An Ethnographic Understanding of Software (In)Security and a Co-Creation Model to Improve Secure Software Development

Hernan Palombo* Armin Ziaie Tabari* Daniel Lende Jay Ligatti Xinming Ou
University of South Florida, Tampa, FL, USA
Email: {hpalombo, aziaietabari, dlende, ligatti, xou} @usf.edu

Abstract

We present an ethnographic study of secure software development processes in a software company using the anthropological research method of participant observation. Two PhD students in computer science trained in qualitative methods were embedded in a software company for 1.5 years of total research time. The researchers participated in everyday work activities such as coding and meetings, and observed software (in)security phenomena both through investigating historical data (code repositories and ticketing system records), and through pen-testing the developed software and observing developers' and management's reactions to the discovered vulnerabilities. Our study found that 1) security vulnerabilities are sometimes intentionally introduced and/or overlooked due to the difficulty in managing the various stakeholders' responsibilities in an economic ecosystem, and cannot be simply blamed on developers' lack of knowledge or skills; 2) accidental vulnerabilities discovered in the pen-testing process produce different reactions in the development team, often times contrary to what a security researcher would predict. These findings highlight the nuanced nature of the root causes of software vulnerabilities and indicate the need to take into account a significant amount of contextual information to understand how and why software vulnerabilities emerge during software development. Rather than simply addressing deficits in developer knowledge or practice, this research sheds light on at times forgotten human factors that significantly impact the security of software developed by actual companies. Our analysis also shows that improving software security in the development process can benefit from a co-creation model, where security experts work side by side with software developers to better identify security concerns and provide tools that are readily applicable within the specific context of the software development workflow.

1 Introduction

It has long been recognized that human factors play a dominant role in ever-present software vulnerabilities, with substantial research devoted to this area [1–10]. These past efforts have used a variety of research methods including surveys, interviews, controlled experiments, studying code artifacts, and analyzing data collected from secure-coding competitions. It is also understood that there is a fundamental economic problem underlying software insecurity [11], and in general there often appears to be an unwillingness in industry to give code security equal importance as other business considerations, such as time to market and richness of features. It is therefore important to recognize that the (in)security of software produced by software companies is impacted not only by individual developers' knowledge and skills and the types of programming languages/environment they use, but also by the various incentives at play both in the market and at the organizational level. Thus, to produce real impact in secure software development, it is indispensable to study this problem in the context of where the process happens, i.e., in the software companies.

Recent work by Sundaramurthy et al. [12, 13] showed that by employing the anthropological research method of participant observation [14, 15], researchers successfully obtained deep insights into the challenges faced by security analysts in security operations centers (SOCs). Moreover, embeddings in the SOCs allowed researchers to produce both technical and non-technical interventions that improved SOC operations by uncovering and addressing the pain points in the overall work process and environment. Encouraged by the success in that work, we conducted an extensive ethnographic study in a software company, using the same method of participant observa-

* Both are first authors. Names are ordered alphabetically.

USENIX Symposium on Usable Privacy and Security (SOUPS) 2020.
August 9–11, 2020, Virtual Conference.

tion. Two PhD students in computer science were trained in qualitative methods by an anthropologist and spent 1.5 years of total research time doing fieldwork in the company. They participated in everyday work activities in the company such as coding and meetings, and observed software (in)security phenomena both through investigating historical data (code repositories and ticketing system records), and through pentesting the developed software and observing developers' and management's reactions to the discovered vulnerabilities. The fieldworkers then shared their observations with a larger research team that includes the anthropologist and computer science professors collaborating on this research.

In this paper we report our findings on specific examples where incentive structures, organizational relationships, work flow, and other contextual factors shape how security considerations come into play (or not) during the development and updating of software. Some findings were not what we would have expected before the research and shed light on some deep relationships between secure software development and the various incentive structures inherent in the software industry. We present them in the hope of eliciting a broader look into the secure software development process, taking into account at times forgotten human factors that significantly impact the security of software developed in companies.

Finally, through the fieldwork, the embedded researchers were able to make interventions into the firm's software development process, providing methods and tools to help fix discovered vulnerabilities and prevent similar mistakes from occurring in the future. In doing so, we found that the fact that the fieldworkers were part of the software development team became a major reason that the methods and tools they provided actually worked within the context of the company and were taken up by the developers. This co-creation approach provides a model for how security experts may increase their ability to improve the security of software.

2 Research Methods

The main method utilized in this research was participant observation [14, 15]. This method was developed by anthropologists and sociologists as an effective way to study human behaviors and cultures through participating in daily activities and observing people's behaviors through long-term study (typically more than a year). These activities help researchers obtain a solid understanding of a particular culture and gain insights into subjects' activities, knowledge, and habits. By adapting this approach to work within a software company, we can provide an in-depth examination of the complexity of the software development process, the various incentive structures among the stakeholders impacting human behaviors, and the tight coupling of both technical and human factors that impact software security.

In this research, the participant observers were two computer science PhD students, each of whom underwent systematic training in qualitative research method under the guidance of the anthropologist (Lende) on our research team. Being CS students and possessing a substantial amount of security knowledge enabled them to get quickly immersed into the company's software development process and start observing practices that might have an impact on the software products' security. Being inside the company enabled them to observe both contemporary events as they unfolded, as well as past events studied through ticketing systems and checking the relevant code in the repositories. The students' role in the company – working as if they were an employee of the company – helped with two important assets of our research. First, their daily interactions with the developers while doing regular on-the-job tasks provided a unique angle to observe the subjects' authentic behaviors as they performed their job duties. Second, they not only acted as passive observers but as advocates of software security inside the company. This approach enabled the team to observe how the various stakeholders reacted to discoveries of security vulnerabilities, providing valuable insights into why those vulnerabilities were introduced in the first place and the constraints under which they could be fixed (or not).

Each researcher worked at the company 20 hours a week, spread across three week-days. One researcher worked for 12 months and the other for 6 months. The researchers were not paid directly by the company. However, the company provided both financial and in-kind contributions to this research. In general, the researchers' tasks included debugging existing implementations to find bugs' root causes, writing code fixes or implementing new features, performing code reviews, and software quality assurance. The researchers took field notes about their observations, including both security issues found in the software and everyday interactions with developers and other employees involved in the development process. Notes had two forms: descriptive and insightful. Descriptive notes were intended to be as informative as possible, avoiding personal judgments or opinions. Insightful notes aimed to capture "ah-ha" moments and provide reflective analysis of the situations experienced by the observers.

To derive research insights from the raw notes, we applied the general inductive approach [16], augmented by specific techniques for qualitative data analysis [17]. The initial step was to find patterns that emerged directly from the data themselves. In our research, this process happened via weekly meetings of the larger research team including both the fieldworker(s) and the professors, where comparisons could be made across researchers, discussions could address both the human and technical dimensions of software development in a company, and plans made for further exploration of interesting topics. Identifying themes and links between ideas proved central to the inductive analysis, as well as developing contextual analysis around key examples. Data analysis continued through the coding of field notes based on identified themes. These codes included themes related to software

security, human elements of the work, important explanatory concepts that emerged during the research, and data linked to the key examples. A more detailed description of the coding process as well as the codebook can be found in the Appendix. Research meetings then shifted to further developing our joint understanding of the data and identifying ways to explain the observed patterns, as well as potential solutions to how human and technical factors combined to shape (in)security.

It is important to highlight two unique aspects of our participant observation approach. First, participant observation is often a solo affair in the social sciences; having two embedded researchers permitted the examination of the company from two different but complementary perspectives. The researchers were assigned different tasks, had slightly different hours at the company, and developed relationships with company personnel at different points of time. This dual approach to participant observation increases the robustness and validity of the data from this research. Second, the research team consisted of experts in engineering and social science. This multidisciplinary team participated with the embedded researchers in developing the analysis over months, permitting the identification of themes and ideas that crosscut disciplines and had both theoretical and applied dimensions. This team-based approach to both data collection and analysis is a significant contribution to how this type of research can be done effectively.

3 Context

3.1 The Company and Its Products

At the company, the researchers worked in the same space as four other developers, four support engineers, two network engineers, one customer-facing onboarding specialist, the CTO, a marketing and sales manager, and other staff. The researchers' work focused on two products: a solution for controlling network access and a solution for allowing users to securely access networks remotely. The solutions configured third-party network devices (e.g., routers and access-points), enforced operator-defined access-control policies, and managed remediation flows. Typical customers were medium- and large-size organizations, and common users were IT staff who managed the organizations' networks. Organization end users attempting to connect to its network were prompted first by a captive portal that asked for credentials. Once authenticated, they were asked to remediate any issues that prevented them from complying with policy, e.g., they might be required to download and run a client-side monitoring agent and update their anti-virus software.

3.2 Development Process

The company followed general agile development principles. The development team held a scrum meeting every morning that lasted 15-30 minutes. In this meeting, each developer briefly commented about any progress accomplished or roadblocks encountered the day before and discussed the plan-of-work for the current day. This was an opportunity for developers and managers to give and receive feedback from each other. The meeting was led by the dev team lead. The CTO was usually in the room but did not lead the meeting.

Work was organized, prioritized, assigned, and tracked using ticketing and code management systems. In general, tickets were generated by developers, support techs, or customer-facing specialists, ranked in prioritization meetings held by the dev team lead and CTO, and assigned and tracked by the dev team lead. After implementation, tasks were moved into the peer-review stage in which other developers (often more experienced ones) reviewed any code changes, added pending tasks if necessary, and finally approved merge requests. After code changes were approved by all reviewers, tickets were reassigned for quality assurance and integration testing, which was often done by both developers and support/customer-facing specialists. When all tests had been passed, tickets were marked as "done" and merged into the code repository's development branch. When the set of target features for a release had been implemented, the team lead created a release candidate branch. Every release candidate was tested in-house one last time before being finally moved into release and installed on customer environments.

3.3 Study Participants

The main participants in the study were the four software engineers on the development team where the student researchers were embedded. The dev team lead was an experienced developer who had been at the company long-term and written many parts of the system. Two of the other developers had been with the company for several years and another had recently joined. One developer specialized in front-end development and two were full-stack developers. The researchers also interacted with other personnel at the company, including the CTO, via company meetings, work communications, and everyday activities such as breaks and lunches where people often "talked shop" in informal ways.

3.4 Research Ethics

In our research, the employees of the company (developers, support techs, and managers) were considered human subjects. The study was reviewed and approved by the Institutional Review Board (IRB). Researchers explained the study goals to participants and obtained verbal informed consent from participants. Field notes were anonymized, as well as discussions during weekly research meetings. This paper follows that same anonymization approach. Throughout the paper, we use the term application under study (AUS) to refer to a

specific application in the company's product suite. We also anonymized all product-specific terms in the paper.

One ethical dilemma that emerged during the research was what to do when security vulnerabilities were discovered. Given ethical standards among cybersecurity professionals, we made the decision to present these discoveries to the software development team. This process proved crucial to the further development of the research. Rather than simply observing what happened while continuing to work at the company, the researchers raised these security concerns, and where directed, actively worked on addressing them. This active engagement led the research team to a co-creation model, where research, programming, and security were all ongoing parts of what happened during the fieldwork.

4 A Historical Study of a Security Flaw/Feature: Silently Allow Failed Authentication

In this section we describe a security flaw (or feature) that was encountered by the first researcher soon after he started working in the company. Studying the origin and evolving usage of this feature provided a lens through which we observed a number of dilemmas the developers and company had to deal with. This could be helpful for understanding the root causes of other similar security flaws. We first describe this feature and the methods used to study it, then analyze how it was used in multiple instances, and finally draw some conclusions through our reflective analysis.

In this AUS, the process of authenticating users and devices into the network involved assigning an authentication state to every authentication attempt and subsequent authentication queries. Authentication queries were self-triggered by the AUS and configurable, i.e., AUS operators specified how frequently users and devices must authenticate and the AUS executed the corresponding assessments by querying authentication servers, the operating system on the client's device, or sometimes prompting the user directly through a web browser. The authentication states were: pass, fail, unreachable server, unknown username, and the silently-allow (SA) state. Once an authentication flow entered the SA state, it acquired an SA role, which was associated with a built-in policy that granted full access to the protected network. From a security perspective, SA was a dangerous authentication bypass mode, providing access without full authentication (hence, silently-allow). Specifically, an authentication attempt assigned to an SA state was treated as successful and granted full access to the network the AUS was expected to protect. Further, the SA state was relatively persistent. Once a device acquired an SA role, it remained in this state until forced to re-authenticate (by default every 4 hours), or when an administrator manually added policies that would forcefully change the role assigned to the device.

4.1 Specific Methods Adopted in This Study

In addition to the participant observation research methods discussed before, the researcher also used the company's ticketing system and code repositories to understand the rationale behind the SA feature, particularly why it was introduced in the first place and why it kept being used later. In the ticketing system, the researcher located all records where comments were made about the SA feature. Developers rarely used security-related terms in the tickets' description, so uncovering the different instances of SA required the researcher to correlate ticket information with the implementation code. After an initial SA instance was discovered, the researcher searched the code repositories for potential SA-related terms and variable names, formalized the authentication flows, and set up a lab for demonstrating that the SA state could be triggered at runtime. These led to the discovery of other SA instances not mentioned in tickets.

4.2 Observed SA Instances

The AUS had five configurable authentication back-ends: three databases (one legacy implementation, a second one for network guests, and a third one for admin user accounts) and two integration components (one connecting to LDAP servers, and a second connecting to SAML identity providers). The SA state could be triggered in each of these configurations, under one of the following scenarios: 1) the AUS failed to communicate with some authentication server, 2) the AUS was misconfigured, 3) an unexpected runtime exception occured, or 4) a back-end implementation was lacking.

4.2.1 Broken Integration

Broken integration with customer's backend authentication server was the main trigger of the SA state. In the LDAP authentication flow, the AUS communicated with Active Directory (AD) servers. The connection parameters must be pre-configured by network operators and stored in the AUS's internal database. At runtime, these parameters were pulled from the database and LDAP search queries were executed. If the network connection to the AD server failed (e.g., due to misconfiguration, a physical link failure, DNS down, or timeouts due to request overloads), no domain entries were found in the AD server, or any other runtime exception occured (e.g., invalid credentials or duplicate entries in the AUS database), then the device was assigned the SA role and obtained full access to the network.

Most developers and support techs were aware of this SA scenario and often framed it as a feature that alleviated the configuration burden for network operators. It was often justified by explaining that network operators were more interested in not blocking legitimate users into the network than in protecting their networks from intruders with stricter policies that could affect network usability.

For example, support techs said that there had been scenarios in which customers became very frustrated because policy changes had forced most users out of the network and required them to reauthenticate. According to interviewed participants, network operators would see these scenarios as "network outages" that would cause a high load of help-desk support calls for them, which were not well-received.

It remains unclear how many customers actually preferred this SA solution to one that would have required the customers to fix their own integration issues. A support ticket generated from a customer call four years before the study showed that at least one customer would not prefer this SA solution:

> *"Customer doesn't want failed authentication attempts due to LDAP errors to fall into SA but to try another server."*

There was scarce evidence on customer-facing documentation to support the claim that all customers were well aware of SA authentication. SA was only mentioned as a footnote in the release notes PDF of a version of the product that had been released five years before the study, which read:

> *"A system failure will not cause a network-wide outage and will silently-allow authentication for existing and new users attempting network access."*

When the researcher raised the concern that some customers may not be fully aware of SA, some participants explained that customers would be informed only if they asked about the SA role (which was only visible when triggered, on a secondary page on the UI). Since this SA instance was not seen as a security vulnerability but as a feature, no action was taken to remediate the issue.

4.2.2 Misconfiguration

The second way in which the SA state would be triggered was when the AUS was misconfigured. Specifically, a drop-down menu in the UI for policy creation in the administrative portal allowed operators to select an SQL authentication option that mapped to a non-functional legacy database. This would result in assigning an SA state to all authentication attempts, regardless of which credentials were entered by users. Developers explained:

> *"This authentication method is probably broken. I believe it has been deprecated a long time ago."*

When asked why the UI was still showing this option, they said, *"I don't know why but it should not be there."*

We were unable to find any written documentation other than the code to confirm if and when this authentication method had been officially deprecated, or if any customers were still using it. An internal testing ticket from three years back suggested that this authentication method had already been deprecated, but some customer-facing documentation still listed SQL authentication as a possible authentication method. The issue was documented on a ticket by the researcher but at the time of this writing was not yet prioritized for development.

4.2.3 Unexpected Runtime Errors Due to Implementation Bugs

The SA state could also be triggered by unexpected runtime errors. Several instances of authentication code were surrounded in try-catch blocks that would catch SQL and other runtime exceptions and set authentication state directly to SA. Exceptions were somewhat common across the AUS and sometimes caused it to halt operation. Some SQL exceptions occurred after upgrades that resulted in tables with missing attributes, or because the AUS had incorrect database permissions. Other runtime exceptions included null pointer exceptions and out-of-bound array access.

Like the broken-integration case, this instance of SA was an intentional choice. The code that implemented SQL server authentication was added more than 15 years ago and since then had been revisited a few times. Although not explicitly stated, it is possible that this SA instance was an ad-hoc solution for dealing with code complexity and legacy implementations, allowing the AUS to continue execution despite any incomprehensible bugs. The code preceeding the catch blocks looked complicated (with several sections commented out). From reading the code, it was hard to tell what code paths could be executed in each scenario.

4.2.4 Unimplemented Protocol Flows

Perhaps the most critical SA instance was an unimplemented SAML[1] authentication flow that allowed users directly into the network without even checking the credentials against the SAML identity provider. Unlike previous SA instances which were somewhat acknowledged by developers, developers said that they were unaware of this SA instance and believed previous developers who no longer worked at the company were responsible for this problem. When asked about the impact of the problem, some developers said that

> *"Very few customers are probably using SAML authentication on their networks."*

Yet, in a separate conversation, a support tech said that he knew of at least ten customers who were using SAML for authentication.

The SAML SA instance was fixed by the researcher by implementing the missing SAML authentication flow.

[1] Security Assertion Markup Language, an XML-based standard language for communicating security assertions, often used in single sign-on protocols.

4.3 Reactions from Developers

According to members of the support team, SA was an "effective" solution to reduce customers' frustration for the number of help-desk calls that customers received when end-users experienced network interruptions. SA addressed customers' requirements for a product that would deliver an easy-to-use and frustration-free experience for both network operators and end-users. However, we found that in general, developers were somewhat hesitant to talk about SA, either because they did not understand how that part of the software worked or it was code they were not proud of.

Attempts to explain the security implications of SA to the developers and get the problem fixed was not a straightforward process. A common perception is that security vulnerabilities are introduced into code because developers are not aware of the security issues involved, and explicit exposure to the issues would allow them to understand and take immediate corrective actions to remediate their software. This was not our experience in the case of SA. First, for three out of the four SA instances described above, developers were aware of them. And for all the SA instances, they acknowledged that it was problematic. However, even after we brought these issues to their attention, the developers still did not implement the expected security fixes. Moreover, reactions after being exposed to the insecure code were not always consistent with subsequent behaviors, which suggested that there were other reasons why the SA problem persisted in the code. Some examples are explained below.

- Blaming misconfiguration and broken integration issues on customers. Developers complained that customers' limited understanding of their product and networking was the main reason why they weren't able to configure the product within their networks correctly. However, sometimes even the most senior developers spent days to get the product and network configured right.

- Writing limited documentation about authentication flows, which made them difficult to understand for anyone other than the code's original authors. While initial explanations were that there was not enough time to write documentation, months later some participants admitted that some areas were not documented because they believed their implementation could be wrong.

- Not practicing what they preached with respect to testing practices. Everyone in the development team said things like *"we should all do more testing"* and some asked the researchers to write very detailed test cases. But often they did not hold themselves to the same standards and wrote very few tests for the code they wrote. There were also many trivial tests in the codebase from the time when they were using code coverage tools. Some developers admitted that at times they would write trivial tests and minimize code changes just to get the code coverage numbers required for a release.

- Blaming previous developers who were no longer working for the company. Developers often talked about what previous developers did was wrong, but it was not clear whether anyone attempted to correct the problematic code prior to being released.

In summary, the attempts of the researcher's intervention in the SA case were mostly unsuccessful. The developers seemed to be aware that this security flaw could be a major issue for the software and company in the future. However, of all the SA instances identified by the researcher, only the SAML flow was fixed (by the researcher). For the other instances, developers did not see great value in fixing them. They also thought that if they had to fix all security bugs the company would go out of business.

4.4 Lessons Learned

Incomplete stories told by developers, attitude-behavior inconsistencies, and poor documentation were good reasons to believe that the introduction of the different instances of SA deserved deeper analysis. In addition to the evidence directly related to SA, the embedded researchers collected information about interactions among participants during design meetings, prioritization meetings, and informal discussions. Because the company's organizational structure was organic, they were able to interact not only with developers but also with support techs, network engineers, and managers. These interactions provided valuable information for analyzing the deeper motives behind the introduction of the different SA instances. Moreover, the research team identified relationships that connected observed behaviors of the participants with the evidence found on the ticketing systems, the code, the internal wiki, and release notes. The main insights learned from the analysis are described next.

1. Vulnerabilities are sometimes introduced to make errors unnoticeable in an attempt to reduce the number of customer support tickets.

One of the effects of SA was that it would make runtime errors unnoticeable by customers. This reduced the chances that customers would complain that the AUS was not working correctly. When developers talked about these complaints, they implied that customers would blame the company if the AUS could not communicate with other servers, even if the problem was extraneous to the AUS. As one developer stated:

> *"If the system breaks because we followed the specification and the system cannot talk to another server because they are not following the spec, we are probably going to lose money. So we need to code to prevent that."*

Moreover, preventing those errors from bothering customers would imply that less integration-support tickets would be generated, reducing the chances that some of those tickets would be escalated to the development team. Because developers often complained about how much time they spent debugging issues reported by customers (documented on integration-support tickets), it is likely that developers introduced SA in part to alleviate their job stress.

Integration-support tickets were created by support techs assisting customers in integrating AUS with their other network products (e.g., routers and switches). Whenever support techs were unable to resolve tickets in this category, they would escalate them to the development team for further investigation and potential development of bug fixes or custom integration code. Although developers understood that assisting support techs was necessary, they appeared to be more interested in developing new features than fixing bugs or writing custom integration code (which would later require more effort to be maintained).

Further, debugging the issues described on the tickets was challenging because it often required setting up environments that were similar to their customers', which was difficult because of the diversity of network device vendors. In this context, SA reduced the number of integration-support tickets, so developers would spend less time debugging integration problems, and have more time to develop new features.

2. Managers and developers prioritize tasks by doing a heuristic cost-benefit analysis, ranking tasks by urgency and effort required, and security improvements rank low and are usually not fixed because they are not considered urgent or easy to implement in practice.

In general, tasks were prioritized based on heuristic urgency and effort estimations for cost-benefit analysis. Task urgency was often estimated by development managers who were also part of sales and support-prioritization meetings. Urgency estimations tried to measure how much positive or negative impact some new feature or bug fix would have on the business. For example, the development of a new feature could help win or lose a new deal, or a bug fix could help retain or let go of an existing customer. Task effort estimations were often briefly discussed at scrum meetings and more in-depth during "scrum poker" and prioritization meetings. In scrum poker meetings developers estimated the effort required to resolve certain development tasks by assigning them numbers in a scale from 1 to 100. Every week, task priorities were re-evaluated and reassigned or put on hold if necessary.

Of the four instances of SA, at the time of this writing our field researcher was only assigned to fix one, the unimplemented SAML flow, which was considered to be more urgent. Managers and developers believed that fixing the other SA instances would have no positive impact on the business, so they considered them not urgent, low priority, and thus did not address them.

3. Some security vulnerabilities were introduced by leaving deprecated features in production code, and this could avoid breaking existing implementations.

Another possible reason why SA was still in production code was that developers wanted to avoid introducing potential issues that could be caused by fixing the problem. For instance, when asked why misconfigurations SA (Section 4.2.2) was still there, developers' answer was that nobody took the time to remove it. In fact, removing the SA implied a risk, i.e., some other part of the software could break on production systems. Thus developers likely would rather not to take this risk, especially because there was no need for it (customers were not demanding it). In summary, there were just not enough incentives to remove it.

4. Some security vulnerabilities are not detected during development partly because testing is not always embraced by developers.

Any of the SA instances could have been detected if the appropriate tests were executed. However, because developers didn't necessarily like testing and could just write tests to pass the minimum testing requirements, the tests were ineffective and thus would not detect the SA vulnerabilities.

5 Live Discovery through Pen-testing during Ethnography

This section describes how the research combined vulnerability discovery with participant observation of developers' behaviors and reactions. This technical-ethnographic combination is similar to the method used in studying the SA issues, but is also unique. Unlike in the SA case, none of the pen-testing discovered vulnerabilities were intentionally introduced by developers, and as such the researchers had the opportunity to observe the unfolding of developers' reactions with the discovery of a totally unknown problem. It also provided the opportunity for researchers to intervene in a way that resulted in a co-creation model, in which security experts work jointly with developers to improve code security.

5.1 Specific Methods Adopted in This Study

Penetration testing, also known as pen-testing or ethical hacking, is an authorized simulated cyber-attack process against a computer system to reveal security flaws. The goal is to identify weaknesses which might provide a passage for unauthorized users to gain access and alter the integrity of the computer system. There are multiple software pen-testing methodologies, including the Open Web Application Security Project (OWASP) [18], Open Source Security Testing Methodology Manual (OSSTMM) [19], NIST SP 800-115 [20], Penetration Testing Execution Standard (PTES) [21], and Information System Security Assessment Framework (ISSAF) [22]. Results vary based on the way the process is performed.

Two of the company's products were picked by the researcher for further study. By acquiring some information and insights about the applications, the researcher started to apply a customized penetration testing methodology. The basic information such as the software workflow, authentication information, and so on was captured by talking with developers and the support team. Both products were designed to work on a web platform, so OWASP's top ten security vulnerabilities [18] were chosen for the testing.

At the same time as developing the pen-testing process, the researcher worked to gain the developers' trust. This process required building rapport, an understanding with research informants, by participating in daily tasks and getting to know individuals who worked there. The role of a security pentester also needed accurate planning and time management. Code injection was selected for the first vulnerability to be tested. It is one of the well-known vulnerabilities which allows attackers to inject malicious codes into a computer system and change the course of execution. The result of a successful injection can potentially be catastrophic. Before we describe the pen-testing findings and developers' reactions, we first briefly introduce the three types of vulnerabilities found.

5.2 Vulnerabilities Found

- Cross-site Scripting (XSS): generally found on web platforms. Attackers typically use web applications to inject malicious codes into the application which can be viewed by other users.
- HTML Injection: similar to XSS. However, instead of inserting malicious scripts, the attacker can inject valid HTML tags and modify the content of the target website.
- Shellcode Injection: a type of vulnerability that allows an attacker to inject malicious code into a system and provide the attacker a shell on the system.

5.3 Behaviors and Reactions from Developers

During the first day of pen-testing, an XSS was found in the AUS by the researcher. The vulnerability was brought up to the developer team, and a proof of concept was provided for why it was significant. While they showed interest in the finding, since the vulnerability was in a 3rd-party application integrated with the AUS, their first reaction was to hope the problem had been fixed by the 3rd-party. One participant said:

> *"This vulnerability belongs to our 3rd-party application, and we did not develop this part. It is better to upgrade the software and see if we will still have the issue"*

They also mentioned that it would be more interesting if the researcher could find any vulnerability inside part of the company's code. Thus, they expressed interest in security, but saw solving this problem as the responsibility of an outside group even though the application formed part of the company's software.

In the next round of testing, the researchers tried other parts of the software to see if there were any other vulnerabilities. Multiple XSS vulnerabilities were found. Developers were both excited and concerned about the findings. They said things like,

> *"If they want to test more, it seems that they will find more things inside our software"*

and

> *"We tried to minimize our bugs, but it seems something is wrong."*

Once again,the third-party issue came up:

> *"We are using Angular, and I thought we shouldn't have the XSS. Angular should take care of this issue."*

On the same day, the researcher found another vulnerability in the AUS; this time it was shellcode injection. The vulnerability allowed attackers to inject their customized shellcode into a valid file and upload it into the server and get backdoor access with a powerful user's account on the server. The attacker must be someone who already had a regular account inside the AUS web platform. The finding was interesting to the researchers and the developers for different reasons. For the research team, this was a critical vulnerability – customers should never have escalated access to the server. They should only be able to perform some limited commands on the OS such as changing the network IP address. This essentially allows their customers to jail-break out of the sandbox set up on the server. Developers on the other hand were more interested in understanding how access had been gained. They said things like,

> *"Interesting! Could you show us how you got the access?"*

and

> *"What is your user's privilege?"*

At the same time, they discussed the risk in the context of the product,

> *"because we already ship the OS to the clients with everything inside it, it's kind of okay! They have the box already. ...We do not have any important information on it."*

During the subsequent discussion with the developers, one researcher asked, *"Do you have any hard-coded password or credentials?"* The answer was *"yes."* This hard-coded

credential would allow a customer who successfully exploited the shellcode injection vulnerability to see other customers' information. Faced with this fact, the developers indicated that this vulnerability was bigger than they originally thought, and that they should take imminent action on it. However, that did not come to fruition at the next group meeting, where they continued to talk about these vulnerabilities. Based on this discussion, the research team inferred that there were other factors that affected the developers' actions. In fact they said *"fixing the vulnerability has additional impacts and may cause some problems for other parts of the application or customers."* They also downplayed the significance of the shellcode injection vulnerability, and said that the team should focus on developing new features. They added,

> *"If we want to fix every bug in our system, we will be out of business very soon."*

Contrary to the researchers' initial hope, our intervention effort to fix the discovered security vulnerabilities proved ineffective in the context of the company's overall functioning. This moment helped the researchers realize that they needed to rethink how security researchers engage developers to create positive change.

It started by simply offering to work on the issue and building the tools and libraries for them to prevent XSS. The bug was then fixed by one of the researchers. One challenge the researcher faced was that the AUS was uniquely designed and would only accept specific types of input entry for the various fields. As a result, it was not possible for the researcher to utilize standard input sanitization solutions, e.g., one that removes all special characters, because that would break the application. Working at the company and interacting with the developers helped the researcher understand this uniqueness and come up with a customized solution. It was a number of specially designed regular expressions that enforce the proper formats for the various types of fields. The researcher included the application of these regular expressions in a standalone Javascript file that can be invoked at the front-end pages. It turned out that the company's existing code already contained a similar mechanism for checking other properties of front-end input fields, e.g., if a field is empty. The researcher only needed to extend this mechanism to include the regular expression checks he designed for preventing XSS. Developers could then simply invoke these checks in the same manner they had been doing for the other types of checks. This allowed for easy integration of the security check into existing code with minimum change, and was readily accepted by the development team. For back-end input sanitization, the researcher first tried to apply standard OWASP sanitization functions for Java, which was the language the back-end was written in. However, due to the uniqueness of the formatting requirement of the AUS, those standard checks were blocking some legitimate inputs. Thus the researcher needed to customize those OWASP functions to work properly with the AUS's requirements. During the fixing process, another XSS was found in the AUS. When the researcher brought up the problem to the development team this time, they accepted it very fast. A new ticket was created and the researcher was asked to fix the issue as soon as possible: *"...Go ahead and fix this bug as well."*

This example highlights the importance of "being there" for security experts to drive positive change for secure coding. The researcher was able to accomplish this in this case due to two factors: 1) he understood the company's existing code and designed an effective security check that minimized disruption; 2) he provided the needed security expertise in designing the proper checks using regular expressions and the customization of the OWASP functions, and this expertise was delivered through code artifacts that were readily applicable within the existing software workflow. Both factors were important for this success.

The researcher tried later to bring up the shellcode injection vulnerability once more in a discussion and tried to convince them to start fixing the issue, but the suggestion was turned down. One of the developers responded,

> *"It's somewhere in our backlog. We didn't do anything about it, and no one has found that exploit so far. So we are safe."*

This comment matched similar instances where developers reacted as though if there were no problems at present, the vulnerabilities might not be an issue that needed urgent attention. The research team considered a possible explanation why the shellcode injection vulnerability was not treated as urgently as the XSS. Exploiting the shellcode injection vulnerability would require a rogue player that can be held accountable (a customer's IT staff member who possessed the regular account access to the server). This may have alleviated the concern on the company's liability resulting from this vulnerability.

Later on in the research, an HTML injection was found inside a newly developed part of the code. Like the XSS, the issue was brought up to the developers. Initially, developers mentioned,

> *"Angular should cover it and not allow the HTML tag in the code! It seems it does not."*

During the next group meeting, they recognized that they had omitted security issues previously. They said,

> *"When we discussed the development of the page, we talked about everything except security and XSS problems. They didn't come to our mind."*

At this time, the researcher thought that the developers would ask him to fix this issue like in the XSS case, but they started to fix it by themselves and did not ask the researcher for any help. Most interestingly, the developers created correct

solutions to fix the HTML injection vulnerabilities, based on the way the researcher solved the XSS problem. This showed that the developers learned from the researcher how to create security fixes within their code base, without being explicitly taught so. They learned by simply observing the code artifacts created by our researcher.

This is an example that illustrated the importance for security experts to be in the development environment and "co-create" security solutions with the developers. The difference in the developers' reactions in this case, compared to earlier ones, pushed the researchers to realize that co-creation happens more in the moment, rather than trying to retroactively fix things. Our earlier interventions mainly focused on fixing vulnerabilities found in code written in the past. We had success in getting some fixed (by the researchers). Whereas in this case, the developers took their own actions and fixed the bugs using the knowledge and tools provided to them by the researchers. This shows that if security professionals are present and part of the team when a product is in the process of being designed and implemented, their views are more likely to be taken into account when decisions about what to do are being made. It was also the researcher's feeling that the quickness with which the development team accepted his suggestion to fix this issue was related to the increased level of trust he enjoyed from the development team at this point in the research progress.

5.4 Analyzing the Findings

After initially finding the first bug in the pen-testing process, the researcher assumed that developers did not know about these security problems, and lack of security knowledge led them to write code with the vulnerabilities. After working with them on various tasks, he realized that they actually possessed quite a bit of security knowledge. As our research progressed, group discussions and analysis of field notes highlighted some non-intuitive reasons for developers' behaviors. This indicates that there were other significant factors in causing these vulnerabilities. We outline these factors in the rest of this subsection.

5.4.1 Developers Should Not Totally Trust Programming Languages and Frameworks

One of the important conversations that the research team had with the developers was that the developers believed that the programming language/framework should take care of some vulnerabilities by default.

"...Angular should take care of this vulnerability..."

In this case and according to Angular documents [23], the Angular engine could handle most of the XSS and HTML injection attack scenarios by sanitizing the input fields. Angular documents also mentioned that developers need to take care of backend servers to make sure injection vulnerabilities are not introduced there. After analyzing our field notes carefully, we found that the developers believed (incorrectly) that Angular could handle all XSS and HTML injection vulnerabilities.

In the past decade, programming languages and frameworks have been doing a great job in creating built-in security measures to prevent accidental mistakes by developers, but they still do not offer a comprehensive security solution. How can developers know accurately where they can rely on language/framework and where they must rely on their own code to achieve a security property? Can this be communicated in a way that does not require sophisticated knowledge on all possible ways attack could happen?

5.4.2 Outsider vs. Insider, and from Deficit Model to Co-Creation Model

One of the embedded researchers in the past worked as a pen-tester for four security consulting companies in three countries for four years. In his experience, the pen-testers were not incorporated into the development team. The developers might only receive a document with discovered vulnerabilities and statements about what they should or should not do. This appeared to be a common industry approach to software security pen-testing [24]. The problem was that the security pen-testers did not understand how much workload the developers had, nor the actual reasons for the vulnerabilities. As a result, this approach did not often lead to the desired changes in the development process, but set up an outsider/insider dynamic, where developers felt the need to defend what they had done and/or minimize the security issues. The developers would say that the report came from an outside group who did not really understand how software development was done, and the security pen-testers would say that the developers wrote defective code in the first place and did not appreciate security, otherwise they would have done something to fix all those problems. Having these past experiences, in contrast with what he experienced in this research where he worked *inside* the development team as a software pen-tester, helped the research team to understand the impact the outsider/insider dynamic had on effectuating changes in software development processes.

From our fieldwork experience, we clearly see how this outsider/insider dynamic can play out. When we first found the vulnerabilities about SA, XSS, or code injections, the researchers' initial thought was that the developers did not know about these security issues. However, after researchers explained to developers and developers had clearly understood the technical details, still some vulnerabilities were not fixed. It was only after further communication with the developers, reflecting on other relevant observations made by the researchers, and brain-storming among the larger research team, that we better recognized why some vulnerabilities were not prioritized to be fixed. Most importantly, *it is when we had*

this understanding, and produced an easy-to-apply solution that fit into the company's development workflow, that our intervention was the most successful.

The point of view that if developers know better and work harder, they should be able to write software without any security flaws, can be characterized as the so-called "deficit model," where the problem of software insecurity is attributed to the developers' lack of knowledge or efforts. The solution driven by this deficit model would mainly involve experts explaining to developers the various software security issues and how to prevent them, and hoping this would drive the needed changes. Research in fields such as education, anthropology, and science communication have examined how using such a deficit model does not prove as useful as imagined because it localizes the problem inside the person and assumes that simply fixing that internal lack will also successfully address larger concerns such as successful learning, cross-cultural understanding, and the application of science to at times controversial topics [25–27]. One analytic concept of note that emerged through the research was our own use of a "deficit model" to initially interpret why people in the company did not respond to security concerns. We assumed that they might not have the knowledge or awareness to understand security risks and recognize how and why particular aspects of the software might increase those risks. Our research found that this deficit model-driven approach was not working well. Simply communicating security issues found and presenting solutions for fixing them did not lead to the anticipated fixes.

Overcoming this "deficit model" in our own thinking helped us to better interpret why participants responded or not to security issues and to recognize how security concerns existed alongside other factors that shaped their work. We then developed a co-creation model, where developers and security experts collaborate together. Co-creation is a form of collaboration in which ideas and processes are shared and improved together rather than kept to only one-party side. By having a co-creation model, security auditors have the chance to jump into the development process and provide the knowledge and tools that developers can readily apply to prevent vulnerabilities. Part of this co-creation model meant that our embedded researchers did not work exclusively on security but dealt with different tickets. This showed the developers that the researchers knew how to program, and could do so as part of a team, while also having expertise in security that they could draw on if needed.

It appeared to us that developers prefer to trust a person inside their team rather than an outsider. Moreover, our field notes showed that a security person inside the developer team can provide more in-depth knowledge than outside resources such as pen-testing reports, internet, and so on. For example, after the XSS got fixed on the AUS, when developers faced the HTML injection they said: *"Is this HTML Injection going to be easy to fix? It should be very easy to fix,"* and without asking the researchers to provide the solution for them, they fixed the issue.

5.4.3 Thinking as an Attacker, Thinking as a Developer

It has almost become a platitude in the security field that one must "think as an attacker." Applying this to software development, the developers can put themselves in the attackers' shoes and understand how software may be misused. It is an interesting question as to *how much* developers need to think as an attacker. These days, understanding the mechanisms of all types of cyber attacks can be overwhelming even to a security expert. Our data implied that the developers and the company were aware of some of those threats that they may face, but just knowing them was not enough. The problem is not necessarily about the lack of understanding the attackers; it is more about not being able to implement security features correctly into software, which unfortunately requires some non-trivial amount of security knowledge. Is it realistic to expect all software developers to become security experts? How much time should developers spend on thinking about how their code may be attacked, among all the other competing demands they face? Security professionals can help bridge this gap by starting to think like developers, just like how we ask developers to think like attackers. Security professionals need to better understand how developers have to negotiate many competing interests, not just a sole focus on security. This could help in providing security knowledge and information at the right level of abstraction that can be easily integrated into the software development process. The co-creation model we used as part of this research allowed the security researchers to think like a developer, and to create some positive impacts in the software development process.

6 Discussion

Our ethnographic study found that software security or lack thereof emerges from a network of technologies and humans, rather than happening solely because of deficits in the software and/or in developers' knowledge or efforts. The people in this company were trained professionals acting in good faith to create successful products. They faced dilemmas that can be common in the software industry and aimed to resolve them in ways that produced a viable product, established good relationships with customers, maintained profitability, and enhanced usability and security. Security issues were not directly attributable to deficits in knowledge, but rather both emerged and could be resolved in terms of the dynamics that shaped how developers dealt with both technical and human demands and balanced their primary aim to develop successful features for the software with their understanding that security is an important part of software integrity and functioning. The co-creation model emerged by attending to these dynamics in the workplace, and responding to what worked and did not work

to address specific vulnerabilities during the daily work of development. This co-creation model could form an integral part of a workable solution to improve software security. In fact, some success in applying secure development life cycle (SDL) in industry echoes our findings [28]. Whether the co-creation model could work under financial realities, and what other ingredients need to be there to form a working solution, remains an intriguing question for future research.

7 Related Work

Assal and Chiasson [1, 2] utilized interviews and surveys to explore the interplay between developers and software security processes. Their research found that developers were motivated to develop secure code, but were often hindered by a mismanaged organizational process. The authors advocated looking beyond developers and examining broader organizational factors that may impact the security of the developed software. Our work is one such attempt, utilizing an extensive ethnographic study in a software company. Many of our findings confirmed the analysis results from Assal and Chiasson's work. Our work also revealed some deeper insights into the reason of software (in)security, as well as a co-creation model that can help address them.

Ruef et al. [3] and Votipka, et al. [4] conducted a series of studies based on data collected from the Build It, Break It, Fix It (BIBIFI) contests. A number of patterns of developer mistakes leading to vulnerabilities were analyzed. Our work examined the software development process in a real company. Our in-depth ethnographic study is complementary to the analysis based on large-scale competition data. One possible cross-over between the two types of studies is that one can use the insights from one to drive the analysis in the other. For example, an observed real-world phenomenon that has significant security impact could be replicated in the BIBIFI contest to further examine a hypothesis on a much larger and more diverse population.

Oorschot and Wurster [5] posited that developers have different skills which often do not include security and suggest that the focus should be on those who design APIs, because it is unrealistic to expect all developers be taught sufficient security. We raise a similar question in our paper from our ethnographic data, regarding how much security knowledge developers can realistically master, and whether a co-creation model where security experts and developers closely collaborate would be a more effective approach.

Green and Smith [6] discussed that developers are not the problem for insecure code. The focus should be on creating more developer-friendly and developer-centric approaches and supporting them when they are dealing with the security tasks. Our ethnographic data supports this conclusion. Moreover, our fieldwork resulted in a co-creation model that could be part of a solution to provide the needed support to developers for writing more secure code.

In addition to the works mentioned above, the research community has explored this area through a number of angles. Oliveira et al. [7] conducted surveys to understand developers' attitudes toward security which leads to understanding that APIs and tools can be improved significantly. Votipka et al. [8] performed semi-structured interviews to compare how hackers and testers find vulnerabilities. Acar et al. [9] studied whether different documentation resources influence the security of programmers' code. Naiakshina et al. [29] conducted a qualitative study with 20 computer science students and investigated how and why they failed with regards to secure password storage. Gorski et al. [30] designed a controlled online experiment with 53 participants to study the effectiveness of API-integrated security advice. Acar et al. [10] conducted an online study and evaluated five cryptographic APIs with GitHub Python developers about the usability of the crypto APIs. There has also been research that studied and characterized different aspects of software bugs [31–33]. These studies focused on the quality of bug reports and found that important information was often missing in bug reports which made it harder to reproduce and fix them.

To the best of our knowledge, our work is the first in using participant observation and long-term ethnography to study secure software development in a real company. It allowed us to observe and reflect upon all contextual factors that have an impact on secure development processes in a real company. Our data and findings serve to complement the efforts discussed above, and often times reveal deep insights not obtainable through other approaches.

Going beyond secure software development, research into other aspects of usable security has also revealed the importance of incorporating broader stake holders' perspectives in thinking about security solutions [34, 35]. Haney et al. studied the role of cybersecurity advocates within organizations [36–39]. Much of the findings in that line of research echoes ours, in particular the importance of co-creating security solutions with relevant stake holders.

8 Conclusion

This research shows how security intersects with software development on the ground, based on two embedded researchers with one and half years' data. There remains a considerable gap between security and developers. Our research shows that security professionals can better bridge the gap by understanding how (in)security emerges from the interacting technological and human factors in the development process. Our ethnographic study provided a way to understand this complicated phenomenon, both by better understanding the competing demands under which developers work, and by demonstrating how security can successfully be integrated into software development through a co-creation model.

Acknowledgments

We thank Raj Rajagopalan for bringing us the idea of using anthropological methods to study secure software development in its native environment, and continuous support for this research. We thank Michael Hicks and Michelle Mazurek for insightful discussions throughout this ethnographic study. The authors of this paper owe all the above, and the anonymous reviewers, for the many valuable comments that helped tremendously in improving the paper. We owe gratitude to the company, and its employees who participated in the study. This research is supported by the National Science Foundation under Grant No. 1801633 and 1801545. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *14th Symposium on Usable Privacy and Security*, Baltimore, MD, USA, 2018.

[2] Hala Assal and Sonia Chiasson. Think secure from the beginning: A survey with software developers. In *CHI Conference on Human Factors in Computing Systems*, Glasgow, UK, 2019.

[3] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L Mazurek, and Piotr Mardziel. Build it, Break it, Fix it: Contesting secure development. In *2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[4] Daniel Votipka, Kelsey Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *29th USENIX Security Symposium*, Boston, MA, USA, 2020.

[5] Glenn Wurster and Paul C Van Oorschot. The developer is the enemy. In *New Security Paradigms Workshop*, Lake Tahoe, CA, USA, 2008.

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

[7] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *30th Annual Computer Security Applications Conference*, New Orleans, LA, USA, 2014.

[8] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2018.

[9] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2016.

[10] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic APIs. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2017.

[11] Ross Anderson. Why information security is hard – an economic perspective. In *17th Annual Computer Security Applications Conference*, New Orleans, LA, USA, 2001.

[12] Sathya Chandran Sundaramurthy, Alexandru G. Bardas, Jacob Case, Xinming Ou, Michael Wesch, John McHugh, and S. Raj Rajagopalan. A human capital model for mitigating security analyst burnout. In *11th Symposium On Usable Privacy and Security*, Ottawa, ON, Canada, 2015.

[13] Sathya Chandran Sundaramurthy, John McHugh, Xinming Ou, Michael Wesch, Alexandru G. Bardas, and S. Raj Rajagopalan. Turning contradictions into innovations or: How we learned to stop whining and improve security operations. In *12th Symposium on Usable Privacy and Security*, Denver, CO, USA, 2016.

[14] James P. Spradley. *Participant Observation*. Holt, Rinehart, and Winston, 1980.

[15] Kathleen M. DeWalt and Billie R. DeWalt. *Participant Observation: A Guide for Fieldworkers*. Lanham: AltaMira Press, second edition, 2011.

[16] David R Thomas. A general inductive approach for analyzing qualitative evaluation data. *American journal of evaluation*, 27(2):237–246, 2006.

[17] H Russell Bernard, Amber Wutich, and Gery W Ryan. *Analyzing qualitative data: Systematic approaches*. SAGE publications, 2016.

[18] OWASP top 10 most critical web application security risks, 2017. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf.

[19] The open source security testing methodology manual. https://www.isecom.org/OSSTMM.3.pdf.

[20] Karen Scarfone, Murugiah Souppaya, Amanda Cody, and Angela Orebaugh. Technical guide to information security testing and assessment. NIST Special Publication, 2008.

[21] Penetration testing execution standard (PTES), 2014. http://www.penteststandard.org/.

[22] Information system security assessment framework (ISSAF), 2019. https://untrustednetwork.net/files/issaf0.2.1.pdf.

[23] *Agular security*. https://angular.io/guide/security#xss.

[24] Michael Felderer, Matthias Búchler, Martin Johns, Achim D Brucker, Ruth Breu, and Alexander Pretschner. Security testing: A survey. *Advances in Computers*, 101:1–51, 2016.

[25] Hee-Je Bak. Education and public attitudes toward science: Implications for the "deficit model" of education and support for science and technology. *Social Science Quarterly*, 82(4):779–795, 2001.

[26] Susan D Blum. Why don't anthropologists care about learning (or education or school)? an immodest proposal for an integrative anthropology of learning whose time has finally come. *American Anthropologist*, 121(3):641–654, 2019.

[27] Tom Humphries. Schooling in American sign language: A paradigm shift from a deficit model to a bilingual model in deaf education. *Berkeley Review of Education*, 4(1):7–33, 2013.

[28] Steve Lipner. Lessons learned through 15 years of SDL at work. Online Opinion Piece: https://www.csoonline.com/article/3440120/lessons-learned-through-15-years-of-sdl-at-work.html, September 2019.

[29] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong? a qualitative usability study. In *ACM SIGSAC Conference on Computer and Communications Security*, Dallas, Tex, USA, 2017.

[30] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse. In *14th Symposium on Usable Privacy and Security*, Baltimore, MD, USA, 2018.

[31] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *IEEE 31st International Conference on Software Engineering*, Vancouver, Canada, 2009.

[32] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *32nd ACM/IEEE International Conference on Software Engineering*, Cape Town, South Africa, 2010.

[33] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *22nd ACM conference on Computer Supported Cooperative Work*, Savannah, GA, USA, 2010.

[34] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel von Zezschwitz. "If HTTPS were secure, I wouldn't need 2FA"- End user and administrator mental models of HTTPS. In *IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2019.

[35] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. Investigating system operators' perspective on security misconfigurations. In *25th ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada, 2018.

[36] Julie M. Haney and Wayne G. Lutters. "It's scary... it's confusing... it's dull": How cybersecurity advocates overcome negative perceptions of security. In *14th Symposium on Usable Privacy and Security*, Baltimore, MD, USA, 2018.

[37] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. "We make it a big deal in the company": Security mindsets in organizations that develop cryptographic products. In *14th Symposium on Usable Privacy and Security*, Baltimore, MD, USA, 2018.

[38] Julie M. Haney and Wayne G. Lutters. Motivating cybersecurity advocates: Implications for recruitment and retention. In *Computers and People Research Conference*, Nashville, TN, USA, 2019. Association for Computing Machinery.

[39] Julie M. Haney and Wayne Lutters. Security awareness in action: A case study. In *Workshop on Security Information Workers, USENIX Symposium on Usable Privacy and Security*, Santa Clara, CA, USA, 2019.

Appendix: Coding

Coding – including the development of specific codes and of a codebook – proceeded in an iterative fashion. Weekly meetings facilitated the discussion of emerging research themes and specific examples. Codes emerged from these discussions, where the researchers built consensus on specific analyses. The researchers relied on a general inductive approach [16], as well as techniques derived from grounded theory and related approaches for doing qualitative analysis [17]. Overall, the development of codes initially focused on the Silently Allow example, then on emerging results from penetration testing, the development of a specific coding system for each researcher for their field notes, and a final collaborative phase to find commonalities in codes for both the specific examples and overall corpus of field notes.

Initial Coding of Silently Allow

The coding for Silently Allow grew from weekly meetings and initially involved using flash cards to organize information. These flashcards then became organized into different types of Silently Allow, which were written up and shared with the research group for further input. Here are the written notes on the first two categories:

Type 1

Device allowed network access. Device not authenticated but needs network access to function. Silently Allow an "internal state" that is used to get devices onto customers' network.

This is the original "Silently Allow" (SA)

In this case, not a problem with the actual code. They programmed the code to do this. Some other SAs are because of problems with the code itself.

**Need vignette – use field notes and participant observation to describe the situation.*

Potential category: IoT devices
Initial problem: Customer

Type 2

Saml – authorization protocol, a way to give access to the network. This is a protocol problem, because hadn't implemented the necessary checks

This is an issue with users who input credentials and/or third party authentication server.
If user tried to use Saml, went into SA automatically. And thus gets network access.
This problem doesn't affect all users; affects users who are set up with Saml authentication. Users had a single login; with SA, once clicked login, would be able to login, no matter what. Didn't implement security checks. Whatever result, grant access.

Potential category: Authentication
Initial problem: Internal/program

Full Set of Codes

Subsequent research focused on developing a full set of codes by each of the student researchers. Because the two embedded researchers often worked on different projects and at different times, each wrote their own field notes and then subsequently engaged in coding of their own notes. This process permitted inductive analysis from their own data, which could then be shared in research meetings to produce consensus. Below are the sets of codes developed by the two embedded researchers:

Codes Developed by Researcher One

- Ad-hoc-development
- Customer-driven-development
- Debugging-infrastructure-vs-security
- Developers-not-trained-in-security
- Difficult-to-fix
- Inconsistent-narratives
- Ineffective-peer-reviews
- Insecure-defaults
- Insecurity-to-avoid-breaking-implementations
- Lack-of-documentation
- Lack-of-security-audits
- Lack-of-security-awareness
- Learn-by-doing
- Legacy-code
- Limited-testing-practices
- Misconfiguration
- Non-urgent-issues
- Outdated-libraries
- Positive-bias
- Rapid-prototyping
- Reactive-security
- Revenue-streams
- Security-vulnerabilities-documented-as-bugs
- Selling-unimplemented-features
- Social-frictions
- Technically-challenging-problems
- Time-spent-debugging
- Time-to-market-vs-security
- Trivial-tests
- Underestimating-effort
- Unmanageable-code
- Urgencies
- Usability-over-security

Codes Developed by Researcher Two

- Caring-about-subject
- Changing-attitude
- Development-process
- Documents-not-updating-frequently
- Fix-first-update-later
- Joking-about-intern-work
- Knowing-bug-do-nothing
- Lack-of-knowledge
- Learning-process-with-company
- Looking-for-new-idea
- New-idea-vs-tasks
- Not-caring-about-subject
- Not-trusting-other-developer-or-intern
- Performance-reaction
- Protective-about-subject
- Say-something-do-something-else
- Security-vs-performance
- Security-vulnerabilities-blocked
- Security-vulnerabilities-concern
- Security-vulnerabilities-denying
- Security-vulnerabilities-execution
- Security-vulnerabilities-fixing
- Security-vulnerabilities-interested
- Security-vulnerabilities-process
- Security-vulnerabilities-reaction
- Security-vulnerabilities-thinking
- Security-vulnerabilities-upgrading

Final Collaborative Phase

In the final collaborative phase, researchers often worked on a whiteboard to find the overlap between different types of data, specific examples, and inductive insights. For example, "co-creation" emerged as an overarching conclusion that came out of working through the data to find commonalities in both field notes and in researcher experience during participant observation. Reviewing similarities, via the coding and then the notes, also led to data-driven conclusions about what was successful and what proved to be bottlenecks or limitations in cybersecurity during the months of embedded research.