The Erich Spengler Student Paper Award Winner for 2014

A Structured Approach to Student-Discovered Bugs and Vulnerability Disclosure

James Sullivan, Michael E. Locasto

University of Calgary

Abstract - There is a high demand for software developers and security professionals with strong software analysis skills. Currently, many students learn software analysis as an auxiliary exercise to their programming projects, and their experience is limited to white-box testing of applications that they or their peers have written. This type of experience does not give students a realistic or practical set of skills which they can immediately apply to more complex tasks. We describe our experiences with an information security course project in which students were tasked with discovering and analyzing software flaws in real software projects, giving students practical experience in flaw analysis and bug reporting. We discuss the focuses and goals of this project, including its emphasis on responsible disclosure, and the trends in student's comfort with analysis techniques and tools.

1. INTRODUCTION

Quality testing and security auditing have become key steps in the software development cycle. A successful large-scale software project will extensively and pervasively test its codebase to uncover flaws in their software, particularly those with security implications. While a number of analysis tools exist to aid developers in identifying bugs, it is still a learned skill to determine the root cause of a flaw, or to create a useful bug report to aid another developer, and the industry expects students to be versed with these skills.

Many students learn software testing only as an auxiliary exercise to software development, and testing is often limited to operation on expected input. Furthermore, most computer science students only perform white-box application testing during their education, and most of the time this is of their own work. It is a fundamentally more challenging problem to black-box test applications, or to work in unfamiliar codebases. The skills needed for both of these acts are in high demand in the industry, for security professionals and for developers alike.

Software testing requires a fundamentally different frame of mind than development does. During development, the focus is on making the software work as intended. On the other hand, the goal in software testing is exactly the opposite– to break software in creative and unexpected ways, and in doing so expose its flaws. In [1], Bishop argues for the educational value of penetration testing, which allows students to consider a system as it is actually used rather than it is intended to be used.

A fourth-year computer security course we offered in 2013 aimed to fill the gap in student's application testing and flaw analysis. A term project tasked students with locating two flaws in any software of their choice, creating a comprehensive report and analysis of the flaw and its root cause. This project encouraged students to question trust assumptions, and better understand the perspective of both the security expert and the malicious adversary.

By the end of the term, the students had successfully located and reported almost 80 flaws in a wide variety of software, many of which have subsequently been resolved by the software maintainers. The flaws were discovered with minimal guidance from the instructor, most students were able to find these flaws almost entirely through independent effort.

The reports created by the students were collected and scored based on their use of a variety of diagnostic methods, the quality of the report, and interactions with the software vendor or maintainer. The goal of the project was not to produce a corpus of security-significant vulnerabilities in software, and many of the bugs found had no security implications. However, a number of students were able to find interesting bugs, and those who did not still had an opportunity to hone the skills prerequisite to application testing.

As [2] discusses, whenever students are taught how to break systems, it is vital to maintain perspective on the ethics and legality of the work. To this end, our project had a significant emphasis on responsible disclosure, which is equally as important to educate students on as the technical skills associated with finding software flaws. All of the students were required to submit a formal report of the flaw, detailing its probable cause and discussing mitigation strategies. By making the report a marked component of the assignment, and laying out guidelines for creating quality reports, we also aimed to foster good bug reporting practices, another invaluable skill for students.

We encouraged students to disclose their findings to the software vendor or maintainer in question, and found that most students did this despite it not being a requirement (see Section 5). These interactions were promising and showed that students can be capable of self-motivated professionalism and accountability with such findings.

Our results from this project demonstrated the following:

- Students have the skills to find and analyse software flaws,
- Students are able to give back to the community with such activities, and
- The risk of teaching these skills to students is minimal if responsible disclosure is a focus.

In Section 3, we will describe the setup of the project, including the rubric which the students were given and our experimental design (namely, the rating scheme we have used for reports). We also discuss properties of good bug reports, and how this motivated our rating system. In Section 4 we discuss the student's preferred techniques for flaw analysis, discussing the trends seen in these preferences between different report classifications. We discuss our efforts to encourage responsible disclosure in Section 5, in which we found students exceeded expectations of professionalism.

2. RELATED WORK

Security training has recently become much more accepted and practiced at educational institutions, including practical exercises designed to teach students adversarial techniques. Numerous recent publications such as [3, 4, 5, 6] have described the use of practical security exercises to educate students on security analysis. We observe a number of common trends in these articles; firstly, that they are often based on designed scenarios, and secondly that they rely on a closed environment (either physical or virtual).

The use of pre-designed scenarios for information security learning is a common practice, and is used both in an academic setting, for security competitions (capture-the-flags), and for informal security education (crackme exercises, intentionally insecure local applications, et cetera). In [7], Poulious et. al describe scenario-based learning as a necessary accompaniment to lecture-based course delivery. The authors describe a sample workshop that follows their methodology, with a focus on specific steps that the students could follow to achieve their goal, and the tasks were clearly defined to the students.

Another educational framework that makes use of scenarios for security education is the EDURange [8] system. EDURange gives students access to a cloud-based sandbox for practical security exercises, focusing on network security. This framework is an open-ended one that focuses on goals that the students should accomplish more than their methods, which can be simultaneously more challenging and rewarding for students.

Scenario-based learning, as described in [7] and similar works, enables a more detailed description of the problem that students are to solve. This can be useful for less advanced students, and to maintain focus on key lessons. However, we find that this is not reflective of the actual analysis process, which is generally a task that has few initial specifications or goals and is less linear than pre-made exercises.

While open-ended software analysis is undoubtedly more challenging for students, our results show that students can succeed with less direct instruction and guidance if they are motivated to do so, and are relatively advanced in their education. By having students investigate real software in a less rigid environment, we encourage more creative analysis techniques. Using pre-defined security scenarios is useful if one wishes to focus on a particular lesson, and open-ended exercises such as the one we describe may be more challenging to focus the educational content in. Balancing both of these scenario-based and ad-hoc delivery methods is important for an educator.

A common trend for security exercises is to take place in a closed environment, either physical or virtual. Virtualization is a popular technique for containing security exercises, since it is easy to instantiate and modify the environment, and is cheaper than purchasing physical lab equipment. In [4], Andel et. al compare two common approaches for virtualized lab environments, namely operating system virtualization or full virtualization. The former is a memory-efficient tool to run virtual machines, but the latter allows for different system architectures to be instantiated, which is perhaps a more realistic simulated environment.

A virtual laboratory for security exercises is described in [3], focusing on making a link between the design of cryptographic protocols and their implementation. Another example is given in [9], where the VITAL framework is described. Both of these systems make use of virtualized laboratory environments to deliver security exercises in a controlled manner. The exercises described by Yuan et. al in [6], which includes web application testing and fuzz testing, were executed from within a decentralized virtual machine that each student had a copy of, another example of a virtualization based delivery platform.

While virtualized environments are convenient and cost-effective, they sacrifice some degree of realism for these benefits. A less common approach is to use a physical laboratory, as was done in [5]. This particular laboratory is described to be a networked laboratory that students used under careful supervision. This approach can be more challenging to control since students can turn their tools towards other devices, and cooperation with the IT department of the institution was required since some of the traffic the students generated would have been dropped by the firewall of the institution. The author describes a transition from using a fully enclosed laboratory environment to one that has restricted access to some other hosts on the internet, which was useful to expand the students' set of resources but required close cooperation with the IT department.

Our approach is unique in that it requires little lab infrastructure for its execution, and students can work from their own environment. This was both a cost-effective method and gave students a realistic environment to work in, an advantage over either virtualized or physical lab infrastructure.

Placing restrictions on what students can do during their security exercises is a important to prevent abuse of the system, and for supporting targeted educational exercises that reinforce particular concepts. Physical and virtual labs can enforce proper conduct by monitoring the student's behaviour. Our method, in contrast, enforces good conduct by making accountability a key component of the assignment, otherwise giving the students reasonable freedom to analyze real software in their own environments. We found that the students exceeded our expectations of accountability (see Section 5 for details), and while the open-ended environment was challenging for many students, it encouraged some interesting discoveries.

Other similar recent projects have put emphasis on student's abilities to work with foreign codebases and to perform software maintenance. [10] proposes the use of older student projects to give current students a foreign code body to practice software maintenance on. They found that using student-created code samples was useful for a realistic practice environment, since the code was not textbook perfect and was unfamiliar to the students. Our use of real software projects similarly gave students a realistic environment to practice their analysis skills in.

3. SETUP

Each student was tasked with collecting and reporting two bugs from any software of their choice. The bugs did not have to be from the same system, and there was a length of time in between the first and second report for the students to reflect on their first report.

The reports were to contain a variety of details:

- Summary of the software, flaw, and environment,
- Description of initial evidence of the flaw,
- Description of the diagnostic techniques used,
- A proof of concept input or steps to trigger the flaw,
- Discussion of the flaw's security implications, and
- Description of contact with the software vendor/maintainer.

With each section being a contribution to the total mark on the report. In total, we collected 79 reports from the 40 enrolled students.

Students were encouraged to use a wide variety of tools for software analysis, including debuggers, interactive disassemblers, and source-code analysis (where applicable). We detail in Section 4 the most common techniques that the students used in their searches.

A 2010 study [11] by Bettenburg et. al of three high-visibility open source projects- Eclipse, Apache, and Mozilla- aimed to empirically define what a good bug report should contain, through developer and reporter surveys. We have used this evidence provided to create a template for a good bug report, and assigned scores to the student's reports.

…the most widely used items across projects are steps to reproduce, observed and expected behavior, stack traces, and test cases.[11]

For the importance of items, steps to reproduce stands out clearly. Next in line are stack traces and test cases…[11]

We defined a report to be of high quality if it contains relevant information as outlined in [11] - in particular, we wished to see a *summary of the flaw, steps to* *reproduce, observed and expected behaviour, screenshots* where relevant, and *test cases*. Other information that was beneficial to the student's report was also a factor in the scoring.

A score of *low* was assigned to reports with incorrect or incomplete information, *medium* for reports that were sufficient but not as complete as possible, and *high* for reports that contained all necessary information. The scoring mechanism was not intended as a strong empirical quantification of bug report quality, but instead evidence that a given student report match the major structural features of the template report.

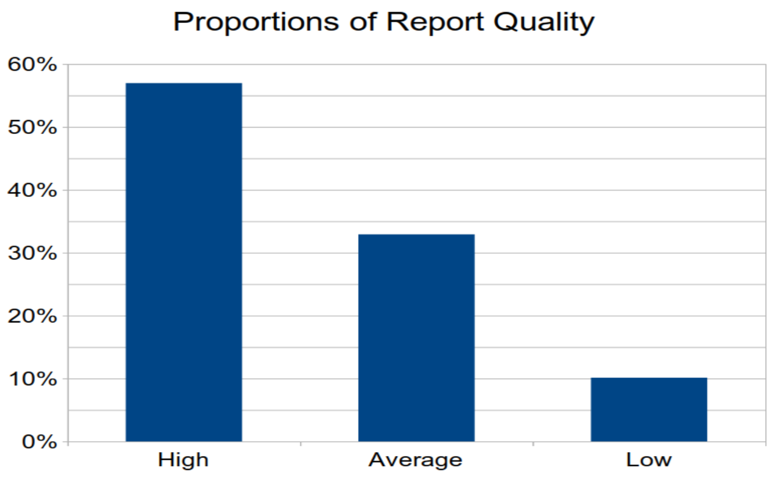

Figure 1: Proportion of student report quality scores

As seen in Figure 1 (pp. 7), 57% of the reports created were found to be of *high* quality, while only a small 10% of reports were given a *low* score. Because the rubric provided gave students a clear guideline on how to succeed in this assignment, this is not an unexpected outcome, and is evidence that students are more than capable of producing useful bug reports given some guidance.

4. STUDENT TECHNIQUES

Many tools and methods exist for the diagnosis of flaws in software. These range from the basic skills (such as visually identifying flaws) to the more advanced (using more powerful debugging tools). A successful analyst will use a wide variety of tools to gather evidence of the flaw and piece the information together to close in on the root cause.

We identified eight diagnostic vectors (see Fig. 2) that the students used in their reports. To quantify the student's use of these vectors, the reports were given unweighted scores in these eight vectors; a score of 0 indicated little or no use of the vector, 0.5 indicated partial or occasional use, and 1 indicated a high use. The intentionally coarse scoring mechanism attempts to minimize subjectivity, rather than provide an empirical quantification of a student's use of a given technique.

In this section we describe the common techniques used by students, distinguishing two broad investigative techniques- active and passive analysis. Passive analysis techniques include the following:

- Source Code analysis, in which students manually read source code to attempt to discover vulnerabilities or the root cause of a flaw,
- External Research, in which students locate vulnerabilities or vulnerability classes in software by way of examining past reports and attempting to reproduce the reports (note that the methods of reproduction may include other analysis techniques),
- Visual Evidence, in which students identify flaws in software based on its user interface, including accidental discovery of flaws through normal use, and
- Log Files, in which students use the log files of the software to identify misconfiguration of the software or to analyze a software crash.
- Active analysis techniques include the following:
- Handcrafted Active Input, in which students create an input or environment of execution for the software designed specifically to trigger a flaw that they have identified, generally through trial-and-error,

- Specialized Tools, which includes network analysis tools, application testing frameworks such as Metasploit [12], fuzz testing tools such as afl [13],
- Handcrafted Tools/Scripts, which are specialized tools or scripts that the students create themselves specifically for the purpose of analyzing the given software, and
- Debuggers and interactive disassemblers, where students use these applications to examine the state of the application after a crash or to determine its path of execution for some given input.

While active analysis techniques are more advanced, both passive and active analysis are useful tools for flaw identification. We found that most students focused on passive analysis techniques, perhaps due to lack of comfort with active techniques, but discovered that the students who used active techniques were more likely to discover interesting reports. Figure 2 shows the diagnostic methods used by students, and the raw values can be found in Table 3.

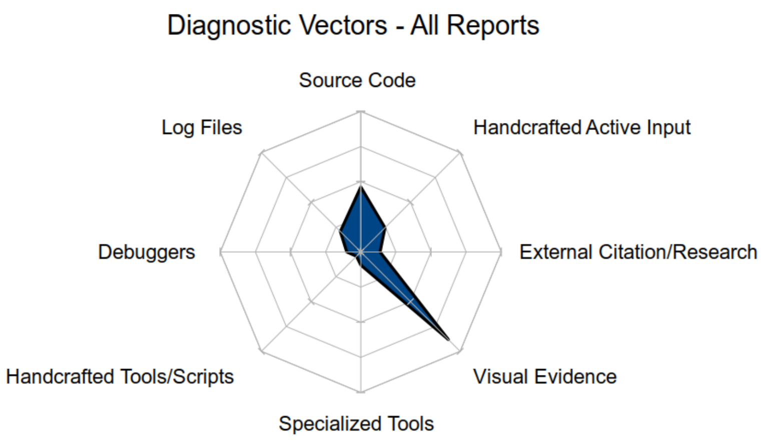

Figure 2: Bug diagnostic methods used by students (Average, all reports). The value of each vector is the average of all students' use of the given diagnostic tool, rated in terms of high, medium, low, and no reliance.

Avg. student vector reliance	All reports	High Quality Reports	Medium Quality Reports	Interesting Reports
Source Code	0.47	0.63	0.21	0.46
Log Files	0.21	0.23	0.23	0.12
Debuggers	0.10	0.14	0.02	0.15
Handcrafted Tools/Scripts	0.04	0.07	0.02	0
Specialized Tools	0.09	0.12	0.08	0
Visual Evidence	0.89	0.84	0.94	0.96
External Research	0.14	0.21	0.06	0.19
Handcrafted Input	0.25	0.30	0.23	0.38
Count	79	45	26	13

Table 3: Raw scores for student's utilization of bug diagnostic methods.

4.1 Passive Analysis

In the analysis of the diagnostic techniques used by the students, it was found that there was a high degree of clustering on passive techniques. In particular, visual evidence was overwhelmingly the most common choice, where the students identified software flaws by evidence in the user interface of the program.

These students demonstrated little use of common but vital tools such as debuggers, despite being close to graduating from their information security program. For an advanced group of students, this was somewhat concerning, and is evidence that students are not comfortable using these analysis tools.

Even when considering the high quality reports, there is still a marked disuse of some diagnostic techniques. A comparison of the diagnostic vectors used in medium and high quality reports in Figure 4 shows that higher quality reports tended to use their tools to a greater depth, but demonstrated similar patterns of preference. We also observed a marked increase in the reliance on source code between medium and high quality reports, which could indicate that open-source projects were easier to analyze, or that the medium quality reports simply did not make use of source code.

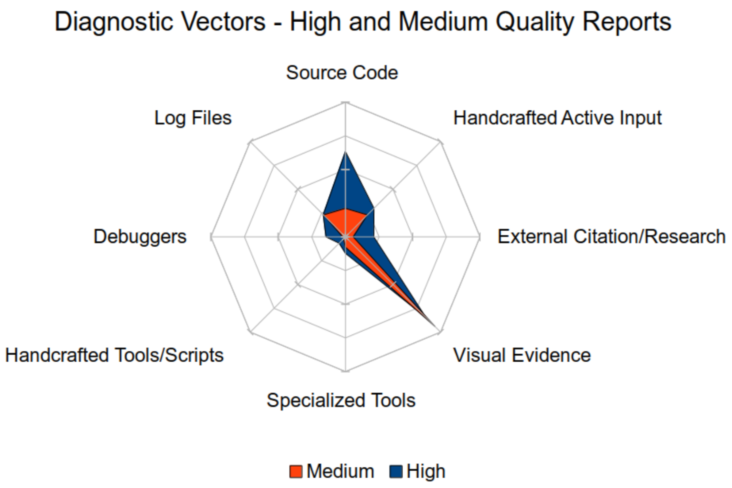

Figure 4: Bug diagnostic methods over High and Medium quality reports (Average, all reports in category). The value of each vector is the average of all students' use of the given diagnostic tool, rated in terms of high, medium, low, and no reliance.

Note that due to an insufficiently large sample size, an analysis of the low quality reports is omitted from Fig. 4.

The preference for certain techniques is not unexpected- a graphical bug will almost certainly be quickly noticed, whereas a memory leak may go unnoticed for some time without examining the resource usage or log files. However, even in their continued investigations, many students did not use the more advanced tools.

A small subset of students demonstrated that they were capable of effectively using a wider variety of techniques, but these students were the minority. This is evidence of a need for greater encouragement to use these advanced techniques, and more opportunity for activities such as this project in which students can hone their analytic abilities and the security mindset.

4.2 Active Analysis

More advanced students used techniques similar to those found in penetration testing exercises. This included the use of debuggers and interactive disassemblers, specialized analysis tools, and handcrafted input designed to trigger the vulnerability in the program. Use of such active techniques is what we believe to be a strong indicator of the security mindset. It is evidence that students are actively questioning trust assumptions, and creatively interacting with a system in an adversarial way. Use of these techniques were, in general, more fruitful for security related vulnerabilities (see Figure 5, pp. 12).

A student that is capable of this type of analysis demonstrates a much stronger understanding of the system in question. This skill requires a deep, cross-layer understanding of the target system rather than a superficial understanding of an API or protocol that typical use cases of software imparts.

In contrast, passive analysis of software can also reveal flaws, but more often than not, it suffices only for the more superficial bugs. While these methods can undoubtedly reveal important flaws in software, they are less useful to trigger unexpected behaviour in the system.

We found that students who were comfortable with these active methods tended to find the more interesting flaws in software. We categorized this subset of interesting bugs as those that posed a security risk to a non-trivial system, or were a highly unusual or unexpected flaw. Some examples of these interesting flaws included:

- A method to download paid Android applications directly from the Google Play Store without any net payment.
- Ability to bypass the Gnome Display Manager login screen in Scientific Linux 6 without authentication, providing access to some areas of the filesystem and some binaries (including the Python interpreter).

- Ability to create many smurf accounts in the Piazza web service from a single email address, and to share group privileges across all of these accounts.

Of the 79 reports, we separated out 13 such interesting bugs. We found a higher tendency for the use of advanced analytic techniques in the analysis of these flaws– in particular, providing handcrafted input to the system in an attempt to trigger unexpected behaviour (Figure 5). These particular reports show that some students will make good use of these opportunities in the classroom, and are able to find flaws that exceed what one may expect from an undergraduate term project.

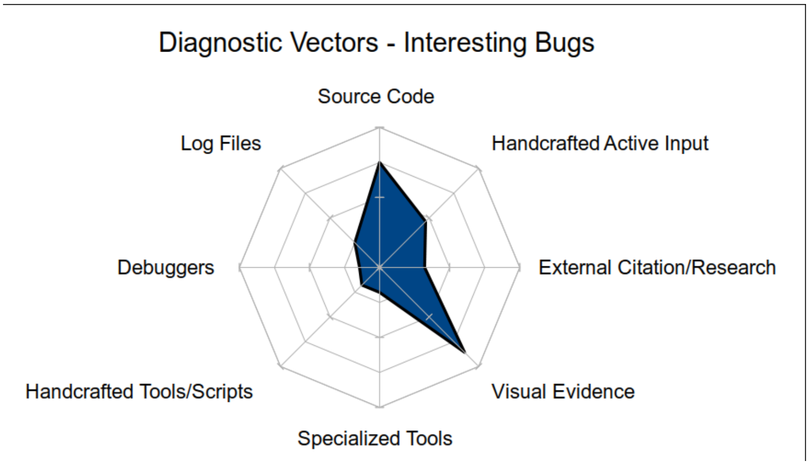

Figure 5: Bug diagnostic methods used in interesting reports (Average, all reports in category). The value of each vector is the average of all students' use of the given diagnostic tool, rated in terms of high, medium, low, and no reliance.

5. ACCOUNTABILITY

Many institutions are wary of offering this training to students, despite its possible educational value. There is a belief that because these offensive techniques can be used maliciously, that they will be used maliciously. In [14], Cook et. al describe their experiences with student misbehaviour, detailing a number of incidents where students used their adversarial knowledge inappropriately. While this is always a risk when teaching this type of material, we do not believe it reduces its importance to teach, but care must be taken to encourage good student conduct during and after the exercises.

A course that has been regularly offered over the last decade at the University of Calgary has students study existing malware and write their own in a controlled and secure lab environment. Registration in the course is a rigorous process and prospective students have to adequately prove to the instructor that they have the integrity and responsibility prerequisite to the material.

In his 2005 publication *Viruses 101* [15], Dr. John Aycock, Univ. Calgary, describes the importance of teaching these skills in a controlled environment such as an educational institution.

> *...it is very easy to learn how to create malware, even for people with no programming expertise. It is not easy, however, to learn this in a safe environment, nor to get an objective view of the entire field, both malware and anti-malware. [15]*

There is no doubt that there are many ways to learn these skills outside of an academic setting, but we believe that an educational institution is unique in its ability to structure this information, and couple it with an examination of ethics and legality.

The project placed a strong emphasis on responsible disclosure of flaws, and encouraged (but did not require) students to interact with the maintainers and community about the bugs they discovered. We found that despite the fact that community interaction was not required, it was still frequently performed by the students.

In one particular report, the student identified an improperly sanitized user input field in a small research organization's database. After identifying this vulnerability, the student was able to gain full access to the user database via SQL injection, including full email addresses and names. This particular flaw was reported to the organization as per the recommendation of the assignment, and was promptly patched. Not only did the student enhance their ability to pro-actively analyse and compromise a system, they did so in a way that benefited the vendor. This was a very common trait in most of the reports. A total of 86% of student reports (see Figure 6) indicated that the student had taken action to contact the software vendor or maintainer, despite the fact that the assignment only encouraged this and did not explicitly require it.

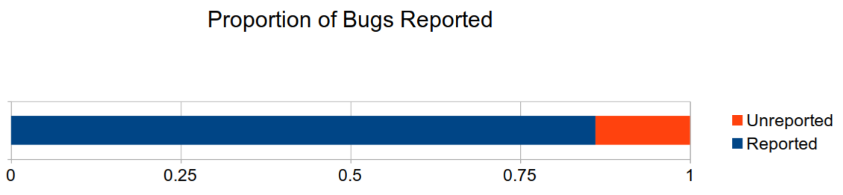

Figure 6: Proportion of bugs reported to software vendor or maintainer. Note that this was not a required component of the project, but was encouraged.

The students demonstrated a high degree of professionalism and accountability with their findings, a credit to the structured nature of this project. With enough guidance, it is clear that students can operate in this way, even when tasked with sensitive work.

6. CONCLUSION

The bug report project was designed not only to give students hands-on experience with flaw detection and security analysis, but also to impart the essential skills of communicating these findings, and the social responsibility to do so. The program was successful in this regard, and is evidence that this type of training is not a threat to the integrity of our students, as long as strict ethical standards are adhered to by the program.

The rubric described in Section 3 was reflective of these project goals and was useful in guiding student's behaviour while still maintaining a relatively open-ended specification. We specified a number of required report components that agree in part with the results of [11]. These components include steps to reproduce the flaw, the software environment used during its invocation, a proof of concept input to trigger the flaw (in essence, test cases), and a discussion of observed and expected behaviour.

We observed in Section 4 that students biased their efforts towards simpler tools and techniques. Students overwhelmingly preferred passive flaw analysis techniques such as visual evidence, source code analysis, and log file analysis. Comparatively few students made use of active analysis tools such as debuggers, interactive disassemblers, and specialized penetration testing tools, despite the importance of such techniques in actual security audit. Incorporating small practice labs into the class may be a useful strategy for ensuring students have a better idea of how to use these analysis tools, and we will explore this in future projects.

As could be expected, the more interesting reports (those with security implications or were creative or unexpected) were on average discovered through a more balanced set of tools and techniques compared to the average of all 79 reports. This may be indicative that the student's increased comfort with the toolset and techniques involved in security analysis gave them an advantage over their peers, or that the flaws in question required a more thorough analysis than some more superficial flaws. In either case, we believe that our results are evidence that there is room for improvement with the average student's auditing expertise and flaw analysis skills.

A key focus of our project was on accountability of the students, and on responsible disclosure. We found that in general students were willing to exceed expectations for responsible disclosure, with 86% of reports indicating that the student had notified the software vendor or maintainer of the flaw. This high proportion was in spite of the fact that communication with the maintainer was not a required component of the assignment. Despite the sensitive nature of the work, these students demonstrated professionalism with their findings. Maintaining a focus on responsible disclosure and ethics (and, of course, legality) is vital for any such project.

The open-ended nature of the project was challenging for students, particularly when many of them had little to no experience with analyzing foreign code bases (possibly without source code). Future work will seek to refine the structure of the assignment to ease students into software analysis. We also plan to further explore the effectiveness of this type of practical training, gauging student's comfort with software analysis practices before and after the project, and use these results to refine our procedure.

REFERENCES

[1] M. Bishop, "About penetration testing," *Security Privacy, IEEE*, vol. 5, pp. 84–87, Nov 2007.

[2] P. Y. Logan and A. Clarkson, "Teaching students to hack: curriculum issues in information security," in *ACM SIGCSE Bulletin*, vol. 37, pp. 157–161, ACM, 2005.

[3] T. Zlateva, L. Burstein, A. Temkin, A. MacNeil, and L. Chitkushev, "Virtual laboratories for learning real world security," in *Proceedings of the 12th Colloquium for Information Systems Security Education, Dallas, TX, 2008.*

[4] T. R. Andel, K. E. Stewart, and J. W. Humphries, "Using virtualization for cyber security education and experimentation," in *Proceedings of the 14th The Colloquium for Information Systems Security Education (CISSE 2010)*, 2010.

[5] B. Eckart, "Real-world security lab environment," in *Proceedings of the 16th The Colloquium for Information Systems Security Education (CISSE 2012)*, 2012.

[6] X. Yuan, J. Hernandez, and I. Waddell, "Hands-on laboratory exercises for teaching software security," in *Proceedings of the 16th The Colloquium for Information Systems Security Education (CISSE 2012)*, 2012.

[7] N. S. Poulious and D. Pradhan, "Scenario based exercises in IA courses," in *Proceedings of the 15th The Colloquium for Information Systems Security Education (CISSE 2011)*, 2011.

[8] S. Boesen, R. Weiss, J. Sullivan, M. E. Locasto, J. Mache, and E. Nilsen, "Edurange: meeting the pedagogical challenges of student participation in cybertraining environments," in *Proceedings of the 7th USENIX conference on Cyber Security Experimentation and Test*, pp. 9–9, USENIX Association, 2014.

[9] E. Gavas and K. O'Brien, "Teaching network security using vital," in *Proceedings of the 16th The Colloquium for Information Systems Security Education (CISSE 2012)*, 2012.

[10] C. Szabo, "Student projects are not throwaways: teaching practical software maintenance in a software engineering course," in *Proceedings of the 45th ACM technical symposium on Computer science education*, pp. 55–60, ACM, 2014

[11] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 308–318, ACM, 2008.

[12] Metasploit, "The metasploit project," 2006.

[13] lcamtuf@coredump.ctx, "American fuzzy lop."

[14] T. Cook, G. Conti, and D. Raymond, "When good ninjas turn bad: Preventing your students from becoming the threat," in *Proceedings of the 16th The Colloquium for Information Systems Security Education (CISSE 2012)*, 2012.

[15] J. Aycock and K. Barker, "Viruses 101," *SIGCSE Bull.*, vol. 37, pp. 152–156, Feb. 2005.