Understanding the how and the why: Exploring secure development practices through a course competition

Kelsey R. Fulton
University of Maryland

Daniel Votipka
Tufts University

Desiree Abrokwa
University of Maryland

Michelle L. Mazurek
University of Maryland

Michael Hicks
University of Maryland

James Parker
Galois, Inc.

Abstract

Secure software development is a difficult task owing to the various pressures faced by developers (e.g. system performance and correctness). This paper investigates why developers introduce different vulnerabilities, the ways they evaluate programs for vulnerabilities, and why different vulnerabilities are (not) found and (not) fixed by developers. To understand the various processes that developers might employ and the types of vulnerabilities, they may introduce, find, and fix, we conducted an in-depth study of 14 teams' development processes during a three week undergraduate course organized around a secure coding competition. Participants were expected to build code to a specification while emphasizing correctness, performance, and security. Additionally, participants searched for vulnerabilities in other teams' code while being responsible for fixing any exploited vulnerabilities in their own code. We used iterative open coding to manually analyze data including code, commit messages, team design documents, and various surveys. We find associations between design and development processes and resulting code security, as well as trends in the exploitation, discovery, and patching of different vulnerabilities. For example, teams which codified more detailed designs before writing code tended to have fewer vulnerabilities in their code, but also were unlikely to revisit their design despite the discovery of vulnerabilities. Our results point to possible changes to improve secure programming processes, secure programming tools, and development team organization.

1 Introduction

Secure software development is a difficult task, exemplified by the fact that vulnerabilities are still discovered in production code on a regular basis [8, 21, 28]. Many solutions have been put forward to solve this problem: more security education [6, 12–14, 27], better secure development tools [2, 4, 5, 10, 11, 16, 22, 34–37], and better integration of security in to the software development cycle [3, 7, 15, 20, 32]. Given the difficulty of balancing various business pressures (e.g., costs, customer experience, product delivery) during the development lifecycle [31], it is important to understand which solutions aid secure development most effectively and efficiently. Companies simply will not adopt every secure development practice; how should they prioritize the various choices? To answer this question, we must understand *why* developers introduce different vulnerabilities, as well as how and why testers (do not) find and fix them, in order to identify processes and tools that most effectively reduce real risks.

Prior work has considered secure development in controlled settings, allowing clear comparisons among different tools and strategies [1, 24–26, 29]. While valuable, these studies are limited in ecological validity, as the program size and flexibility of approach are restricted by necessity. Conversely, other work has reviewed open-source repository commits to identify practices correlated with greater vulnerability incidence, providing results from a real-world setting [17–19, 30]. However, it is difficult to make clear comparisons between these codebases due to significant differences in the goals and functional requirements of each project. This research also typically cannot investigate developer motivations or thought processes, as only submitted code (with often-terse commit messages) is available. Finally, some recent work has taken an ethnographic approach, embedding researchers in companies to observe secure-development practices [31, 38]. This work provides rich insights into the development process, but to date, has mostly focused on organizational processes impeding security, not technical issues.

Ruef et al. sought to establish a middle point along this spectrum with the *Build It, Break It, Fix it* (BIBFI) secure-coding competition, which balances ecological validity with study control by having participants complete a multi-week, well-defined programming project with few process constraints [**?**]. Votipka et al. then reviewed code submitted during four BIBIFI competitions to uncover an in-depth taxonomy of the vulnerabilities introduced by developers while building secure software [39]. They manually analyzed submissions to discover characteristics of vulnerabilities developers intro-

duced, such as general vulnerability type, severity, and ease of exploitation. However, as they only reviewed submitted code, they were not able to determine ***why and how*** developers introduce, find, and address vulnerabilities. Understanding this would enable better recommendations to improve secure software development, security education, and secure development tools.

To address this limitation, we conducted an in-depth study of 14 teams' development processes during a three week undergraduate course centered on a BIBIFI secure-coding competition. Student teams built a software-based home IoT system with role-based access control policies. Teams then attempted to find vulnerabilities in other teams' code and fixed vulnerabilities in their own code found by other teams. The course scoring emphasized real-world constraints and priorities, i.e., security, performance, and functionality.

Implementing the BIBIFI competition as a short course allowed us to collect fine-grained data about participants' mindsets and approaches, both while developing software and when finding vulnerabilities. Doing so allowed us to understand why the participants introduced vulnerabilities, as well as how and why they found them and (sometimes) fixed them. Prior exploration of BIBIFI submissions [39] revealed, in depth, the type and details of introduced vulnerabilities. Our work confirms their results and adds insight into *why* developers introduce these vulnerabilities. We consider three key research questions:

RQ1 What types of vulnerabilities do developers introduce? Why?
RQ2 What types of vulnerabilities are found in code review? Why?
RQ3 Why do developers fix different types of vulnerabilities? How?

We identify key trends answering each question, and our results suggest the importance of including security in detailed designs, and revisiting and updating those designs while following secure development best practices.

2 Data and Analysis

The course followed a modified BIBIFI competition structure [33], organized into three phases: *build*, *break*, and *fix*. Course participants worked in teams for one week to *build* a lightweight IoT smart home controller that manages a smart home by receiving updates from sensors and controlling output devices. After the build phase, the course used a hybrid break-fix phase. All teams' code was made available to the other teams, which could then attempt to *break* their classmates' submissions by producing test cases demonstrating vulnerabilities. As breaks were identified, teams could update their code to *fix* vulnerabilities; teams lost points for every 24 hours that a known break against their code went unfixed.

In total, we analyzed the data of 14 teams through *iterative initial coding* [9, 23]. The codebook we developed provides labels for the different elements in Table 1. Inter-rater reliability (IRR) was generally not calculated, as the small number of responses and submissions for many aspects of the data did not allow for it. This study was approved by our institution's ethics review board.

3 Results

Our analysis of code submissions identified 147 unique vulnerabilities introduced throughout the build round and 80 unique vulnerabilities remaining in participants' code at the conclusion of the build round. Teams submitted 52 unique and 104 total breaks and left 19 vulnerabilities unfound and not exploited. Finally, teams fixed 66 vulnerabilities in their code during the build round and 31 vulnerabilities in their code during the fix round. 38 vulnerabilities were left unfixed (19 exploited) at the study's conclusion (47%).

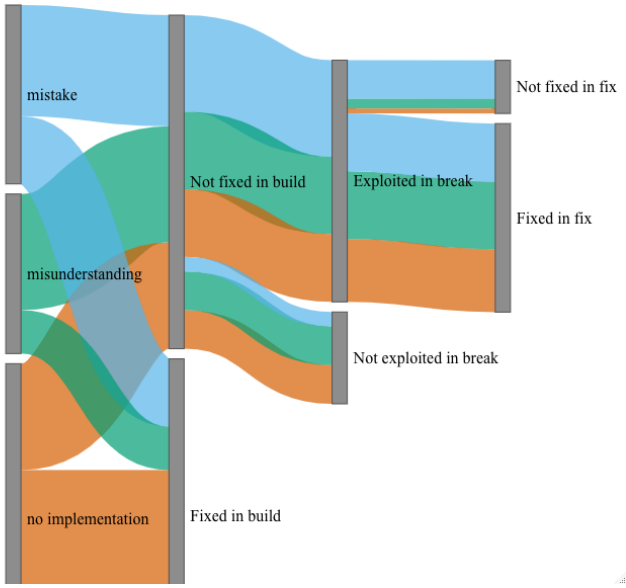

Figure 1: Number of vulnerabilities through each phase of different types.

3.1 Classes of Vulnerabilities

No Implementation A vulnerability was labeled as *No Implementation* when participants failed to attempt to implement necessary security mechanisms (i.e., access control, authentication, or timeout mechanisms) at all. We further divide this type into three sub-types depending on whether the requirements were mentioned directly in the specification. Specifically, the *All Intuitive* and *Some Intuitive* codes were used when teams failed to implement all or some, respectively, of the stated security requirements (e.g., missing all or some access control commands). *Unintuitive* requirements were not as explicit within the specification (e.g. recursive delegation).

Round	Data	Subcomponent	Description	Frequency	N
Build	Build submission	Code	–	When added single feature or fixed single bug	676
		Commit message	Description of change Reason for change Associated requirement How did the team come up with this change? Did teams change their design?	Each build commit	676
	Design documents	–	Team design of system Detail of potential threats Detail of potential mitigations	Before build, after build	27
Break	Break submission	Attack submission	Test to be run on target Description of exploit	Each attack	275
		Commit message	Description of issue in target How was the issue found? Requirement broken by issue Fixed version of failed break Difficulty to find break Difficulty to exploit break	Each attack	275
Fix	Fix submission	Code	Addresses vulnerability in their code	Each fix submission	48
		Commit message	Issue with implementation Cause of the vulnerability How did the team fix the vulnerability Confidence in security of their code Difficulty to identify the vulnerability Update to design required?	Each fix submission	48

Table 1: Description of data collected throughout the study.

Misunderstanding A vulnerability was labeled as a *Misunderstanding* when teams attempted to implement necessary security requirements but misunderstood the requirements or some security concepts during implementation.

Mistake A vulnerability was labeled as a *Mistake* when participants attempted to correctly implement security checks and functionality, but made a programming mistake that resulted in a vulnerability.

3.2 Development approach's security impact

Detailed design is common with fewer vulnerabilities When comparing a team's design depth to the security of their submission, we observed teams with detailed designs tended to introduce fewer *No Implementation* and *Mistake* vulnerabilities. Teams that failed to mention details of how they would mitigate certain attacks or manage certain aspects of security tended to fail to account for those security features in their codebase.

Teams with detailed designs did not revisit their design even if it had vulnerability, especially for *Misunderstandings* We observed that teams with a detailed design tended to stick with those designs, which sometimes encoded initial *Misunderstandings* into their projects. Teams that had a fundamental *Misunderstanding* of security requirements designed in detail for these features within their design documents. This suggests that teams that misunderstand the system's security needs from the beginning (i.e., design-time) are unlikely to catch these issues later.

Building security early, from the start correlates with secure development When we consider teams' development timelines in comparison to the number of vulnerabilities introduced during the build phase, we note several trends. Teams with the fewest vulnerabilities tended to do no security work on the last days of the build phase and started to build their security code early and edit it slowly throughout the build phase. Further, we note that teams that waited to implement access control until late in the build phase often ran out of time to implement less intuitive requirements, despite building these requirements into their designs.

3.3 Analysis of (Un)exploited Vulnerabilities

Missing access control found when checking related issues In general, teams exploited missing access control checks

while targeting a tangential, but related, access control requirement, rather than via specific targeted attacks where they reviewed code for a specific vulnerability.

Teams target more glaring issues rather than complex vulnerabilities Of the *No Implementation* vulnerabilities that were left unexploited, none of them were *All Intuitive* security features. The *No Implementation* vulnerabilities that were left unexploited were in code that had other, more glaring issues; teams generally favored attacking these issues rather than the slightly more complex vulnerabilities. Teams favoring glaring issues rather than attacking more complex issues may be an artifact of the study, as teams knew that the developers of the code were students and the code was likely to contain bugs. However, we expect that code review and testing in the software development lifecycle also commonly target more obvious problems, and we incentivized specific targeting by giving more points for the discovery of novel bugs.

***Misunderstanding* vulnerabilities were exploited with targeted testing** Breaks exploiting these *Misunderstandings* were often crafted to test for a specific *Misunderstanding* rather than testing for a broader, related requirement or being found incidentally. The *Misunderstanding* vulnerabilities that were left unexploited required deeper knowledge to exploit, and these vulnerabilities were likely left unexploited since no single break submitted targeted these vulnerabilities in any project.

***Mistake* vulnerabilities were exploited incidentally** Teams exploited nearly all *Mistake* vulnerabilities incidentally, while targeting an unrelated vulnerability. That *Mistake* vulnerabilities were widely caught incidentally using high-level, broad testing points to the ability for fuzzers to uncover these vulnerabilities during the development phase. Teams only needed to test for basic functionality, akin to the testing performed by fuzzers, to uncover these *Mistakes*. Several teams did not comprehensively test in the build phase, likely due to time constraints, but were able to build a set of comprehensive tests during the break phase, uncovering many vulnerabilities in other teams' code. This suggests that with sufficient time and effort, developers could test for and uncover most *Mistake* vulnerabilities even with minimal security training. This suggests that in principle developers could use (and generate seeds for) tools like fuzzers, if the tools were sufficiently available and usable.

3.4 Analysis of Fixed Vulnerabilities

***No Implementation* fixes require restructuring the program** While many *No Implementation* vulnerabilities were fixed during the build phase, nearly half of them were left unfixed at the conclusion of the study. The exploited *No Implementation* vulnerabilities required teams to redo their entire security codebase to address the vulnerability. of designing in depth for access control requirements from the outset, as designing in detail from the start prevents heavy redesign to address issues later.

***Misunderstanding* vulnerabilities are typically only fixed when pointed out** Vulnerabilities caused by a *Misunderstanding* of the security requirements were often not found and fixed until pointed out by either instructor-provided tests (during the build phase) or submitted exploits against a team's codebase (during the break phase). However, instructor-provided tests only covered testing for more basic functionality and failed to test for more complex access control requirements. As a result, more complex *Misunderstandings* of access control were often not found until they were exploited in the break phase. Receiving detailed input about security misunderstandings in their code allowed teams to address this issue and understand where they went wrong. Overwhelmingly, *Misunderstanding* vulnerabilities were fixed once they were pointed out and explained to teams, pointing to the benefit of including explanation of security *Misunderstandings* in the development process. Teams demonstrated the ability to learn from these explanations by crafting tests for other teams based on what had been exploited in their own code.

4 Conclusion

Secure software development is a challenging task. To prioritize among security solutions and provide the most help to developers, we must understand how and why developers introduce vulnerabilities, as well as how and why they are (not) found and fixed during software testing. To this end, we conducted an in-depth study of 14 teams' development processes during a three-week undergraduate course as they built a software-based home-IoT controller, attacked other teams' code, and fixed exploited vulnerabilities within their own code. We collected a wide variety of data throughout different portions of the course, allowing us insight into participants' thought processes and decision making. Overall, our results reaffirm the importance of secure development best practices.

References

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *IEEE Symposium on Security and Privacy*, pages 154–171. IEEE, 2017.

[2] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *IEEE Pacific Rim International Symposium on Dependable Computing*, pages 301–306, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Symposium on Usable Privacy and Security*, pages 281–296, Baltimore, MD, 2018. USENIX Association.

[4] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *International Symposium on Empirical Software Engineering and Measurement*, pages 97–106, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Dejan Baca, Bengt Carlsson, Kai Petersen, and Lars Lundberg. Improving software security with static automated code analysis in an industry setting. *Software: Practice and Experience*, 43(3):259–279, 2013.

[6] Center for Cyber Safety and Education. Global information security workforce study. Technical report, Center for Cyber Safety and Education, Clearwater, FL, 2017.

[7] Pravir Chandra. Software assurance maturity model. Technical report, Open Web Application Security Project, 04 2017.

[8] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. Trend analysis of the cve for software vulnerability management. In *Proceedings of the 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 1290–1293. IEEE, 2011.

[9] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.

[10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Google. Go Programming Language. https://golang.org/, 2020.

[12] Mariana Hentea, Harpal S Dhillon, and Manpreet Dhillon. Towards changes in information security education. *Journal of Information Technology Education: Research*, 5:221–233, 2006.

[13] Melanie Jones. Why cybersecurity education matters. https://www.itproportal.com/features/why-cybersecurity-education-matters/, 2019.

[14] Timothy C Lethbridge, Jorge Diaz-Herrera, Richard Jr J LeBlanc, and J Barrie Thompson. Improving software practice through education: Challenges and future trends. In *Future of Software Engineering*, pages 12–28. IEEE Computer Society, 2007.

[15] Gary McGraw, Sammy Migues, and Brian Chess. Software security framework | bsimm, 2009. (Accessed 05-22-2018).

[16] Gary McGraw and John Steven. Software [in]security: Comparing apples, oranges, and aardvarks (or, all static analysis tools are not created equal. http://www.informit.com/articles/article.aspx?p=1680863, 2011. (Accessed 02-26-2017).

[17] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[18] Andrew Meneely, Alberto C Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44, 2014.

[19] Andrew Meneely and Oluyinka Williams. Interactive churn metrics: socio-technical variants of code churn. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–6, 2012.

[20] Microsoft. Microsoft security development lifecycle practices. https://www.microsoft.com/en-us/securityengineering/sdl/practices, 2019.

[21] Mitre. CVE. https://cve.mitre.org/, 2020.

[22] Mozilla. Rust Programming Language. https://www.rust-lang.org/, 2020.

[23] Johnny Salda na. *The coding manual for qualitative researchers*. Sage, 2 edition, 2014.

[24] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. "if you want, i can store the encrypted password": A password-storage field study with freelance developers. In Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos, editors, *Proceedings of the 2019 Conference on Human Factors in Computing Systems*, CHI, page 140. ACM, 2019.

[25] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong?: A qualitative usability study. In *ACM Conference on Computer and Communications Security*, pages 311–328. ACM, 2017.

[26] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Symposium on Usable Privacy and Security*, pages 297–313, Baltimore, MD, 2018. USENIX Association.

[27] William Newhouse, Stephanie Keith, Benjamin Scribner, and Greg Witte. Nist special publication 800-181, the nice cybersecurity workforce framework. Technical report, National Institute of Standards and Technology, 08 2017.

[28] NIST. National Vulnerability Database. https://nvd.nist.gov/general, 2020.

[29] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. API blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 315–328, Baltimore, MD, August 2018. USENIX Association.

[30] Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 6, pages 10–5555, 2006.

[31] Hernan Palombo, Armin Ziaie Tabari, Daniel Lende, Jay Ligatti, and Xinming Ou. An ethnographic understanding of software ({In) Security} and a {Co-Creation} model to improve secure software development. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 205–220, 2020.

[32] Tony Rice, Josh Brown-White, Tania Skinner, Nick Ozmore, Nazira Carlage, Wendy Poland, Eric Heitzman, and Danny Dhillon. Fundamental practices for secure software development. Technical report, Software Assurance Forum for Excellence in Code, 04 2018.

[33] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *ACM Conference on Computer and Communications Security*, pages 690–703, New York, NY, USA, 2016. ACM.

[34] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *ACM Conference on Computer and Communications Security*. ACM, 2017.

[36] Larry Suto. Analyzing the effectiveness and coverage of web application security scanners. Technical report, BeyondTrust, Inc, 2007.

[37] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. Technical report, BeyondTrust, Inc, 2010.

[38] Anwesh Tuladhar, Daniel Lende, Jay Ligatti, and Xinming Ou. An analysis of the role of situated learning in starting a security culture in a software company. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 617–632, 2021.

[39] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 109–126, 2020.