November 2012. Then we replicated our own study (Feb–Jul, 2013). We sent emails to the reporters whose address could be found. The emails were sent as we processed the information about vulnerabilities and identified reporters. Typically 10-15 emails were sent per week. We sent a reminder typically once a month and stopped after three reminders. Some reporters were featured in both rounds of survey. We did not send any email to a reporter in the second round if he/she responded to the survey in the first round; everyone else received emails.

The email we sent contained the questions and a reference to a specific vulnerability. We referred to the vulnerability with the BugTraq ID (BID) used in SecurityFocus and the CVE identifiers used by NVD, when applicable. For reporters of multiple vulnerabilities, we listed all vulnerabilities reported by the person in the email, but asked him/her to focus on the last vulnerability reported.

Pilot Study. Before the first round, we launched a small pilot study to train ourselves with the process and finalize the questions to ask. We selected 10 reporters of buffer overflow vulnerability from the SecurityFocus repository during the first week of November 2011. Each reporter received the initial version of the questionnaire containing 9 questions.

We received 4 responses. The responses helped us adapt our study. For example, one responder suggested that we include CVE identifiers of a vulnerability as well as the BugTraq ID (BID) that SecurityFocus uses to identify a vulnerability. Also, we restructured the questions during the first round and later in the second round.

3.5 Coding

We applied structural coding [25] to annotate the responses. Having specific and well-defined categories to code also reduced interpreter's bias. The codes we selected had clear partitions. For example, when a responder says that he/she has six years of experience, and we had codes that defines experience as 'less than one year', 'one to less than five years', 'five to less than ten years', and 'more than ten years', there is only one code that can be applicable.

We used TAMS Analyzer [32], an open source tool, for coding as well as analyzing the codes. Both authors applied codes to all responses in TAMS Analyzer. To ensure reliability, their codes were compared. Because of structural coding, the codes mostly matched (initial Krippendorff's alpha [18] value was 0.8339). The coders analyzed the codes that differed. Some mismatches were from one coder missing a code. Most mismatches were because of the limits of TAMS Analyzer: if two codes were applied to the same paragraph in a different order, TAMS Analyzer identified them as different. We resolved all the issues to reach consensus.

4. RESULTS

Most of the results presented in this paper come from applying analytic generalization [35]. While we present some statistical information about the tools and methods used by reporters (statistical generalization [35]), these only summarize the findings. We present the results as aggregates of the two rounds of study. More details of the results of individual rounds are in the project webpage: `http://www.munawarhafiz.com/research/scienceofsecurity`.

We formulated four categories of research questions to structure the study and the questions we asked the reporters.

- **RQ1: General Approach:** Are vulnerabilities discovered by chance or is there a general approach that reporters follow? How does a reporter select a target application? Are there different approaches when people work with source code or binary?
- **RQ2: Use of Tools:** Are there tools that are commonly used? Why and how are they used? How is the information about these tools disseminated among reporters?
- **RQ3: Vulnerability Reporting:** How and where are the vulnerabilities reported? Do reporters work with the vendors to fix vulnerabilities? What are the consequences of the choices made by reporters?
- **RQ4: Exploit Generation:** Do reporters write exploit code after they detect vulnerabilities? What types of tools are used to create exploit code?

We first describe the distribution of participants. Then we describe the findings in the four categories of research questions—each will be described with a separate heading.

4.1 Distribution of Participants

This section describes that the participants represent reporters with different experience levels (years of experience), levels of success (vulnerabilities reported), types of involvement (professional or hobbyist), and working contexts (type of target software, operating system, etc.,). This validates the analytic generalizability of the responses. Also, we can analyze the data from different perspectives, e.g., how do experts do X (reporting practices, Section 4.4), or what do developers who target Unix-based systems do for X (tools used by reporters, Section 4.3), etc.

4.1.1 From one to twenty one years of experience

Of the 58 participants, 43 responded to the question about their experience with a precise number of years. The group includes beginners as well as experts. Only 2 reporters have less than one year of experience, while 13 reporters have between one and five years of experience. In contrast, 15 have between five and ten years of experience, while 13 reporters ($13/43 \approx 30.23\%$) have more than ten years of experience. One reporter has 21 years of experience, another one has 18 years, and three have 15 years of experience.

4.1.2 Single to a thousand vulnerabilities

48 reporters responded to the question about the number of vulnerabilities reported. Some study participants are very experienced. We have 8 reporters who reported more than a hundred vulnerabilities. In fact, one of them publicly reported over thousand vulnerabilities. Most of them, 20 in total, reported more than ten vulnerabilities. 12 others said they had reported fewer than ten. Only 8 reporters said that the vulnerability in question was their first.

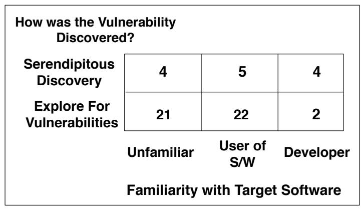
4.1.3 A blend of professionals and hobbyists

We had 53 responses about whether the reporters consider vulnerability discovery as a hobby or as a profession. Most of the reporters who responded are professionals working on security vulnerabilities—34 of them ($34/53 \approx 64.15\%$). Some of them are experts: 8 have over ten years of experience. 10 professional developers identified themselves as software developers who have security as a secondary interest. 9 identified themselves as hobbyists.

4.1.4 Windows and Unix, uncritical and critical

In total, the 58 responders worked on 55 different software. Among these, 32 are Windows-based and 20 are Unix-based. 3 have both Windows and Unix versions.

Our responders have worked on operating systems such as Microsoft Windows and Linux; popular software such as Microsoft Visual Studio and Microsoft .NET framework; driver software such as the driver for BlazeVideo HDTV Player; and systems that protect critical infrastructure, e.g., GE Energy D20/D200 Substation Controller.



How was the Vulnerability Discovered?

	Unfamiliar	User of S/W	Developer
Serendipitous Discovery	4	5	4
Explore For Vulnerabilities	21	22	2

Familiarity with Target Software

Figure 1: Context of Discovery

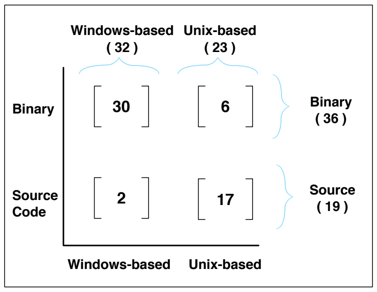


Figure 2: Platforms and Code

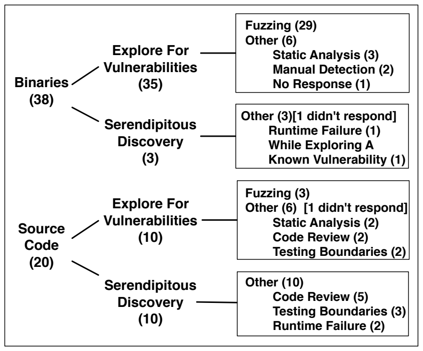


Figure 3: Popularity of Fuzzing

4.2 RQ1: General Approach

In this section, we report about the general approach of the reporters. This includes whether reporters actively search for vulnerabilities, how reporters choose target software, and whether they follow a common method.

4.2.1 *Reporters start blindly*

We distinguish between three levels of familiarity—a reporter of a vulnerability can (1) be a part of the software's development team (*developer*), (2) select a software that he/she uses (*user*), or (3) select a software from some source to explore a vulnerability (*unfamiliar*). Surprisingly, a lot of the reporters (25, 25/58 ≈ 43.10%) fell in the third category; *they decided to work on the target software just out of curiosity, boredom, or hunch.* One of them said, "I just downloaded an evaluation version on a rainy and boring day." Another reporter said, "I have never used the program. I don't even know exactly what it does ... The only thing in which I was interested was the fact that there were files with registered extension and so a real security scenario."

A lot of reporters selected a target software from amongst the software used by them (27, 27/58 ≈ 46.55%). But only a few reporters fell in the developer category (6, 6/58 ≈ 10.34%). It is surprising that only a few reporters are affiliated with the development process. This is perhaps because the vulnerabilities discovered by developers are internally fixed as bugs; they are never reported to repositories.

4.2.2 *Reporters actively probe popular targets*

Figure 1 matches the reporters' initial familiarity with a software with the circumstances of finding a vulnerability. It reveals several trends:

(1) *Vulnerabilities are not discovered serendipitously; reporters actively explore for them.* Even most of the serendipitous discoveries are a result of reporters exploring for vulnerabilities in some other parts of the code. For example, 4 reporters serendipitously discovered the vulnerabilities when they were testing the software; 5 reporters found the vulnerabilities while doing code reviews (Figure 3). Sometimes, a 'lucky' runtime failure leads to a vulnerability, but they are less common (3 in Figure 3).

(2) *Reporters explore vulnerabilities in software that they download from the Internet or in software that they use.*

The target selection process is not entirely random. We asked reporters about how they select an application to target; 20 reporters responded. Most respond that their choice is influenced by *the popularity of target software* (11, 11/20 ≈ 55%). Some reporters additionally mentioned that they target software that has *high visibility* (4, 4/20 ≈ 20%), e.g., SCADA systems. Sometimes reporters have a *hunch about some software being easy targets* (5, 5/20 ≈ 25%), such as a network service ("Network facing daemons are desirable targets") or a software written in a particular language ("I mainly choose software written in C / C++ as a target application. In this language there are many ways to make a mistake, and most of them are likely to happen if the programmer is not experienced enough."). Only one person responded that the choice of the application was random.

4.2.3 *Windows binaries and Unix source code*

Of the 58 reporters, 55 respond to the questions about the code artifacts and the platforms they worked on.

There is a strong correlation between the platform and the code artifact on which the reporters work (Figure 2). *Reporters exploring Windows-based software mostly work on binaries and reporters exploring Unix-based software mostly work on source code.* Since developers generally adopt a fuzzing approach (Section 4.2.4), familiarity with the software or access to source code is not necessary. Many responses support this. For example, one reporter found a vulnerability in Adobe Flash on the Unix platform. He replied, "We do have access to the source code to some degree, as Adobe has open sourced its ActionScript virtual machine. However, because fuzz testing is a black box approach, we don't need that kind of details to accomplish our task."

4.2.4 *Fuzzing is the chosen method*

The responders unanimously agree that *fuzzing is the most suitable method for detecting a buffer overflow vulnerability* (Figure 3). Developers exploring binaries almost exclusively use fuzzing. Fuzzing is also a common approach for reporters working on source code, but not as exclusively.

The reporters suggest that the general approach to explore for a buffer overflow vulnerability is to run a fuzzer and find a failure. Then one should use a debugger to trace through the fault and identify a buffer overflow situation. At this point, a vulnerability can be reported.

If a beginner is interested in creating an exploit, specifically on the Windows platform, (s)he can follow the recipe of one of the reporters: "1. Find a bug; 2. Examine the memory registers to see what it looks like; 3. See if it overwrites EIP (Extended Instruction Pointer) or SEH (Structured Exception Handler); 4. Determine how much space you have to work with. This will dictate if you have to use an egghunter or some sort of stack alignment technique; 5. Find any bad chars; 6. Tidy things up and complete the exploit".

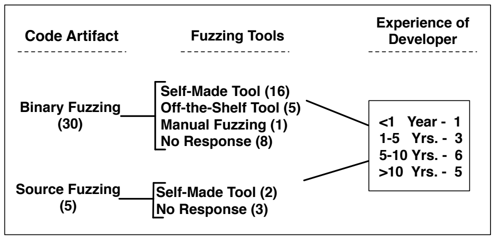


Figure 4: Custom Fuzzing Tools

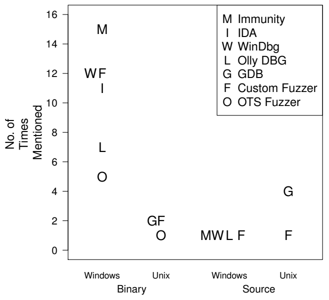


Figure 5: Tools In Use

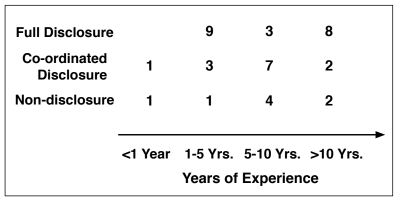

Full Disclosure 9 3 8
Co-ordinated Disclosure 1 3 7 2
Non-disclosure 1 1 4 2
<1 Year
1-5 Yrs.
5-10 Yrs.
>10 Yrs.
Years of Experience

Figure 6: Reporting Practices

4.2.5 Applying static analysis is uncommon

Surprisingly, *reporters did not follow static analysis approaches to detect vulnerabilities.* Figure 3 shows that only 5 reporters used static analysis. 2 other reporters said that they did not use static analysis for the specific case, but they used static analysis for detecting other buffer overflows.

There seems to be a general consensus about adopting fuzzing. One reporter who applied static analysis on binaries even admitted, “I’m one of the few that rarely fuzz”.

It may happen that developers use static analysis during writing and testing their code and they were not represented in the study. Indeed, developers are a minority in our study—only 6 reporters were directly involved with developing the vulnerable software that were reported (Figure 1). However, only one of them used static analysis. In contrast, 3 developers discovered overflows during code review.

4.2.6 Code review and other manual approaches are useful in detecting vulnerabilities in source code

Reporters working on source code often detected vulnerabilities manually. Among the 20 reporters working on source code, 7 detected overflows during code reviews (Figure 3). Only one of the vulnerabilities—a buffer overflow in the Linux kernel—was detected through a systematic code review process. Two vulnerabilities were detected when the reporters were working on fixes for other bugs in the same code; both of these reporters were developers of the software. In the remaining 4 cases, reporters were trying to understand third-party code when they found the vulnerabilities.

In 5 instances, reporters detected buffer overflow vulnerabilities by trying to systematically test the target software (Figure 3). 3 reporters tested the target software based on an assumption that a critical feature may harbor vulnerabilities (serendipitous discovery). For example, one reporter said, “I was looking at how HTTP headers were added to a string and started to wonder how the boundary checks worked... I had seen a similar vulnerability reported against the project a few months prior, which made me wonder whether there were other unchecked strings.” In 2 other cases, reporters were exploring the software to detect buffer overflows, but they did not use fuzzing; instead they wrote test cases to simulate the bug using debuggers in the process.

4.3 RQ2: Use of Tools

Following the general method discussed in Section 4.2.4, fuzzing and debugging tools are primarily used by reporters. In this section, we report about whether reporters use off-the-shelf tools and whether they use a few common tools.

4.3.1 Off-the-shelf fuzzing tools are typically not used

Although fuzzing tools are more common, *people typically do not use off-the-shelf tools; they prefer making their own fuzzers* (Figure 4). These fuzzers target various protocols (SSH, IMAP, etc.). Other tools target complex data types. For example, one reporter wrote a fuzzing tool targeting programs that take as input textual data that is regulated by a grammar, such as regular expressions. Interestingly, this is the only tool written by a researcher; it was written for a research work that was published at a security conference.

Both new and expert reporters build fuzzing tools; but it is more common for experts to build such tools (Figure 4). For new reporters, the experience is a part of the learning process (“I wanted to see if I could do it , that’s the only reason”, “Although ‘reinventing the wheel’ is not everyone’s thing, I feel it helps me understand how tools and the protocols truly work”). The main reason, however, is because *perhaps the existing off-the-shelf tools do not meet the specific needs.*

18 reporters developed custom fuzzing tools. Most (14) of the custom tools were ad hoc; they were used once. Three reporters mentioned that they reuse their fuzzing tools; however, they added that the tools were developed in their respective organizations. Their original intent was to find good case studies for their tools. One other reporter, who works at Google, used an internally developed fuzzer.

Among the off-the-shelf fuzzing tools, reporters mostly mentioned open source tools (SPIKE, COMRaider, Radamsa, Sulley, and Peach). One reporter said that “off-the-shelf tools typically cost money and independent researchers strive for higher ROI, especially when resources are low.”.

4.3.2 Immunity Debugger, IDA Pro, and WinDbg in Windows platform, GDB in Unix platform

Reporters mentioned 51 different debuggers; Figure 5 shows the debuggers and other tools mentioned. *Among Windows-based debuggers, Immunity Debugger, IDA Pro, and WinDbg are the most common.* Immunity Debugger, a powerful tool for writing exploits, is mentioned the most times as a debugger (mentioned 15 times). One of its extension packages written in Python, named Mona, is mentioned 4 times. IDA Pro, a Windows-based commercial tool, is mentioned but mostly as a tool for exploit generation. *Among Unix-based tools, GDB is the chosen debugger* (mentioned 6 times).

Different reasons are given by reporters about why they use Immunity Debugger; we had 7 reporters commenting on this. All of them suggested that Immunity Debugger is very easy to use. Four developers mentioned that Immunity Debugger is the most useful because of the Mona plugin (`mona.py`)[1]. Other responses suggest that IDA Pro and WinDbg are perhaps more versatile and used for more complex jobs; however they are more difficult to use ("For more complex reversing jobs, I'm using IDA.", "I SHOULD be using WinDbg, that is the most powerful debugger, but it is also the hardest to use."). Most reporters in the study who used IDA Pro and WinDbg were security professionals with over 5 years of experience. On the other hand, the experience of users of Immunity Debugger ranged from one year to more than 10 years. Thus the reasoning that Immunity Debugger is easier to use has some truth in it.

4.3.3 Static analysis tools are almost never used

Reporters prefer fuzzing-based blind search over more systematic white box exploration methods. That is why static analysis tools have limited applicability during vulnerability detection (Figure 3). One reporter mentioned Coverity Security Advisor tool (commercial), but not as a tool he used to detect the vulnerability in the study. Instead, he mentioned that he had used Coverity for detecting other vulnerabilities. Another reporter mentioned a custom tool called Kint, while a third reporter mentioned Flawfinder, an open source tool that searches for multiple security weaknesses. One other reporter mentioned that he uses static analysis for development work, but uses fuzzing for vulnerability detection—this seems to be the common trend.

11 reporters mentioned using IDA Pro (Figure 5). IDA Pro has multiple features—it is a disassembler with various static analysis, a powerful debugger, and a decompiler. However, all the reporters mentioned that they used IDA Pro as a debugger. A few mentioned using the WinDbg plugin available in IDA Pro, because they were familiar with IDA Pro, but also wanted WindDbg features.

4.4 RQ3: Vulnerability Reporting

This section explores how and where do reporters report, and what are the consequences of their decisions. We distinguish three reporting practices: (1) *non-disclosure* (contact with vendor only), (2) *coordinated disclosure* (contact with vendor first, then make public), and (3) *full disclosure* (publicly release vulnerability without contacting the vendor).

4.4.1 A lot of reporters prefer full disclosure

54 reporters answered about their reporting activity.

A lot of reporters adopt the full disclosure approach and post to vulnerability lists (28, 28/54 $\approx$ 51.85%). *Other reporters communicate with vendors: some release the vulnerabilities later in a mailing list (co-ordinated disclosure), others do not (non-disclosure).* The study results in both rounds suggest that coordinated disclosure (16, 29.63%) is a little more popular than non-disclosure (10, 18.52%).

Figure 6 shows the reporting preference in terms of the experience of the reporters. There is no clear trend here, but it shows that full disclosure is preferred by all kinds of reporters. However, reporters do show some concern about the consequences of full disclosure. Some suggest that they contacted the vendors, but the vendors never replied.

4.4.2 Experienced reporters have a follower group

We have 13 reporters who have more than ten years of experience, of whom 12 answered the question about reporting. We wanted to track how they reported vulnerabilities.

[1]From Jan 2013, Mona is also available with WinDbg.

These expert reporters differ in their way of reporting. However, most of them have their own follower bases; they choose reporting options with high visibility.

The first reporter mentioned that he used to keep track of Bugtraq, but is "no longer doing it, mainly for lack of interest." Instead, he uses Twitter to disclose vulnerabilities. The second reporter said that he does not check forums anymore. Two other reporters use Twitter, including one who used to work as a manager at a well-known vulnerability repository. The fifth reporter said that for the particular vulnerability he reported, he directly contacted the security lead. Another reporter contacted Microsoft to report a bug, because it provided high visibility. The seventh one prefers reporting through ZDI for monetary incentives. The eighth one disclosed the vulnerability, which is about a SCADA system, at a security conference to get public attention ("I chose this forum because the vendor showed no interest in fixing security problems. I wanted to pressure the vendor ... via press exposure."). The ninth person only reported to a repository established and maintained by him. Two other reporters were developers of the target applications.

Only one reported to a vulnerability repository. This reporter is a software professional (see Section 4.1.3), not a security professional. Therefore, he does not have the usual follower base that experienced security professionals have.

4.4.3 Fully disclosed vulnerabilities may remain unfixed

Experienced reporters justify full disclosure as a way to put pressure on vendors to fix vulnerabilities. They expect vendors to pay attention to their activity ("now I use Twitter when I like to show the releasing of a vulnerability that deserves some attention."). In practice, it may not work.

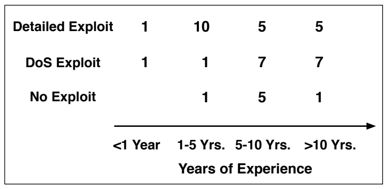
Surprisingly, *if reporters, both beginners and experts, disclose publicly but do not contact vendors, the vulnerabilities are likely to remain unfixed.* Figure 9 shows that 20 of the 27 publicly disclosed vulnerabilities remain unfixed (74.07%). We determined whether a vulnerability remains unfixed by searching for patch information at multiple sources: SecurityFocus, NVD, Secunia, OSVDB repositories, vendor webpages, etc. Each of the repositories periodically update by collecting information from the vendors and other repositories. Hence, if a vulnerability gets fixed, the information is likely to be available from these sources.

Even if the reported vulnerabilities have detailed exploit code, most remain unfixed (13 out of 14 in Figure 9). To date (March 24, 2014), these 13 vulnerabilities have remained unfixed for an average of 774 days since they were reported. Some of these are important software, such as a Windows graphics driver of nVIDIA (unfixed 260 days), or a SCADA system controller of GE (522 days). There are five CVEs associated with these 13 vulnerabilities; the average Common Vulnerability Scoring System (CVSS) scores [21] for these vulnerabilities are high (9.3 out of 10) with high average impact sub-score (10 out of 10) and high average exploitability sub-score (8.6 out of 10). Thus leaving the vulnerabilities unfixed may have severe impact. The only fixed vulnerability was fixed by Microsoft.

4.4.4 Exploit Database (EDB) is the most popular

Reporters mention 55 repositories in their answers.

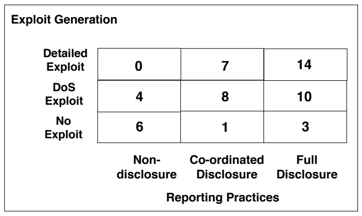
Most favor Exploit Database (EDB); it is mentioned 16 times (29.09%). All reports to EDB had some exploit code;



	<1 Year	1-5 Yrs.	5-10 Yrs.	>10 Yrs.
Detailed Exploit	1	10	5	5
DoS Exploit	1	1	7	7
No Exploit		1	5	1

Years of Experience

Figure 7: Exploit Generation



Exploit Generation	Non-disclosure	Co-ordinated Disclosure	Full Disclosure
Detailed Exploit	0	7	14
DoS Exploit	4	8	10
No Exploit	6	1	3

Reporting Practices

Figure 8: Reporting with Exploits

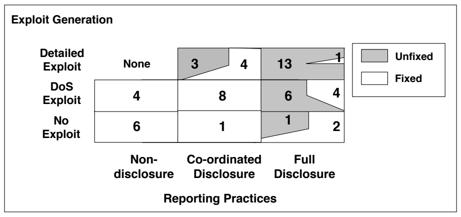


Figure 9: Which Problems Are Fixed?

12 had detailed exploit code (Section 4.5). Other repositories such as SecurityFocus (11 times), Secunia (6 times), OSS-Security (6 times), and CERT (4 times) are mentioned. 5 reporters posted to ZDI, because of commercial interest.

4.5 RQ4: Exploit Generation

This section explores whether reporters value the effort on writing exploits. We distinguish three kinds of exploits: (1) no exploit code written, (2) a simple denial-of-service (DoS) exploit that crashes the vulnerable application, and (3) a detailed exploit that takes control of the computer system.

4.5.1 *Reporters typically write exploits*

We have 57 reporters answering questions about exploits. 47 of them ($47/57 \approx 82.46\%$) prepared some exploit code to the vulnerability report. Half of these reporters prepared exploits that only crashed the application (24, $24/57 \approx 42.1\%$ overall). Other reporters created detailed exploits that executed the reporters code (23, $23/57 \approx 40.35\%$ overall). 10 reporters did not prepare any exploit code (17.54%).

4.5.2 *Perception of exploit generation effort varies*

Reporters, as they get more experienced, think that their time is more well-spent in finding new vulnerabilities as opposed to writing detailed exploit codes ("I'm not interested in demonstrating code execution at 100% because my hobby is just finding the vulnerability."–Reporter with 12 years experience). They often write exploits only to demonstrate the crashes (DoS exploit, Figure 7). On the other hand, beginners take exploit generation as a challenge by itself, which, they believe, will enhance their knowledge. These professionals regard exploit generation with high esteem. They were the more enthusiastic bunch; their responses were very detailed especially in the part where they answered how they generated the exploits. Other new reporters aspire to become an exploit developer. For example, a reporter who had three years of experience but reported only one vulnerability replied about why he did not write an exploit code, "I'm not competent enough to do so."

We investigated the 5 reporters with over ten years of experience who wrote exploits (Figure 7). Two mentioned doing it as their job responsibility. One reporter developed an exploit because the target was highly visible. Another reporter demonstrated the exploit in a security conference. Therefore he spent the time to develop it; otherwise he would not have developed a detailed exploit ("it was my first Metasploit module"). The fifth reporter developed the exploit in 30 minutes only, perhaps reusing some existing code (since detailed exploit development takes time). Thus these reporters had some incentives in developing detailed exploits.

4.5.3 *Exploits are written to make public*

Figure 8 matches the exploit generation and reporting practices of reporters. It shows that reporters who spent time on exploit development eventually released the exploit code, either independently or after co-ordinating with vendors. In fact, when reporters wrote detailed exploits, they always released exploit code.

Conversely, full disclosure almost always has some exploit code—minimal or detailed. When a reporter reports to a vendor only, his/her reports may contain a minimal exploit, but not a detailed exploit.

5. DISCUSSION AND RECOMMENDATIONS

Our study had two significant results: (1) We found that reporters mostly use fuzzing despite the fact that researchers and tool vendors have concentrated on static analysis approachess. (2) We also found a problem in the way vulnerabilities are reported: reporters do not coordinate with vendors and their choice to publicly disclose vulnerabilities, often with detailed exploits, is not useful for vendors to react and produce patches in time. We first discuss these issues.

Fuzzing vs static analysis—Understanding the divide. Security vulnerabilities are not only detected by independent professionals and hobbyists (we refer to them as 'non-developer reporters') who are interested in finding flaws in random software, but are also detected by the developers who are directly affiliated with the software. Their goals are different. Non-developer reporters are interested in quickly finding isolated flaws in target software; on the other hand, developers are interested in finding and fixing all flaws in the code they write. This is why their choices differ.

Most of our study participants were non-developer reporters (Section 4.2.1). They favor fuzzing since it is easy to adopt and it quickly finds bugs ("a quick fuzzing test was enough to crash the application"). Also, the vulnerability detected is instantly reproducible. Reporters may spend more time on exploits or quickly move on to the next target. Since non-developer reporters want to maximize their effort, they prefer a quick approach, as long as it is effective. Fuzzing suits their needs perfectly.

However, there are static analysis tools that are widely used to detect buffer overflows during software development [8]. Perhaps, the developers are not well-represented in our study. However, even among the few developers, static analysis approaches were not mentioned much (Section 4.2.5). It may also happen that developers relate static analysis with coding and testing and not necessarily with security analysis and vulnerability detection. Hence they did not mention it in this context. That saying, recent research has also focused on the impediments of adopting static analysis. Johnson and colleagues [17] have identified several factors that developers dislike about static analysis tools—lot of false positives, difficult I/O, hard to understand workflow, etc. These factors may actually hinder developers from using static analysis tools; static analysis may actually be used less than desired.

The Practice of Reporting Vulnerabilities. Most reporters in our study prefer posting to lists, i.e., full disclosure (Section 4.4.1); in fact they often post to multiple lists. Posting to vendors of vulnerable software appears to be a rational choice, but it is not followed in general, especially when reporters become more experienced (Section 4.4.2). When these publicly disclosed vulnerabilities with detailed exploits remain unfixed (Section 4.4.3), the situation gets worse.

There is a genuine problem in the way reporters and vendors communicate. Several reporters mentioned that they went public after they found that the vendors were non-responsive ("... the developers should have received my message informing about this issue. Several weeks have passed since then, and there is still no official fix. Maybe they didn't read my message, but if that is not the reason; I just have to say that sadly it seems the security of the users is not a priority unless the lack of it causes bad reviews in the Internet."). They consider public disclosure, sometimes with detailed exploits, as a way to pressure the vendors. However, vendors' silence does not mean they are uninterested. Perhaps, vendors have stopped updating this software or have brought a new version. Often, they are overwhelmed by the number of problems reported and take a lot of time to fix [14, 15]. In the meantime, vulnerability information in public domain may undermine software security. In fact, publicly disclosed vulnerabilities increase the chance of a software being targeted by attackers [5].

Recommendations. Synthesizing what we have learned, we now present concrete recommendations for reporters of vulnerabilities, secure system developers, project managers, security engineering researchers, and tool vendors.

(1) *Reporters should apply fuzzing approaches to detect buffer overflows.* Since fuzzing is widely considered to work well in practice, all reporters—developers as well as non-developers—should embrace fuzzing approaches.

(2) *Reporters should follow a combination of approaches to detect overflows.* They should adopt manual and automated fuzzing, static analyses, even manual exploration. Sections 4.2.4–4.2.6 support this. This also follows Austin and colleague's [6] result that a combination of multiple approaches, including manual exploration, is useful for detecting web vulnerabilities.

(3) *Project managers should use code review.* Open source systems should invest on introducing systematic code review. One of our recent works demonstrate the usefulness of code reviews to detect security vulnerabilities [9]. This study also hints at its usefulness (Section 4.2.6).

(4) *Researchers and tool vendors should focus on improved and reusable fuzzing tools.* Reporters who developed custom fuzzing tools mentioned unanimously that their tools were simple and easy to build (Section 4.3.1). Perhaps, this engineering challenge does not appeal to researchers. But this should change.

(5) *Reporters should report vulnerabilities to vendors.* Vulnerabilities reported directly to vendors (non-disclosure and coordinated disclosure) get fixed quickly and early [5]. Vendors may find it hard to follow the many channels for full disclosure, e.g., numerous public forums, blogs, Twitter, etc., thus leaving the vulnerability unfixed (Section 4.4.3). Coordinated disclosure provides a middle ground between helping vendors and getting recognition, and it should be adopted more.

(6) *Vendors should respond more quickly to reported buffer overflow vulnerabilities.* Reporters expressed a general dissatisfaction about their experience in communicating with vendors (Section 4.4.2). Vendors should look into this issue and react. Recent studies suggest that incentives provided by vendors, such as Vulnerability Rewards Programs, have been successful [13]. This may improve reporters' perception towards reporting to vendors.

(7) *Detailed exploits should only be disclosed to vendors.* Fully disclosed vulnerabilities with detailed exploits in public domain may be dangerous. The experienced reporters typically do not make this mistake. Among the 28 reporters with over five years of experience, only 4 created detailed exploit code and released in public. On the other hand, 9 relatively inexperienced reporters did this. New reporters want to get more attention: therefore they show their expertise by publicly releasing exploits. Experts look for exposure too: but they take a more measured approach. Therefore, reporters should report detailed exploits only to the vendors.

(8) *Beginners should be taught about how to report vulnerabilities.* Publicly disclosing exploits is dangerous, a consequence that beginner reporters fail to understand (See previous recommendation). Reporting practices should be an integral part of a beginner's education.

6. THREATS TO VALIDITY

There are several threats to validity of our study; here we discuss them following the four classic tests and discuss how they have been mitigated.

External Validity. There may be a concern about generalizability of our results. This will happen if our sample of 58 responders is not be representative of the larger population of reporters. Our group of responders have diverse experience and background (Section 4.1). The participants possess enough diversity to allow analytic generalization. Moreover, we collected responses from random reporters over two six-month periods (Section 3.2), thereby replicating our own study. When we explored for trends, we treated the data collected from the two rounds separately and reported a trend only if there is a similarity. Another possible threat comes from the lack of the interaction with the responders. For example, when a responder mentions a tool, it is unclear whether he/she is aware of another. However, sample size, the distribution of experience, and the detail in the responses somewhat counter this threat.

Internal Validity. Internal validity is mainly a concern for explanatory studies. Since ours is not an explanatory study, it does not have a threat to internal validity from the interpretation aspect. One problem could have come from interpreting the responses of reporters who reported a single vulnerability (specific response) versus reporters who reported multiple vulnerabilities during the study (general response). Reporters were always asked about a specific vulnerability discovery; for multiple reporters we asked for the last vulnerability (Section 3.4). Another problem could have arisen from efficiently handling and analyzing the results. We used TAMS Analyzer, an automated tool, which greatly reduced the data storage and interpretation problems.

Construct Validity. The success of our study is dependent on asking the right questions. We performed a pilot study to adapt the questionnaire (Section 3.4). Although the pilot study was small, it helped us significantly change the design. During the second round, we rewrote some of the questions to improve clarity (Section 3.4); this was done based on our experience from analyzing the responses of the first round.

Reliability. Because we dealt with open ended questions and answers, there is always a chance of misinterpretation of data. There are three reasons why this is not an issue. First, the coding was done in two phases, once per question and once per document, to improve the reliability of codes. Second, we used structural coding [25] with codes that had clear partitions (Section 3.5); this reduces the chance of coding biases. Most importantly, coding was done by the two authors (investigator triangulation [11, 23]). We calculated Krippendorff's alpha [18] value on the initial code; it was high (0.8339). Even then, the two coders analyzed the codes that differed and reached consensus.

7. FUTURE WORK AND CONCLUSION

Ours is the first study to collect information from a previously untapped source. It finds important details about how reporters detect buffer overflows, how reporters analyze their findings, and how they report detected vulnerabilities. In the future, we want to expand on the study and include interviews to get qualitative information about each of the phases. We also plan to apply our study to understand trends of detecting other important variants of security vulnerabilities, e.g., SQL injection, cross site scripting, etc.

Our results can help Beth, the beginner, decide that she needs to use fuzzing, perhaps use Immunity Debugger, spend time on writing detailed exploits, etc. It will also advise her that it is best to coordinate with vendors when reporting vulnerabilities, even if the experts tell her not to.

8. REFERENCES

[1] O. Alhazmi and Y. Malaiya. Prediction capabilities of vulnerability discovery models. In *RAMS'06*. IEEE Computer Society, 2006.

[2] P. Anbalagan and M. Vouk. Towards a unifying approach in understanding security problems. In *ISSRE'09*. IEEE Press, 2009.

[3] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE '09*. IEEE Computer Society, 2009.

[4] W. Arbaugh, W. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–59, Dec. 2000.

[5] A. Arora, R. Krishnan, R. Telang, and Y. Yang. Impact of vulnerability disclosure and patch availability - An empirical analysis. In *WEIS '04*, 2004.

[6] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *ESEM '11*, 2011.

[7] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg. Improving software security with static automated code analysis in an industry setting. *Software—Practice and Experience*, 43(3):259–279, 2013.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, Feb. 2010.

[9] A. Bosu, J. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Submission to FSE '14*, 2014.

[10] H. Browne, W. Arbaugh, J. McHugh, and W. Fithen. A trend analysis of exploitations. In *IEEE S&P '01*. IEEE Computer Society, 2001.

[11] N. Denzin. *The Research Act: A Theoretical Introduction to Sociological Methods*. McGraw-Hill, New York, 1978.

[12] A. Doupé, M. Cova, and G. Vigna. Why Johnny can't Pentest: An analysis of black-box web vulnerability scanners. In *DIMVA '10*. Springer, 2010.

[13] M. Finifter, D. Akhawe, and D. Wagner. An empirical study of vulnerability rewards programs. In *USENIX Security' 13*. USENIX Association, 2013.

[14] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. Modelling the security ecosystem- The dynamics of (in)security. In *WEIS '09*, 2009.

[15] S. Frei, B. Tellenbach, and B. Plattner. 0-day patch - Exposing vendors' (In)security performance. BlackHat Europe, 2008.

[16] R. Gopalakrishna and E. Spafford. A trend analysis of vulnerabilities. Technical report, CERIAS, 2005.

[17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE '13*. ACM, 2013.

[18] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage Publications Ltd, Singapore, 2004.

[19] L. Layman, L. Williams, and R. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *ESEM '07*. IEEE Computer Society, 2007.

[20] F. Massacci and V. Nguyen. Which is the right source for vulnerability studies?: An empirical analysis on mozilla firefox. In *MetriSec '10*. ACM, 2010.

[21] P. Mell, K. Scarfone, and S. Romanosky. CVSS: A complete guide to the Common Vulnerability Scoring System Version 2.0. Technical report, FIRST.org, 2007.

[22] H. Okhravi and D. Nicol. Evaluation of patch management strategies. *International Journal of Computational Intelligence: Theory and Practice*, 3:109–117, 2008.

[23] M. Patton. *Qualitative Research & Evaluation Methods*. Sage Publications Ltd, Singapore, 3 edition, 2001.

[24] N. Rutar, C. Almazan, and J. Foster. A comparison of bug finding tools for Java. In *ISSRE '04*. IEEE Computer Society, 2004.

[25] J. Saldana. *The Coding Manual for Qualitative Researchers*. Sage Publications Ltd, Singapore, 2009.

[26] B. Schneier. Full disclosure and the window of exposure. *Crypto-Gram Newsletter*, Sep 2000.

[27] T. Scholte, D. Balzarotti, and E. Kirda. Quo vadis? A study of the evolution of input validation vulnerabilities in web applications. In *FC'11*. Springer-Verlag, 2012.

[28] G. Schryen. A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors. In *IMF*, 2009.

[29] SecurityFocus. Bugtraq vulnerability list. http://www.securityfocus.com/.

[30] M. Shahzad, M. Shafiq, and A. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *ICSE '12*. IEEE Press, 2012.

[31] L. Suto. Analyzing the effectiveness and coverage of Web application security scanners. Technical report, eEye Digital Security, 2007.

[32] M. Weinstein. TAMS Analyzer for Macintosh OS X: The native open source, Macintosh qualitative research tool.

[33] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS '03*. The Internet Society, 2003.

[34] Y. Wu, R. Gandhi, and H. Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *SESS '10*. ACM, 2010.

[35] R. Yin. *Case Study Research: Design and Methods*. Sage Publications Ltd, Singapore, 2004.

[36] S. Zhang, D. Caragea, and X. Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *DEXA '11*. Springer, 2011.