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

Kelsey R. Fulton, Daniel Votipka*, Desiree Abrokwa, Michelle L. Mazurek, Michael Hicks, and James Parker†

University of Maryland, *Tufts University, †Galois, Inc.

ABSTRACT

This paper presents the results of in-depth study of 14 teams' development processes during a three-week undergraduate course organized around a secure coding competition. Contest participants were expected to first build code to a specification—emphasizing correctness, performance, and security—and then to find vulnerabilities in other teams' code while fixing discovered vulnerabilities in their own code. Our study aimed to understand why developers introduce different vulnerabilities, the ways they evaluate programs for vulnerabilities, and why different vulnerabilities are (not) found and (not) fixed. We used iterative open coding to systematically analyze contest data including code, commit messages, and team design documents. Our results point to the importance of existing best practices for secure development, the use of security tools, and development team organization.

CCS CONCEPTS

• **Security and privacy → Usability in security and privacy**;

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 [18, 50, 61]. Many solutions have been put forward to solve this problem: more security education [16, 34, 35, 40, 60], better secure development tools [5, 8, 9, 22, 28, 44, 51, 68, 73, 78, 79], and better integration of security in to the software development cycle [6, 17, 43, 49, 66].

Given the difficulty of balancing various business pressures (e.g., costs, customer experience, delivery date) during software development [64], 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 their 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, 2, 54–56, 62]. While valuable, these studies are limited in ecological validity, as the program size and flexibility of approach are restricted by necessity. Other work has reviewed open-source repository commits to identify practices correlated with greater vulnerability incidence, providing results from a real-world setting [45–47, 63]. However, it is difficult to make clear comparisons between these codebases due to significant differences in goals and functional requirements. 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 [64, 82]. 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. [67] sought to establish a middle point along this spectrum with the *Build It, Break It, Fix it* (BIBIFI) 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. [83] then reviewed code submitted during four BIBIFI competitions to uncover an in-depth taxonomy of the vulnerabilities introduced by developers while building secure software. They manually analyzed submissions to discover characteristics of vulnerabilities developers introduced, 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 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 [83] 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.

For RQ1, we find that both the extent of the formal design, and the overall development timeline, seem to relate to security outcomes. Teams with a detailed initial design tended to introduce fewer vulnerabilities than those without. However, teams with detailed initial designs tended to stick with them, so errors or misunderstandings in those designs became vulnerabilities in the finished code. In terms of timeline, teams with the fewest vulnerabilities did little or no security work on the very last days of the build phase, instead working steadily on security throughout. Conversely, teams that waited until the end to implement security often ran out of time and failed to implement key security functions, leading to many vulnerabilities.

For RQ2, we find that different types of vulnerabilities are discovered differently. Teams often found vulnerabilities almost incidentally: vulnerabilities arising from the failure to implement some security requirement(s) were most often found when looking broadly for a related problem, rather than specifically for that vulnerability type. Vulnerabilities that arose from simple programming mistakes similarly tended to be found incidentally, although they were frequently found by testing for almost anything, related or not. In contrast, vulnerabilities related to misunderstanding security properties required deeper knowledge and more focused testing. Overall, student teams strongly preferred to find simple vulnerabilities; more complex misunderstandings (and associated vulnerabilities) were rarely targeted, if at all.

For RQ3, we find that different types of vulnerabilities are fixed differently. Vulnerabilities relating to failing to implement some security requirement(s) were likely to remain unfixed at the end of the study, possibly because fixing them would in many cases require rearchitecting large parts of the system design. Vulnerabilities associated with simple mistakes were often found by teams themselves, during their own build-phase testing (before attacks from other teams), and were typically easy to fix, but sometimes went unfixed due to time constraints. In contrast, vulnerabilities caused by misunderstanding security requirements were rarely found until they were exploited by other teams. Teams could only fix these vulnerabilities once they had corrected their misunderstandings, which could be challenging and time-consuming.

We observe that current automated tools tend to target vulnerability types, such as mistakes, that are easily found and exploited, but are unlikely to flag vulnerabilities that are difficult to both find and fix, such as those owing to a misunderstanding of security concepts. These vulnerabilities might be better addressed by consulting a security expert during the design process. Indeed, 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

This section describes the course structure and the data collected.

2.1 Course structure and data analysis

Course structure. The course followed a modified BIBIFI competition structure [67], organized into three phases: *build*, *break*, and *fix*. Course participants worked in teams for one week to *build* a substantive program (2363 LoC on average), following a provided specification (See below for specification description). Teams earned points based on the number of correct functions they implemented (evaluated via instructor-defined test cases) and their code's performance (running time of instructor test cases).

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. Teams were also allowed to submit vulnerability reports that clearly stated the insecure line(s) of code in the target program and explained the vulnerability, as well as why it was not feasible (within class constraints) to demonstrate it. This could occur due to competition time constraints (e.g., brute force exploitation not feasible in the course timeframe without specialized equipment) or exploit complexity. We added this option because in previous contests it was not clear whether teams had failed to find vulnerabilities, or had found them but been unable to exploit them [67, 83]. None of these reports were submitted.

Teams gained points for each valid break submitted. To incentivize unique breaks, different teams exploiting/reporting the same vulnerability shared its points evenly. As breaks were identified, teams could update their code to *fix* vulnerabilities; teams lost points for every 24 hours that a known break went unfixed. This hybrid break-fix setup is a departure from prior BIBIFI exercises, where the break and fix phases occur sequentially [67]. We made this change to better reflect real-world scenarios in which developers often have to handle fixing issues in code while managing other responsibilities and encourage teams to perform fixes, which was a problem in previous competition iterations [67, 83].

Project Specification. The build-it project was a lightweight IoT smart home controller that manages a smart home by receiving updates from sensors and controlling output devices. The specified controller can run user scripts, written in a domain-specific language, that update output devices (e.g. lights, thermometer, etc), retrieve information about the current state of the system (e.g. lights on/off, thermometer temperature, etc), or store information for future use (e.g. variables, permissions for users, new users, etc). Users of this smart home submit scripts in the domain-specific language. Sensors, output devices, and data are protected by role-based access control policies customizable by special users (admin) and the data's owner. Other users may receive delegated access, directly or transitively. This role-based access control was expected to be recursive, such that if a parent (delegator) lost permission on a variable, its children (delegatees) also lost permission. Further, the controller must validate user-provided scripts so that they will only run if properly authorized. Teams were expected to implement authentication using passwords and usernames. For extra build points, teams could implement some commands that operate similarly to required commands, such as for loops, indexing variable history, or resetting variable history. The full specification can be found at the following link: https://osf.io/s8ztb/?view_only=f8b9fb7804ab46648000ff1f52ca0d6c.

Class Instruction. Since participants were not required to have prior security training, the course started with a short introduction to security and threat modeling taught by one of the researchers. This training ensured a minimum level of security knowledge among participants and covered examples and approaches for thinking about possible attack surfaces and mitigations when developing a program. Teams were given time in class (about 11 hours total, over the 2.5-week course) to build their system, break other systems, and fix breaks. This ensured dedicated time for teams to work collaboratively. Teams were also expected to work on the competition outside class time, in lieu of other homework. During class time, teams were able to ask the instructors—the researcher who provided the original lecture and another research team member—to clarify vagueness in the specification and resolve issues teams had working with the course infrastructure. Any specification clarifications were announced to the entire class and updated in the specification document to ensure consistency between teams. When teams asked for help or advice with system design or implementation, the instructors directed students through problem introspection asking teams first to describe what is correct about their approach and how they know, then to identify differences from correct outputs. This series of introspective questions is taken from prior work investigating successful CS tutoring approaches [29].

Course-centered research design. For this classroom-based research, we drew on methodologies common to CS and engineering education research [13, 24]. In this complex setting, education research suggests collecting three types of data—*activity* (e.g., observations [10]), self-reported *accounts* (e.g., interviews [12, 14] and surveys [23, 71]), and *artifacts* (e.g., program designs [75, 81] and code [76])— and triangulating results among these data points [24, pg 181-186]. We adopt this approach, collecting *accounts* and *artifacts* in each BIBIFI phase, as detailed in the following sections (Sections 2.2-2.4). We considered collecting *activity* data, but chose not to as it did not make sense in the setting of our study. Much of our teams' work was completed outside of class, so direct observations were limited and self-reporting of this kind would likely be vague or unreliable [21]; create significant additional work for participants, potentially causing response fatigue [65]; and would offer limited additional information from the other data sources. Because we collect multiple types of *account* and *artifact* data, we believe we have sufficient information to provide useful triangulation.

Data analysis. We analyzed most of this data through *iterative initial coding*, sometimes also known as *open coding* [19, 52]. This analysis produces a set of labels called a *codebook*. Our codebook 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. When we did calculate IRR, we used Krippendorff's α, a conservative measure that considers how much better coders are than simply guessing [32]. A threshold of $\alpha > 0.8$ is recommended as a sufficient level of agreement [32]. We specifically highlight cases where we calculate IRR in the following sections. For most of the data analysis, the first author and one other team member coded the data. We specifically highlight the one case where this is different. The following subsections describe the coding processes and codebooks for each different major data category.

2.2 Build data (RQ1)

During the build phase, we collected and initial-coded data from three sources: build submissions (*artifact*), commit messages (*account*), and design documents (*artifact*).

Build submissions. While writing project code, participants were instructed to produce regular, atomic commits (i.e., a single change per commit) to a designated gitlab repository accessible by the teaching and research staff. To incentivize regular submissions, each commit automatically triggered performance and correctness testing to update the team's performance and functionality scores, giving participants immediate feedback on their efforts. With regular, atomic commits, we were able to track participants' progression, implementation strategies, and areas of focus.

Students were also instructed to discuss the context of the commit (e.g., what changed, reason for change, associated requirement, impact on design) in each commit message. The full set of items required in build commit messages are given in Table 1. These messages provide an important window into participants' development process: what they were working on, the reasoning behind their approach, and how it fits into their design. By collecting this data throughout the competition, we also learn about the development timeline and what events trigger changes. Perhaps most importantly, detailed commit messages frequently allow us to differentiate vulnerabilities arising from misunderstanding of security concepts from implementation errors, helping us to understand why different vulnerabilities are being introduced (**RQ1**).

Design documents. To provide additional insight into teams' understanding of security requirements, we asked them to produce design documents. Teams were required to submit documents detailing their system's design, and enumerating potential threats and associated mitigations, as described in Table 1. Using this data, we can learn more about the causes of vulnerabilities, and distinguish errors in design from implementation errors (**RQ1**). Students submitted initial design documents individually (16 students) before development began (teams had not yet been formed), and then teams submitted final design documents at the end of the build phase (11 teams), providing insight into how their designs change as they work together to develop, as well as how and whether security issues are introduced and/or eliminated over time.

Analysis of Build data. We analyzed the commit messages and the code submissions using iterative initial coding, as described in Section 2.1. For the build submissions, the first author and an additional author—different from the second coder discussed in subsequent sections—performed the analysis. This second coder was selected specifically because expert security knowledge was required to identify vulnerabilities in the build submissions. The two coders looked for vulnerabilities independently in each team's code; later, break submissions were also reviewed to ensure no vulnerabilities were missed and all were appropriately categorized. A vulnerability was only labeled within the codebase once a team attempted to implement that functionality or security. Teams did not start with *No Implementation* vulnerabilities because their code did not have any access control (because they hadn't built anything yet), but, rather, as teams built individual pieces of functionality, without any access control, vulnerabilities were labeled accordingly.

IRR was calculated for all build submission variables other than binary variables (the presence of a vulnerability or change in design document), since coding binary variables requires little to no interpretation, as recommended in prior work [42]. All vulnerabilities were confirmed by both coders, and consensus for all codes was reached via discussion. We categorize different vulnerability types using Votipka et al.'s codebook [83]. The final codebook and associated IRR values are given in Table 2 in Appendix A.

To analyze the initial design documents, we merged the 16 individual students' submissions into 11 teams, using a conservative process where if one team member's document included something (e.g., a design decision or a vulnerability) we considered it present for the team. When considering whether a design was in-depth, we used the deepest of the team members' designs.

Due to the small sample sizes for the design documents, we coded these collaboratively rather than independently. Two coders analyzed the documents iteratively, coding two documents per round independently and then meeting to discuss and resolve differences. The codebook is provided in Table 5 in Appendix A. From this analysis, we were able to explore trends connecting design depth and the resulting security of the code. This allowed us to better understand the cause of different types of vulnerabilities (**RQ1**).

2.3 Break data (RQ2)

During the break phase, we collected break submissions (*artifact*) and commit messages (*account*). While the break and fix phases occurred simultaneously, we describe them separately for simplicity.

Break submissions. Break submissions include test cases demonstrating insecure behavior (as compared to an oracle implementation provided by the instructors), as well as written vulnerability reports (see Section 2.1). As in the build phase, teams were asked to include detailed commit messages with each submission. Teams were prompted to supply a description of the break, their reasoning, and their perception of the difficulty both of finding the vulnerability and generating a working exploit (test case).

Taken together, these two data sources illuminate participants' testing strategy: what do they try, in which order? Do they fully understand the vulnerability they are exploiting, or are they bruteforcing some aspect of it? Do they focus on finding all problems in one target, or try the same (or similar) exploit against multiple teams? This data also provides insight into what types of vulnerabilities are (not) easy to find and to exploit (**RQ2**). Details about the break submissions and commit messages are given in Table 1.

Analysis of Break data. Despite the large number of submitted breaks (N= 275), only a subset were successful and unique (N=52). The two coders analyzed breaks and associated commit messages in rounds of 20, meeting in between to discuss and resolve differences. Codebook details are given in Table 3 in Appendix A.

2.4 Fix data (RQ3)

In the fix phase, we collected fix submissions (*artifact*) and commit messages (*account*).

Fix submissions. Participants patched their code in response to a break by committing an update to their git repository. Teams were incentivized to produce fixes quickly, as they lost more points the longer a break went unfixed. As before, participants were prompted to provide detailed commit messages, this time describing how they fixed the vulnerability, its underlying cause, their confidence the fix resolved the vulnerability, and whether they changed their design (versus just their implementation). Details are given in Table 1.

Submitted fixes and commit messages allow us to understand whether participants understand the vulnerability and can identify the cause in their code (**RQ3**). This data allows us to understand where and why participants struggled to address vulnerabilities (**RQ3**) and provides further insight into whether vulnerabilities were caused by poor design or implementation (**RQ1**).

Analysis of Fix data. The small number of unique breaks led to a correspondingly small number of fix submissions (N=48). As such, we again coded this data collaboratively, with two coders meeting after every 10 submissions to discuss and resolve differences. Codebook details are given in Table 4 in Appendix A.

2.5 Ethics

This study was approved by the University of Maryland Institutional Review Board (IRB). Participants were informed that a study was being run within the course and that by signing up for the course, they were consenting to being a part of the study. This information was clearly presented on the course website, in the course posting, and directly via email when students registered for the course. We also presented this information to students on the first day of class to ensure all students were clearly informed.

To avoid coercion, this 1-credit course was offered during an off-peak semester and was not major-required. Because students were assigned a grade for the course, we tailored the requirements so that a majority of the points were given if the student met all functionality requirements and was active throughout the break/fix phase. Specifically, students could achieve an A even if they were lower-ranked in the competition. Remaining points were allocated based on competition rank to ensure motivation.

2.6 Limitations

Our goal in this study was to understand the challenges faced by developers when pursuing secure development, using students as proxies for professionals. While students often have less experience than professionals, several recent studies conclude they can be adequate substitutes in secure software development studies [36, 53, 54, 80], as many skilled professionals have limited experience with security specifically [2, 4, 11, 30, 33, 48, 67, 74, 83]. However, it is likely students have lower ability to find vulnerabilities as compared professionals. As such, we consider the vulnerability hunting done by students as a lower bound and, as security experts, provide additional review for missed vulnerabilities. Still, we believe lessons from this work can still apply to more experienced developers.

In some ways, our study represents a best-case scenario for security considerations. Because participants were explicitly asked to reflect on their progress and design, they likely put more effort into considering design and evaluating their development process than they otherwise might have. It is possible that encouraging regular, atomic commits could improve code security, as this is recommended best practice [27], but prior studies of this relationship have been inconclusive [45]. Additional reflection about the reasoning behind commits is helpful for development [38] and learning generally [25, 69], so our results likely demonstrate an upper bound. However, this allows us to understand development progression, misunderstandings of security and requirements, and design approaches. Relatedly, participants were aware they needed to include security in their system from the outset, which may not accurately reflect general developer experiences. However, this allows us to understand how vulnerabilities that occur even when developers explicitly consider security from the beginning.

Our results also have the normal limitations of field work, including that we can only observe associations, not causality. Our study was conducted with a limited sample size (14 teams) at only one institution. Despite its limitations, field work is a common practice in HCI research [39] and has been shown to be useful in security research [37, 64, 77, 82]. The BIBIFI contest does not perfectly simulate realistic development settings. Our participants were able to choose their own development environments and programming languages, which is often not the case in company settings. Our participants were tasked with building an entire (small) system from scratch rather than the common developer tasks of modifying or managing existing codebases. Despite these differences, studying the reason for (in)secure development from this lens gives us insights that may prove useful for improving broader practices.

3 DEVELOPMENT AND VULNERABILITIES (RQ1)

Our analysis of code submissions identified 145 unique vulnerabilities introduced throughout the build round and 79 unique vulnerabilities remaining in participants' code at the conclusion of the build round, with a median of 12.5 vulnerabilities remaining per team. Here, unique refers to non-repeating per team vulnerabilities; the same type of vulnerability could count for more than one team. Throughout Sections 3.1-3.3, we use distinct to refer to distinct types of vulnerabilities, such that two teams could have the same *No Implementation*, *Misunderstanding*, or *Mistake*, but it is only counted once. Vulnerabilities in build submissions were only labeled once teams attempted to implement the missing functionality as mentioned in Subsection 2.2. These vulnerabilities can be categorized into 10 different issues. We employed a similar codebook to the one used by Votipka et al. [83] when analyzing prior BIBIFI submissions, re-deriving codes from the original codebook. Our codebook includes one addition, as our specification required a mechanism for timing out when input is no longer received from a user script, where prior iterations did not.

This section presents vulnerability descriptions and examples, as well as relationships between the software development techniques and strategies teams employed and the vulnerabilities they introduce. Throughout this section, we use V to represent the number of vulnerabilities present and T to represent the number of teams. We select illustrative examples in each of the three major vulnerability categories (*No Implementations*, *Misunderstandings*, and *Mistakes*), but do not comprehensively describe all vulnerability types in each category; as such, the V values within each subsection do not always sum to the total for that category. A comprehensive accounting of all vulnerabilities for each phase of the competition is given in Table 6 in Appendix B.

We coded vulnerabilities based on Votipka et al.'s three high-level categories: *No Implementation*, *Misunderstanding*, and *Mistake*. Our results closely mirror the distribution found in the multiuser database problem (most similar to our project) with slightly fewer *No Implementation* vulnerabilities in our iteration. The tighter control of the classroom setting in our study likely influenced teams to implement needed access control checks, but teams in our study still missed many checks, as described below. We are able to further expand the description of the vulnerabilities from Votipka et al.'s paper by explaining the reason for and cause of the different vulnerabilities.

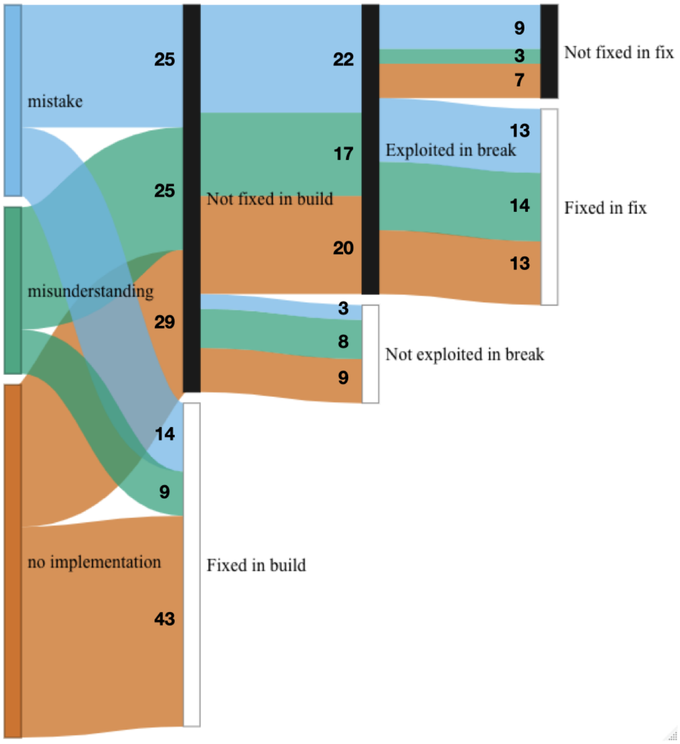

Figure 1: Vulnerability types throughout each phase.

3.1 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).

Teams implement more obvious security features, only missing them when they are rushed. Six teams had a total of 29 distinct *No Implementation* vulnerability types present in their projects after the build phase. Teams were explicitly told to implement access control and issue a timeout when the IoT controller does not receive an indication of termination or input from a user-issued script after 30 seconds (making them *All Intuitive* requirements). Three teams missed this *All Intuitive* requirement (V = 3, T = 3).

Three teams missed some required access control checks (V = 7, T = 3). These teams waited to build the functionality of the primitive commands until late in the build phase and then forgot to add the necessary access control checks, introducing these vulnerabilities within the last 3 days. These teams built their access control checks into each individual primitive command function, rather than building one function to handle all access control checks, so they would have needed to write an entire new check for these last-minute additions (T = 3). We further describe the effect of development timeline on their code security in Subsection 3.4.

Teams fail to implement less obvious security features. Most *No Implementation* vulnerabilities resulted from teams missing the *Unintuitive* requirement of a recursive delegation check (V = 19, T = 5). According to the specification, a principal p has a right r on a variable x if there exists some principal q that has r on x and the current security state includes an assertion that q is delegating r on x to p. Based on this description, teams were expected to manage delegations in a tree-like structure, where loss of permissions from a parent results in loss of permissions to the child. This means that teams must check and manage the rights of children and parents simultaneously at time-of-use. This requirement was not explicitly stated in the specification, meaning that teams had to infer its implementation from the requirements. As mentioned by T6, the reason for this vulnerability in their system was because they "didn't realize [the] system needed to search through all assertions," which was common among the teams that addressed this vulnerability. Teams that introduced this vulnerability often built their code using individual access control checks for each primitive command rather than creating one function to manage it (T = 5).

3.2 *Misunderstanding*

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

Teams conceptually misunderstood access control requirements. Eleven teams had a total of 34 distinct *Misunderstanding* vulnerability types throughout the build phase, with 25 remaining at the end. The most pervasive *Misunderstanding* was caused by incorrect access-control assumptions (V = 20, T = 10). For example, two teams attempted to implement recursive rights but did not properly remove rights for participants in a delegation chain. When a parent delegator lost a right on a variable, these teams would remove the this right from all of its delegatees, correctly implementing recursive rights. However, these teams would remove every instance of this right from the delegatee, incorrectly removing delegations of this right from other parent delegators, causing an availability violation. T4 said this vulnerability was caused by poor "assumptions .... I assumed we always needed to revoke the permission recursively, but I never actually checked if the user still had access." This is a conceptual *Misunderstanding* of the access control requirements, because they thought that they should remove all rights rather than just a subset. In this case, the team had a a fundamental *Misunderstanding* of the requirements in the specification, which was fairly common among teams (T = 3). Other vulnerabilities related to special accounts in the system and their restricted capabilities, how to handle the permissions for a specific primitive command, and how to handle recursive delegation.

Teams implement most of the access control requirements, but fail to consider all cases. The most common cause of *Misunderstanding* vulnerabilities was a team missing a specific subcase of a requirement (T = 4). These included building recursive delegation but failing to account for circular delegations, failing to check the running principal during delegation, and allowing another principal to delegate on behalf of the delegator. These teams did attempt to implement recursive delegation (unlike *No Implementation*) and did not fundamentally misunderstand the underlying concepts. They just misunderstood a single facet of the requirement.

Teams overgeneralized security requirements based on provided tests. Participants also struggled with building input-handling code to correctly handle user scripts (V = 5, T = 4). For example, four teams assumed all user scripts would end with a newline character and wrote their code accordingly (by reading input line by line). This resulted in an availability violation, as an attacker could generate a script that did not end in a newline —a correct script according to the specification—causing the parser to hang and prevent future connections. T2 identified the cause of this vulnerability in their code as being "an assumption about the input cases. I assumed it was not valid if a newline was not present at the end. Since all tests contained one." Note that the specification makes no mention that the scripts must end with a newline. This team overgeneralized assumptions from the provided examples, which caused them to not fully consider how the timeout setup should work.

3.3 *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.

Teams failed to test for edge cases beyond provided tests. Eleven teams had a total of 25 distinct *Mistake* vulnerability types present at the end of, and 39 throughout, the build phase. The most common issue was insufficient error checking (V = 13, T = 8), commonly a lack of null value checking during script parsing. This caused the server to crash on certain attacker-crafted malicious scripts, leading to availability violations (V = 7, T = 5). These crashes were restricted to availability exploits because teams exclusively used memory-safe programming languages. However, if teams were under performance or cultural pressure from their employer to use an unsafe language, these vulnerabilities could have led to even more serious exploits. Teams often attributed these *Mistakes* to missing an edge case when implementing functionality or security: specifically not checking for certain values (T = 3). T8 attributed a *Mistake* in their code to the fact that they "did not have a null check before attempting to access an element." These *Mistakes* were often introduced within the last 3 days of the build phase (V = 7), making them more difficult to catch.

Teams failed to test for security. Other *Mistakes* included teams skipping a step while trying to implement security (V = 7, T = 6) and incorrect implementations of control flow (V = 4, T = 4). Our participants noted two causes for these vulnerabilities: code that runs functionally but not securely (T = 1), and code that runs securely except for a specific input (T = 2). For example, teams were required to roll back all changes in the event of a failure or security violation while processing a user script. T6 set up their system so that when a status of "FAILED" was returned, they would not commit any of the changes from this set of primitive commands. However, while testing, to simplify debugging, they changed the code to print "FAILED" rather than return it and simply forgot to revert this change. The code containing this vulnerability would run functionally but would not meet the security requirements of the specification. Given that the tests provided were centered on functionality, teams did not uncover these vulnerabilities unless they specifically tested their own code for security.

3.4 Development approach's security impact

We next review trends relating different development strategies to the introduction of different types of vulnerabilities (**RQ1**).

Detailed design is common with fewer vulnerabilities. Analyzing the final submitted design documents, we found four teams submitted a detailed design document, four submitted a design document containing minimal design considerations, and three submitted a design document containing few to no design considerations. Teams with detailed design explicitly described how they were planning to implement access control. For example, T1 designed explicitly for access control and transitive rights by using "a directed graph to represent the access rights available to each principal for each variable or rule. Nodes in the graph represent each principal, while edges, labeled with one of the four access types, represent access delegation from one principal to another." Teams with minimal design considerations mention of the necessity of a security feature but do not describe how they plan to address it in their code. In their design, T8 notes that "tampering could be prevented by keeping a strict record of what users have what rights," but they do not mention any details about how they will store or manage this. Teams with little to no design considerations do not mention access control or authentication at all.

When comparing a team's design depth to the security of their submission, we observed teams with detailed designs also tended to introduce fewer *No Implementation* and *Mistake* vulnerabilities. The four teams with detailed design only accounted for one (5%) of the *No Implementation* vulnerabilities and 33% of the *Mistake* vulnerabilities. Conversely, the four teams with minimal design accounted for 56% and 42% of the vulnerabilities respectively. For example, T8 describes one possible threat: "tampering could happen if a user writes to data values it does not have write access to." However, they provide very little detail about how they intend to mitigate this attack: "tampering could be prevented by keeping a strict record of what users have what rights to each variable, sensor, output device, or rule." T8 makes no mention of how they will manage this record or the need for recursive delegation checks, and they failed to implement recursive delegation checks. Similarly, we note that teams that failed to implement the *All Intuitive* timeout requirement (T = 3), missed some access control checks (T = 2), and failed to implement recursive delegation checks (T = 4) failed to make mention of these requirements within their design documents.

It seems plausible that developer experience, instead of just better planning, might cause both better planning and better outcomes. However, course teams exhibited little variation in experience: every team had at least one member self-rated as "novice" in secure development experience and "intermediate" in software development experience. All teams but one had at least one member (most teams both) with academic secure development training.

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 (T = 3). For example, T7 had a detailed design document in which they describe how they will handle the access control requirement described in the specification through the use of "an access control component that controls authentication and monitors which principal should have access to what" in which they "maintain a list of 'direct' objects that keep track of which variables principals can access directly and 'delegate objects' that keep track of rights that have been delegated… and ensures that principals delegating rights are actually permitted to do so." However, despite their detailed design, T7 failed to consider the possibility of circular delegation chains, in which the parent delegator and the last account to receive permissions are the same. This results in a *Misunderstanding* vulnerability, which caused an infinite loop during a delegation check. 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 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. We plot the number of functionality and security commits each day for several notable teams in Figure 2.

As seen in Figure 2, the team that had the fewest vulnerabilities, T1, did no security work on the very last day of the build phase (day 9). T1, which also had no vulnerabilities at the conclusion of the build phase, started implementing their access-control code on day 2 of the build phase and edited it slowly over the following 6 days. In contrast, T11, which had the fourth most vulnerabilities at the conclusion of the build phase, had security commits up until the last day and did not start implementing their access control code until day 6 of the build phase. They built the majority of the access control functionality on day 8. On day 8, they introduced 20 different vulnerabilities, only fixing about half of them before the conclusion of the build round (12).

T11 concluded the build round with 9 vulnerabilities in their code, 8 of which were introduced on day 8. Further we note that teams that waited to implement access control until late in the build phase often ran out of time to implement recursive delegation checks (T = 3). For example, T11 was still implementing basic access control on the last day of the build phase, causing them to run out of time to implement recursive delegation checks despite building this requirement into their design.

Good development practices can help with security, but do

not make up for no design. We note some interesting ways in which good development practices seem to interact with the resulting code security. Teams T2 and T7 had a good design, but both waited until late in the build phase to add security to their codebase, rather than building it in incrementally. T7 introduced 5 different vulnerabilities related to access control, 4 of which occurred on the same day. On the other hand, T9 provided only a minimal design document but built security in gradually over time, ending the build-phase with the second fewest vulnerabilities.

Despite this general trend, we can see that T8 seemingly did everything correctly. They started building for security early and continued steadily throughout the build phase. So what went wrong? The vulnerabilities introduced by T8 early in the build phase relate to poor design decisions or a lack of design. T8 introduced nine different access-control-related vulnerabilities during the build phase, including seven before day 6. These vulnerabilities were conceptual in nature, and they were slowly introduced over each small security commit. Building incrementally does not make up for not designing for security in depth. Other T8 vulnerabilities caused by a *Mistake* in implementation were introduced in the last 3 days of the build phase when T8 was likely rushing to complete their project.

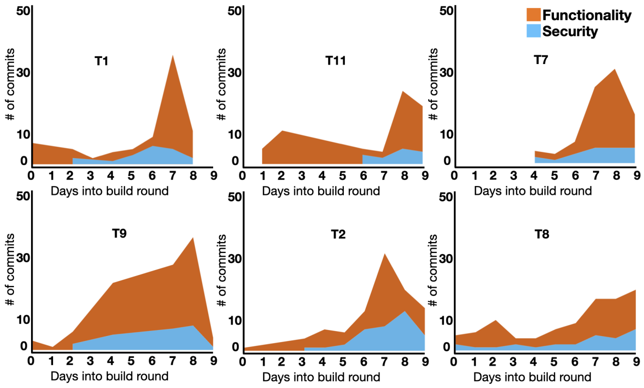

Figure 2: Number of functionality and security commits through the build phase for select teams.

4 ANALYSIS OF (UN)EXPLOITED VULNERABILITIES (RQ2)

This section details the vulnerabilities that were (not) exploited during the break round. We use the same type descriptions in Section 3 to describe the exploits (not) uncovered.

Teams submitted 104 total valid breaks, resulting in 52 non-repeating breaks (where the breaking team did not exploit the same thing more than once in the same team). Teams left 19 instances of vulnerabilities unfound and not exploited. When discussing vulnerabilities (not) exploited, we use E to represent the number of distinct vulnerabilities exploited, T to represent the number of teams that found this exploit, and U to represent the number of vulnerabilities unexploited. We note that one team can exploit the same vulnerability in multiple teams, as the same vulnerability can exist in multiple programs. For example, two teams might both fail to implement recursive delegation checks, so it is possible for E to exceed T. Throughout this section, we select illustrative examples within the three main vulnerability categories rather than discuss every specific type of vulnerability in detail, so E and U do not always sum to the total for *No Implementations*, *Misunderstandings*, and *Mistakes*.

4.1 *No Implementation*

Of the 29 distinct *No Implementation* vulnerabilities present after the build phase, 20 were exploited in the break phase. These related to missing some access control (E = 3, T = 5), not timing out (E = 3, T = 2), and missing recursive delegation checks (E = 14, T = 2).

Missing access control found when checking related issues. In general, teams exploited these missing access control and recursive delegation checks while testing a tangential, but related, access control requirement, rather than via targeted attacks where they reviewed code for a specific vulnerability. For example, T8 exploited a specific missing access control check on a primitive command by testing the target code for a vulnerability related to circular delegation, a tangential access control requirement. They were not specifically testing to see if the target team failed to implement access control on this primitive command. Rather, they were testing for a different access control requirement and found this vulnerability. However, breaks exploiting a lack of timeout were targeted: discovered explicitly by testing timeouts. For example, T3 exploited two timeout vulnerabilities by "testing [malformed user-scripts] and seeing if that caused [programs] to hang."

Teams target more glaring issues rather than complex vulnerabilities. Nine *No Implementation* vulnerabilities were left unexploited: some pertaining to missing some access control checks (U = 4) and missing recursive delegation checks (U = 5). Notably, of the 9 unexploited *No Implementation* vulnerabilities, none were the *All Intuitive* no timeout vulnerability. The no recursive delegation vulnerabilities that were left unexploited were in code that had other, more glaring issues such as problems with input handling or timing out; teams generally favored attacking these issues rather than the slightly more complex vulnerabilities. As breaks that exploited missing access control checks were often found incidentally while targeting other vulnerabilities, these vulnerabilities were likely left unexploited because they were not accidentally triggered. 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.

4.2 *Misunderstanding*

There were 25 distinct *Misunderstanding* vulnerabilities present after the build phase, and 17 were exploited in the break phase. The exploited vulnerabilities included incorrect access control assumptions (E = 13, T = 5) and incorrect assumptions about possible user script inputs (E = 4, T = 6). Specifically, vulnerabilities due to incorrect access control assumptions were overwhelmingly due to a misunderstanding of requirements associated with recursive delegation checks, where teams attempted to implement the recursive check but misunderstood a sub-piece or conceptual piece of the requirement (E = 6, T = 4).

***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. For example, T4 tailored their break to test "that there are no availability issues when creating/deleting a circular chain" rather than broadly attacking access control or recursive delegation checks.

Eight *Misunderstanding* vulnerabilities were left unexploited, mostly related to incorrect access control assumptions (U = 7). These vulnerabilities were comprised of rules around the delegation abilities and restrictions of the running principal of the user script (U = 3), rights management of special principals (U = 2), managing rights when the recursive delegation chain is broken (U = 1), and conceptual misunderstandings of rights management for specific primitive commands (U = 1). Given exploiting *Misunderstandings* requires deeper knowledge and specifically crafted exploits, these vulnerabilities were likely left unexploited since no single break submitted targeted these vulnerabilities in any project.

4.3 *Mistake*

Of the 25 *Mistake* vulnerabilities present at the end of the build round, 22 were exploited in the break round making vulnerabilities due to a *Mistake* the most likely to be exploited. The exploits of vulnerabilities from the build round were attributed to insufficient error checking (E = 13, T = 7), teams skipping a step when implementing security or functionality (E = 6, T = 8), mistakes associated with the control flow of the application (E = 2, T = 5), and an uncaught runtime server error (E = 1, T = 3). This mirrors results from prior work by Votipka et al. [83] in which they find that *Mistake* vulnerabilities are almost always exploited.

***Mistake* vulnerabilities were exploited incidentally.** Teams exploited nearly all *Mistake* vulnerabilities incidentally, while targeting an unrelated vulnerability. T6 exploited a control flow issue in a team's code, in which they forgot to remove debugging statements, while attacking the implementation of transitive delegation in a few teams' codebases. 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 (T = 4). 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.

Only 3 *Mistake* vulnerabilities went unexploited. Two were vulnerabilities resulting from control flow issues in teams' programs and one was from a team skipping a step when implementing security checks. Compared to the exploited control flow mistakes and skipping steps, the unexploited mistakes were buried more deeply within teams' code and required very specific tests to exploit.

5 ANALYSIS OF FIXED VULNERABILITIES (RQ3)

This section describes the details and characteristics of vulnerabilities that were fixed and not fixed. As in the previous section, we use the vulnerability type descriptions from Section 3.

In total, our participants fixed 66 vulnerabilities in their code during the build round (T = 12) and 30 vulnerabilities in their code during the fix round (T = 11). 39 vulnerabilities were left unfixed (19 exploited) at the study's conclusion. When discussing the (un)fixed vulnerabilities throughout this section, we use V to represent the number of vulnerabilities and T to represent the number of teams that this issue was (not) fixed by. As before, we select illustrative examples for the three main vulnerability categories, such that V does not always sum to the total for *No Implementations*, *Misunderstandings*, and *Mistakes*.

5.1 *No Implementation*

During the build phase, *No Implementation* vulnerabilities were fixed the most (V = 43, T = 11). Most were related to a lack of implemented access control (V = 22, T = 7). Several teams started by building their code focused on functionality rather than access control (T = 7) and added access control further into development. Generally, these teams started by adding basic access control and then revisited their code to add checks of the delegation chain (V = 15, T = 6). Five of the six teams that addressed recursive delegation checks did so within the last 2 days of the build phase. *No Implementation* vulnerabilities were largely introduced and fixed as teams worked through the build phase to implement the requirements of the specification, starting with *All Intuitive* requirements and getting to *Unintuitive* requirements if there was time left.

***No Implementation* fixes require restructuring the program.** While many *No Implementation* vulnerabilities were fixed during the build phase, over half of them were left unfixed at the conclusion of the study (V = 16, T = 4). Of those left unfixed, seven were exploited during the break phase. The unfixed vulnerabilities were caused by missing access control checks (V = 2, T = 1), missing recursive delegation check (V = 3, T = 3), and missing a timeout in the code (V = 3, T = 3). Three of these vulnerabilities were associated with T12, including the missing recursive delegation check. T12 did not implement their access control checks with transitive rights in mind, failing to include recursive delegation checks and implementing many access control checks throughout the codebase rather than in one single access control function. Despite exploitation, this vulnerability remained unfixed within their code, as a fix for this would have required T12 to significantly alter their codebase, changing each access control check throughout their codebase and redoing their entire access control system to account for parent-child rights. The two other teams with this vulnerability present in their code at the conclusion of the study required significant alterations to their codebase to address this vulnerability owing to the fact that their access control was spread throughout their codebase. This points to the importance 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.

5.2 *Misunderstanding*

During the build phase, vulnerabilities caused by *Misunderstanding* access-control requirements were the least likely to be fixed (V = 9, T = 6). The only *Misunderstanding* vulnerabilities fixed in this phase were due to incorrect access control assumptions (V = 9, T = 6), such as handling on special principals (V = 3, T = 2) and requirements around who can change a principal's password (V = 1, T = 1). Similarly, these were the most likely to be fixed *Misunderstandings* in the fix round (V = 10, T = 7).

***Misunderstanding* vulnerabilities are typically only fixed when pointed out.** Vulnerabilities caused by *Misunderstanding* 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). For example, T6 misunderstood the requirements for parent child rights, but found and fixed this vulnerability during the build phase because it "didn't run on a test."

However, instructor-provided tests only covered fairly 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. For example, T9 had a vulnerability in their code involving the delegation of rights. When a user script is submitted to the system, the script starts by logging in as a principal. This principal (PA) may delegate rights to another principal (PC) through the use of a third principal (PB). This is not a problem as long as the permissions of PB are checked and they have the necessary permissions. However, T9 misunderstood this requirement and only checked the permissions of the running principal (PA). This vulnerability was not fixed by this team until it was exploited during the break round, despite the fact that T9 directly altered the code containing this vulnerability twice before the conclusion of the build round. Receiving detailed input about security misunderstandings in their code allowed T9 to address this issue and understand where they went wrong. Overwhelmingly, *Misunderstanding* vulnerabilities were fixed once they were pointed out and explained to teams (V = 13, T = 8), 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. For example, T6 began testing other teams for a vulnerability related to incorrectly removing rights once a delegation chain had been broken after this *Misunderstanding* was exploited and fixed in their own codebase. They said they found the vulnerability because it was a "break [in my code] found by [an] enemy team."

Addressing *Misunderstanding* requires time. Similar to the build phase, vulnerabilities caused by a security requirement *Misunderstanding* were left unfixed the most at the conclusion of the study (V = 11, T = 9). Three of these vulnerabilities were exploited during the break phase. These exploited vulnerabilities were due to incorrect assumptions about the access control requirements (V = 3, T = 4). Teams did not design at all for these vulnerabilities (V = 3, T = 3), and two of these vulnerabilities were submitted in the last day of the break phase, leaving teams little time to address something they had not designed for.

5.3 *Mistake*

***Mistake* vulnerabilities are found and fixed with testing.** During the build phase, vulnerabilities caused by an implementation *Mistake* were mostly left unfixed (V = 14, T = 6). The most frequently fixed were related to control-flow errors (V = 12, T = 5). These vulnerabilities were easy to find through testing, as seen by break teams exploiting them incidentally using a variety of tests.

Conversely, vulnerabilities related to insufficient error checking (V = 6, T = 4) and missing an implementation step (V = 3, T = 3) were the most fixed *Mistakes* in the fix phase. Teams that did not think to handle edge cases related to insufficient error checking or skipped algorithmic steps were, unsurprisingly, also unlikely to think to test for these cases while building. Five of these vulnerabilities were exploited in the last day of the break phase, leaving teams little time to address them despite how easy many fixes could be. For example, a vulnerability in T7's code caused by insufficient error checking was exploited on the second-to-last day of the break phase. T7 had failed to check certain commands for syntactic validity, leading to a crash if these commands were improperly formatted. T7 already had a function to check for syntactic validity, so this vulnerability could have been fixed by adding a single line of code invoking this already-existing function in an additional place. However, this vulnerability went unfixed in the short available time window.

Addressing more complex vulnerabilities leaves little time to address simple *Mistakes*. Eleven *Mistake* vulnerabilities were unfixed in our participants' code (V = 12, T = 6) at the conclusion of the study. Nine of these *Mistakes* were exploited. These exploited vulnerabilities were divided among a vulnerability due to insufficient error checking (V = 6, T = 3) and skipping an algorithmic step while implementing security (V = 3, T = 3). Two of the teams with unfixed *Mistakes* had the most and second most successful breaks against them during the break phase leaving them with many issues to address with limited time. Additionally, two teams fixed vulnerabilities caused by a lack of timeout and recursive delegation check, which took considerable time to address. Despite the relative ease of addressing mistakes, teams may not have had time to get to them before the conclusion of the study.

Most unfixed vulnerabilities are easy to exploit. To explore the possible repercussions of unfixed vulnerabilities, we look at how attacking teams and members of the research team (with security experience) rated the difficulty to find and exploit these unfixed vulnerabilities. Overall, we find that, of those that they did rate, teams rated many unfixed *Mistake* vulnerabilities as easy to find and exploit (Find = 7/7 and Exploit = 6/6). Given that *Mistakes* were the least unfixed of any vulnerability, this is not particularly concerning. However, vulnerabilities associated with *Misunderstandings* were also overwhelmingly rated as easy to find and exploit by our teams (Find = 8/12 and Exploit = 10/12). Our experts rated *Misunderstanding* vulnerabilities as less easy to find and exploit than participants (Find = 5/12, Exploit = 7/12), but this points to a possibly alarming trend in which vulnerabilities that are difficult to fix and often weren't fixed until exploited are an easy point of exploitation for possible attackers. Once the vulnerability was discovered, it was easy to craft an exploit. For example, T9 had an unfixed vulnerability associated with removing rights transitively. An exploit for this could easily be crafted in two user-issued scripts, whereas the fix would require rewriting the access control functionality of the system, meaning the effort required from an attacker is minimal, but addressing the vulnerability requires significant changes to the design and implementation of the system.

6 DISCUSSION AND RECOMMENDATIONS

Our results emphasize the importance of existing recommendations and solutions.

Vulnerability classes differ in more than just content. Throughout our data we note that the different types of vulnerabilities (*No Implementation*, *Misunderstanding*, and *Mistake*) fall into distinct classes. These vulnerabilities occur differently from one another: *No Implementations* result from a failure to realize that a feature was needed whereas *Mistakes* result from a failure to thoroughly test the codebase. These vulnerabilities are found differently from one another: *Misunderstandings* were found through explicitly crafted tests whereas *Mistakes* were found incidentally, while testing for a completely unrelated vulnerability. Each of these vulnerabilities needs their own strategy to prevent and address: thorough, well thought out design to address *No Implementations*, consultation and assistance from security experts to address *Misunderstandings*, and broad, complete testing to address *Mistakes*. Employing a single secure development strategy is not sufficient enough to prevent all three classes of vulnerabilities, and a variety of strategies and solutions will be necessary to promote secure development. Throughout the discussion, we'll talk in detail about solutions for the three different classes of vulnerabilities.

Importance of using security tools and comprehensive testing to uncover *Mistakes* confirmed. Nearly every team had at least one *Mistake* vulnerability at the conclusion of the build round (many teams more than one). These vulnerabilities were nearly all exploited during the break round, often incidentally through testing for other vulnerabilities, and were largely attributed to teams not testing for or considering an edge case or null value. These vulnerabilities can generally be caught or prevented by existing secure development tooling such as secure programming languages, fuzzers, and static analyzers [5, 8, 9, 22, 28, 44, 51, 68, 78, 79].

However, teams often struggled to thoroughly test their code for vulnerabilities during the build round but comprehensively tested other teams' code during the break round, finding and exploiting a variety of vulnerabilities. This points to a lack of priority for testing their own code thoroughly, but not a lack of knowledge to do so. Possibly if teams were given more time to complete their project, they would have used this time to perform thorough testing on their own codebase. This highlights that developers likely do not lack the knowledge to thoroughly test their code, given the time and encouragement to do so.

Importance of incremental development and minimally trusted code reaffirmed. Our results suggest the importance of following several security best practices. Teams that worked incrementally on security and built for security and functionality simultaneously, rather than focusing on functionality and then security, introduced less vulnerabilities in their code throughout the build round. Notably, teams with a good design that waited to implement security had more vulnerabilities than some teams with minimal or no design that started early on security. However, starting early on security did not make up for conceptual misunderstandings of security requirements.

Additionally, how teams built their security code played an important role. We note that teams that failed to fix vulnerabilities related to the transitivity of the access control system had individual access control checks for each command rather than building one function. This meant that when these teams wanted to fix this vulnerability they had to redesign their entire system. Moreover, every team that failed to handle the transitive nature of the access control system implemented their access control checks in this way. This fails to follow the security recommendation of minimal trusted code [15] and points to the benefit of of following best practices in implementing security code.

Detailed design is important to secure development, but not a silver bullet. Design appeared to play an important part in the security of our participants' code. More broadly, we found that teams with a detailed design were less likely to miss implementing access control and make mistakes. Teams that misunderstood a sub-case or sub-requirement of an access control check did not design for the requirement at all. However, detailed design isn't a silver bullet. If teams encode an initial *Misunderstanding*, they aren't likely to catch the vulnerability at implementation as they were likely to stick with their designs. A detailed design is important to the secure development process as it helps prevent missing security requirements or introducing programming mistakes, but it is important to ensure the design is secure from the start.

Importance of including security experts in development lifecycle to help uncover *Misunderstandings*. Our results suggest that participants struggle more with misunderstanding security concepts than implementing them. This echoes prior work from Votipka et al. [83] and points to a need for security expertise within the development process. Teams encoded these misunderstandings into their design documents and failed to catch them at implementation time. They often attributed these vulnerabilities to a fundamental misunderstanding of the security requirements, and they were unable to understand them and address them until they were pointed out and explained to them (exploited during the break round). Once the breaking team pointed out and explained the vulnerability, *Misunderstandings* were almost unanimously fixed. The ability to have the vulnerability explained to them allowed teams to address these issues, improving the resulting security of the code. The presence of a security-knowledgeable developer or security professional during the design process to help point to these vulnerabilities and explain their importance and severity can improve the security knowledge of developers, while simultaneously improve the security of the codebase.

7 RELATED WORK

Prior work has used the BIBIFI contest to explore factors relating to secure development, using post-hoc analysis of contest submissions [67, 83]. This paper extends this work through analysis of development processes, allowing us to uncover how and why vulnerabilities are introduced into code, collecting context without relying on participants to recall experiences later.

This paper also complements other prior work examining the introduction and discovery of vulnerabilities, as well as the incorporation of security into the development lifecycle.

Measuring vulnerabilities through metadata. Prior research has leveraged metadata available in version-management systems to explore the introduction, identification, and remediating of vulnerabilities. While investigating the data associated with commits from PHP and the Apache HTTP server, Meneely et al. found that codebases with new committing authors, higher-than-average number of commits, and committers that edit others' code were associated with more frequent vulnerabilities [45, 47]. In follow-up work, they explored whether Linus' Law, that having more eyes on code improves its quality, applies to vulnerabilities [46]. Investigating Chromium, the authors found that source code reviewed by more developers was more likely to have a vulnerability remaining, unless the reviewer had prior vulnerability patching experience. Within the OpenBSD operating system, Ozment and Schechter investigated whether vulnerabilities were introduced during initial feature development or changes committed later, finding that more than half of all reported vulnerabilities were introduced during initial development [63]. Other work has considered the relationship of vulnerabilities to associated code dependencies and segments of the code base, in Eclipse [70], Firefox [59], Windows Vista [84], and RedHat packages [58], finding that vulnerabilitys tended to cluster in particular components. Similarly, there has been significant work investigating vulnerability lifecycles, measuring the time between when a vulnerability is found and when it is remediated [3, 20, 41, 57, 72], investigating the correlation of similar metrics with delays in patch time. Measurements like these enable a high-level view of vulnerability characteristics; we expand on these findings by directly observing development processes, within a fixed project specification, to more concretely learn why and how vulnerabilities occur.

Measuring vulnerabilities through user studies. In addition to measuring production code, researchers also investigate software security by studying developers. Exploring how cryptographic API usability impacts code security, Acar et al. conducted a controlled experiment with developers, uncovering that APIs designed for usability can help improve the security of resulting code [1]. Oliveira et al. identified how developers' "blind spots" about how an API works correlate with decreased ability to answer security questions or identify security problems in associated code [62]. Naiakshina et al. found that freelance developers did not store passwords securely without prompting, had misconceptions about how to store passwords securely, and often used outdated approaches [54].

Our work complements these studies; a larger, cooperative project over a longer timeframe provides additional ecological validity for observing how vulnerabilities are introduced, found, and fixed.

Security during the development process. Some research has aimed to broadly understand security in the software development lifecycle. Assal and Chiasson interviewed professional developers, finding a wide range of security practices that often diverged from accepted best practices [6]. Follow-on work, using a survey of North American 123 developers, found developers are often motivated to write secure code but face organizational obstacles [7].

Exploring company-specific secure-development practices, Haney et al. used 21 interviews to understand organizational challenges during development of products that include cryptography [31]. They found that these developers exhibited a strong security mindset that informs organizational best practices and encourages educational initiatives to expand organizational security knowledge. To understand how security strategies broadly (bug bounty programs, red, blue, and purple teams, and third-party contractors) are applied in practice and the associated challenges, Alomar et al. interviewed 53 security practitioners tasked with vulnerability discovery. They found that vulnerability discovery is often stymied by costs, staffing issues, trust, and communication [4].

In an ethnographic study, Palombo et al. embedded a researcher into a software company for 1.5 years to study secure-development practices. They found developers sometimes do not fix, or even intentionally introduce, vulnerabilities because of business pressures. [64]. In follow-up ethnographic work, Tuladhar et al. observed a shift in secure development practices when developers understood how security applied specifically to their areas of work [82].

Our work complements these studies in a more controlled observational environment than an ethnographic study, without requiring as much self-reporting and recall as interviews or surveys.

8 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. We uncover trends associated with the introduction, discovery, and fixing of three importantly distinct classes of vulnerabilities. We build a small taxonomy of introduced, exploited, and fixed vulnerabilities and uncover several trends that may influence the introduction, discovery, and fixing of vulnerabilities. Design appears to play an important role in the introduction of vulnerabilities: teams with a detailed design stuck with their design despite the presence of *Misunderstanding* vulnerabilities present, and teams with a detailed design tended to introduce less vulnerabilities overall. Broad testing was useful for uncovering *Mistakes* and *No Implementation*s, but targeted testing was necessary for uncovering *Misunderstandings*. Overall, our results reaffirm the importance of secure development best practices.

ACKNOWLEDGMENTS

We thank the anonymous reviewers who provided helpful comments on drafts of this paper. This project was supported by gifts from Accenture, AT&T, Galois, Leidos, Patriot Technologies, NCC Group, Trail of Bits, Synposis, ASTech Consulting, Cigital, SuprTek, Cyberpoint, and Lockheed Martin; by NSF grants EDU-1319147 and CNS-1801545; and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement 70NANB15H330.

REFERENCES

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

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[3] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*.

[4] Noura Alomar, Primal Wijesekera, Edward Qiu, and Serge Egelman. 2020. " You've got your nice list of bugs, now what?" vulnerability discovery and management processes in the wild. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. 319–339.

[5] Nuno Antunes and Marco Vieira. 2009. 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*. IEEE Computer Society, Washington, DC, USA, 301–306.

[6] Hala Assal and Sonia Chiasson. 2018. Security in the Software Development Lifecycle. In *Symposium on Usable Privacy and Security*. USENIX Association, Baltimore, MD, 281–296.

[7] Hala Assal and Sonia Chiasson. 2019. 'Think secure from the beginning' A Survey with Software Developers. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–13.

[8] Andrew Austin and Laurie Williams. 2011. One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. In *International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, Washington, DC, USA, 97–106.

[9] Dejan Baca, Bengt Carlsson, Kai Petersen, and Lars Lundberg. 2013. Improving Software Security with Static Automated Code Analysis in an Industry Setting. *Software: Practice and Experience* 43, 3 (2013), 259–279.

[10] Lecia Jane Barker, Kathy Garvin-Doxas, and Michele Jackson. 2002. Defensive Climate in the Computer Science Classroom. *SIGCSE Bull.* 34, 1 (feb 2002), 43–47. https://doi.org/10.1145/563517.563354

[11] Veroniek Binkhorst, Tobias Fiebig, Katharina Krombholz, Wolter Pieters, et al. 2022. Security at the end of the tunnel: The anatomy of VPN mental models among experts and non-experts in a corporate context. In *USENIX Security Symposium*.

[12] Jeffrey Bonar and Elliot Soloway. 1983. Uncovering Principles of Novice Programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '83)*. Association for Computing Machinery, New York, NY, USA, 10–13. https://doi.org/10.1145/567067.567069

[13] Maura Borrego, Elliot P. Douglas, and Catherine T. Amelink. 2009. Quantitative, Qualitative, and Mixed Research Methods in Engineering Education. *Journal of Engineering Education* 98, 1 (2009), 53–66. https://doi.org/10.1002/j.2168-9830.2009.tb01005.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.2168-9830.2009.tb01005.x

[14] Jonas Boustedt. 2012. Students' different understandings of class diagrams. *Computer Science Education* 22, 1 (2012), 29–62. https://doi.org/10.1080/08993408.2012.665210 arXiv:https://doi.org/10.1080/08993408.2012.665210

[15] Diana Burley, Matt Bishop, Scott Buck, Joseph J. Ekstrom, Lynn Futcher, David Gibson, Elizabeth K. Hawthorne, Siddharth Kaza, Yair Levy, Herbert Mattord, and Allen Parrish. 2017. *Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity*. Technical Report. ACM, IEEE, AIS, and IFIP. 32–36 pages. https://cybered.hosting.acm.org/wp-content/uploads/2018/02/newcover_csec2017.pdf

[16] Center for Cyber Safety and Education. 2017. *Global Information Security Workforce Study*. Technical Report. Center for Cyber Safety and Education, Clearwater, FL. https://iamcybersafe.org/wp-content/uploads/2017/07/N-America-GISWS-Report.pdf

[17] Pravir Chandra. 2017. *Software Assurance Maturity Model*. Technical Report. Open Web Application Security Project.

[18] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. 2011. 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*. IEEE, 1290–1293.

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

[20] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. 2010. Familiarity Breeds Contempt: The Honeymoon Effect and the Role of Legacy Code in Zero-Day Vulnerabilities. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*. Association for Computing Machinery, New York, NY, USA, 251–260. https://doi.org/10.1145/1920261.1920299

[21] Arthur W Combs, Daniel W Soper, and Clifford C Courson. 1963. The measurement of self concept and self report. *Educational and Psychological Measurement* 23, 3 (1963), 493–500.

[22] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. 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*. Springer-Verlag, Berlin, Heidelberg, 111–131.

[23] Sebastian Dziallas and Sally Fincher. 2016. Aspects of Graduateness in Computing Students' Narratives. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/2960310.2960317

[24] S.A. Fincher and A.V. Robins. 2019. *The Cambridge Handbook of Computing Education Research*. Cambridge University Press. https://books.google.com/books?id=vmAwEAAAQBAJ

[25] John H. Flavell. 1976. Metacognitive aspects of problem solving. In *The nature of intelligence*, Lauren B. Resnick (Ed.). Lawrence Erlbaum Associates.

[26] Rudolf Franz Flesch et al. 1951. How to test readability. (1951).

[27] GitLab. 2022. What are git version control best practices? https://about.gitlab.com/topics/version-control/version-control-best-practices/. (2022).

[28] Google. 2020. Go Programming Language. https://golang.org/. (2020).

[29] Arthur C Graesser, Katja Wiemer-Hastings, Peter Wiemer-Hastings, and Roger Kreuz. 1999. AutoTutor: A Simulation of a Human Tutor. *Cogn. Syst. Res.* 1, 1 (dec 1999), 35–51. https://doi.org/10.1016/S1389-0417(99)00005-4

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

[31] Julie M Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. 2018. " we make it a big deal in the company": Security mindsets in organizations that develop cryptographic products. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. 357–373.

[32] Andrew F Hayes and Klaus Krippendorff. 2007. Answering the Call for a Standard Reliability Measure for Coding Data. *Communication Methods and Measures* 1, 1 (2007), 77–89. http://dx.doi.org/10.1080/19312450709336664

[33] Mohammadreza Hazhirpasand, Oscar Nierstrasz, Mohammadhossein Shabani, and Mohammad Ghafari. 2021. Hurdles for developers in cryptography. *arXiv preprint arXiv:2108.07141* (2021).

[34] Mariana Hentea, Harpal S Dhillon, and Manpreet Dhillon. 2006. Towards Changes in Information Security Education. *Journal of Information Technology Education: Research* 5 (2006), 221–233.

[35] Melanie Jones. 2019. Why Cybersecurity Education Matters. https://www.itproportal.com/features/why-cybersecurity-education-matters/. (2019).

[36] Harjot Kaur, Sabrina Amft, Daniel Votipka, Yasemin Acar, and Sascha Fahl. 2022. Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples. (2022).

[37] Elmer Lastdrager, Inés Carvajal Gallardo, Pieter Hartel, and Marianne Junger. 2017. How Effective is {Anti-Phishing} Training for Children?. In *Thirteenth symposium on usable privacy and security (soups 2017)*. 229–239.

[38] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* 25, 4 (2020), 2416–2449.

[39] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2017. *Research methods in human-computer interaction*. Morgan Kaufmann.

[40] Timothy C Lethbridge, Jorge Diaz-Herrera, Richard Jr J LeBlanc, and J Barrie Thompson. 2007. Improving Software Practice Through Education: Challenges and Future Trends. In *Future of Software Engineering*. IEEE Computer Society, 12–28.

[41] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2201–2215. https://doi.org/10.1145/3133956.3134072

[42] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–23.

[43] Gary McGraw, Sammy Migues, and Brian Chess. 2009. Software Security Framework | BSIMM. (2009). https://www.bsimm.com/framework.html (Accessed 05-22-2018).

[44] Gary McGraw and John Steven. 2011. 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).

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

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

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

[48] Abraham H Mhaidli, Yixin Zou, and Florian Schaub. 2019. " We Can't Live Without {Them!}" App Developers' Adoption of Ad Networks and Their Considerations of Consumer Risks. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. 225–244.

[49] Microsoft. 2019. Microsoft Security Development Lifecycle Practices. https://www.microsoft.com/en-us/securityengineering/sdl/practices. (2019).

[50] Mitre. 2020. CVE. https://cve.mitre.org/. (2020).

[51] Mozilla. 2020. Rust Programming Language. https://www.rust-lang.org/. (2020).

[52] Johnny Salda na. 2014. *The coding manual for qualitative researchers* (2 ed.). Sage.

[53] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. 2020. On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[54] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. 2019. "If you want, I can store the encrypted password": A Password-Storage Field Study with Freelance Developers.. In *Proceedings of the 2019 Conference on Human Factors in Computing Systems (CHI)*, Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.). ACM, 140.

[55] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In *ACM Conference on Computer and Communications Security*. ACM, 311–328.

[56] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. 2018. Deception Task Design in Developer Password Studies: Exploring a Student Sample. In *Symposium on Usable Privacy and Security*. USENIX Association, Baltimore, MD, 297–313.

[57] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *2015 IEEE Symposium on Security and Privacy*. 692–708. https://doi.org/10.1109/SP.2015.48

[58] Stephan Neuhaus and Thomas Zimmermann. 2009. The Beauty and the Beast: Vulnerabilities in Red Hat's Packages.. In *USENIX annual technical conference*. 527–538.

[59] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/1315245.1315311

[60] William Newhouse, Stephanie Keith, Benjamin Scribner, and Greg Witte. 2017. *NIST Special Publication 800-181, The NICE Cybersecurity Workforce Framework*. Technical Report. National Institute of Standards and Technology.

[61] NIST. 2020. National Vulnerability Database. https://nvd.nist.gov/general. (2020).

[62] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. 2018. API Blindspots: Why Experienced Developers Write Vulnerable Code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, Baltimore, MD, 315–328. https://www.usenix.org/conference/soups2018/presentation/oliveira

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

[64] Hernan Palombo, Armin Ziaie Tabari, Daniel Lende, Jay Ligatti, and Xinming Ou. 2020. 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)*. 205–220.

[65] Bridget M Reynolds, Theodore F Robles, and Rena L Repetti. 2016. Measurement reactivity and fatigue effects in daily diary research with families. *Developmental Psychology* 52, 3 (2016).

[66] Tony Rice, Josh Brown-White, Tania Skinner, Nick Ozmore, Nazira Carlage, Wendy Poland, Eric Heitzman, and Danny Dhillon. 2018. *Fundamental Practices for Secure Software Development*. Technical Report. Software Assurance Forum for Excellence in Code.

[67] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. 2016. Build It, Break It, Fix It: Contesting Secure Development. In *ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 690–703.

[68] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *International Symposium on Software Reliability Engineering*. IEEE Computer Society, Washington, DC, USA, 245–256.

[69] Marlene Scardamalia, Carl Bereiter, and Rosanne Steinbach. 1984. Teachability of Reflective Processes in Written Composition. *Cognitive Science* 8, 2 (1984), 173 – 190. https://doi.org/10.1016/S0364-0213(84)80016-6

[70] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. 2006. Predicting Component Failures at Design Time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*. Association for Computing Machinery, New York, NY, USA, 18–27. https://doi.org/10.1145/1159733.1159739

[71] Carsten Schulte and Maria Knobelsdorf. 2007. Attitudes towards Computer Science-Computing Experiences as a Starting Point and Barrier to Computer Science. In *Proceedings of the Third International Workshop on Computing Education Research (ICER '07)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/1288580.1288585

[72] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*. 771–781. https://doi.org/10.1109/ICSE.2012.6227141

[73] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *ACM Conference on Computer and Communications Security*. ACM.

[74] Matthew Smith. 2016. Usable Security—The Source Awakens. USENIX Association, San Francisco, CA.

[75] David Socha and Josh Tenenberg. 2013. Sketching software in the wild. In *2013 35th International Conference on Software Engineering (ICSE)*. 1237–1240. https://doi.org/10.1109/ICSE.2013.6606687

[76] James C. Spohrer and Elliot Soloway. 1986. Novice Mistakes: Are the Folk Wisdoms Correct? *Commun. ACM* 29, 7 (jul 1986), 624–632. https://doi.org/10.1145/6138.6145

[77] Rock Stevens, Daniel Votipka, Elissa M Redmiles, Colin Ahern, Patrick Sweeney, and Michelle L Mazurek. 2018. The battle for new york: a case study of applied digital threat modeling at the enterprise level. In *27th USENIX Security Symposium (USENIX Security 18)*. 621–637.

[78] Larry Suto. 2007. *Analyzing the Effectiveness and Coverage of Web Application Security Scanners*. Technical Report. BeyondTrust, Inc.

[79] Larry Suto. 2010. *Analyzing the Accuracy and Time Costs of Web Application Security Scanners*. Technical Report. BeyondTrust, Inc.

[80] Mohammad Tahaei and Kami Vaniea. 2022. Recruiting Participants With Programming Skills: A Comparison of Four Crowdsourcing Platforms and a CS Student Mailing List. In *CHI Conference on Human Factors in Computing Systems*. 1–15.

[81] Josh Tenenberg and Sally Fincher. 2005. Students designing software: a multi-national, multi-institutional study. *Informatics in Education* 4, 1 (2005), 143–162.

[82] Anwesh Tuladhar, Daniel Lende, Jay Ligatti, and Xinming Ou. 2021. 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)*. 617–632.

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

[84] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *2010 Third International Conference on Software Testing, Verification and Validation*. 421–428. https://doi.org/10.1109/ICST.2010.32

A CODEBOOK TABLES

B VULNERABILITY TYPES PER PHASE

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.

Associated data	Analysis focus	Codes	Description	Agreement
Build code submission	Associated requirement	Setup	Setup team's environment	0.844
		Parser	Parsing commands	
		Prim Cmds	Non-security functionality of primitive commands	
		Authentication	Password checking/initialization	
		Access control	Access control checks	
		Networking	Network communication	
		System state	Managing state of program variables	
		Input handling	Command line/config file input	
		Optional cmds	Non-security functionality of optional features	
		Testing	Testing specific inputs	
		Security fix	Fix of a security bug	
	# lines changed in the code	–	–	–
	Introduced vulnerability	–	Number of vulnerabilities introduced	–
	Vulnerability type	See Table 3	Type of vulnerability introduced	–
	Vulnerabilities fixed	Yes, No	Vulnerabilities fixed in future commit	1
	Fix any prior vulnerabilities		Number of vulnerabilities fixed	1
Build commit message	Associated requirement	Same as code	–	0.899
	Reason for change	Initial	First commit for this piece	1
		Bug fix	Fixed bug	
		Missing functionality	Added feature that was missing	
		Simplification	Making code easier to work with	
		Performance	Increased efficiency	
		Extra feature	Optional cmds	
		Integration	Integrate multiple components	
		Testing	Adding testing to look for bugs	
	How team came up with change	Instructor	Info from instructor	0.888
		Prior exp	Based on knowledge from before	
		Spec	Info from specification	
		Personal preference	Code is more readable/easier for later	
		StackOverflow	Read answers from SO	
		Other tutorials	Other online tutorial (not SO)	
		Teamwork	Discussion with teammate	
		Class forum	Discussion from class forum	
		Oracle	Oracle to determine output	
		Brute force	Tried many things	
		Testing	Realized when testing	
	Design document updated	Yes,No	Whether team updated their design	1

Table 2: Descriptions and agreement of codes for build round

Associated data	**Analysis focus**	**Codes**	**Description**	**Agreement**
Test to run	Target	Team name	Name of target team	–
	Successful	Binary	Was the attack a success	–
	Type	Availability	Type of vulnerability	–
		Confidentiality		
		Integrity		
	Issue	Skipped algorithmic step	Team skipped a step	–
		Insufficient error checking	Vulnerability caused by lack of error chekcing	
		Control flow mistake	Flow of code is unintended	
		Incorrect input assumptions	Assume input is formatted certain way	
		Incorrect use of library	Misuse library function	
		Incorrect access control assumptions	Wrong understanding of access control rules	
		No access control	No access control implemented	
		Uncaught runtime server error	Don't catch exception	
		Resource exhaustion	Don't plan for possible resource exhaustion	
		No timeout	Do not implement a timeout	
	Single/multiple	Single	Vulnerable code in one spot	–
		Multiple	Vulnerable code in multiple spots	
	Difficulty to exploit	Single-step	Exploitable out of box	–
		Multi-step, small	Small manipulations to system	
			needed before it is exploitable	
		Multi-step, large	Many manipulations needed	
	Difficulty to find	Execution	Found by testing spec	–
		Execution and source	Dig into source	
			Goes beyond spec	
		Execution, source,	Deep understanding of concepts needed	
		an underlying concepts		
	Spray	Binary	Vulnerability part of a group of vulnerabilities	–
			submitted against every team	
Description of exploit	Clarity	1 - 5	How clear is	–
			provided description	
	Lines of code	Code included in description	–	
	Example output	Example output included in description		–
	Length	–	# of characters	–
	Readability	–	FRES readability [26]	–
Commit message	Description	See Table ??	Description of what	–
		requirement the vulnerability breaks	–	
	How found	Used spec	Found vulnerability by using specification	–
		Break against themselves	Vulnerability was a break against them	
		Testing	Found vulnerability through testing	–
		Vulnerability during build	Vulnerability in their system during build round	
		Intuition	Used intuition to find vulnerability	
		Brute force	Tried many thigns	
		Exploiting other vulnerability	Found this vulnerability while exploiting another	
		Oracle	Used the Oracle	
		Vulnerability in many systems	Found in many systems	
		Code review	Found by reviewing code	
		Previous issue	Found other exploits for this functionality before	
	Associated requirement	Availability	–	–
		Confidentiality		
		Integrity		
	Fix of a failed break	Binary	Fixed a vulnerability that didn't work before	–
	Difficulty to find	1 - 5	How hard to find vulnerability	–
	Difficulty to exploit	1 - 5	How hard to exploit vulnerability	–

Table 3: Descriptions and agreement of codes for break data

Associated data	Analysis focus	Codes	Description	Agreement
Submission	Fix significance	Low	Amount of change to code/design for fix	–
		Mid		
		High		
	Vulnerabilities introduced	Binary	Other vulnerabilitiesintroduced during fix	–
	Break type	Availability	–	–
		Confidentiality		
		Integrity		
	Issue	See Table 3	–	
Commit message	What was wrong	–	Team description of vulnerability	–
	What caused problem	Table 3	Cause of vulnerability	–
	How fixed	Added missing piece	Added missing thing that causes vulnerability	
		Fixed programming mistake	Fixed mistake that caused vulnerability	
		Adjusted code to reflect spec	Altered code to match specification	
	Design updated	Binary	Did fix require update to design	–

Table 4: Descriptions and agreement of codes for fix data

Associated data	Analysis focus	Codes	Description	Agreement
Design documents	Thoroughness of the design	1 - 5	–	–
	Included components	See Table 2	What requirements did they include	–
	Included relationships	Functionality	Relationships between functionality components	–
		Security	Relationships between security and functionality components	
	Visualizations included	Yes,No	Included visualization in design document	–
	Text included	Yes,No	Descriptive text in design document	–
	Components changed	Yes,No	Did components change from version 1 to 2	–

Table 5: Descriptions and agreement of codes for participant data

		Build		Break		Fix	
Type	**Issue**	**Introduced**	**Fixed**	**Exploited**	**Unexploited**	**Fixed**	**Unfixed**
No Implementation	No access control	22	22	–	–	–	–
	Missing some access control	10	3	3	4	1	2
	No timeout	6	3	3	–	1	2
	No recursive delegation check	34	15	14	5	11	3
	Total	72	43	20	9	13	7
Misunderstanding	Incorrect access control assumptions	29	9	13	7	10	3
	Incorrect input assumptions	5	0	4	1	4	–
	Total	34	9	17	8	14	3
Mistake	Control flow mistake	16	12	2	2	2	–
	Insufficient error checking	14	1	13	–	7	6
	Skipped algorithmic step	8	1	6	1	3	3
	Uncaught runtime server error	1	0	1	–	1	–
	Total	39	14	22	3	13	9

Table 6: Number of various issues in each phase