Security Evaluation by Arrogance: Saving Time and Money

Hussain M. J. Almohri
Department of Computer Science
University of Virginia
almohri@ieee.org

Sayed A. Almohri
Columbia Business School
Columbia University
sayed.almohri@columbia.edu

***Abstract*—Software startups can be subject to extreme money and time constraints while hoping for delivering reliable software. In a harsh startup environment, software may face quality downgrade either by improper process management or incapable human resources. Among the many, security is a fragile software quality characteristic responsible for severe negative consequences such as jeopardizing a startup's brand among early adapters. Addressing security evaluation, we report our experience in developing a startup's internal software engineering process that includes a continuous security evaluation cycle at the heart of the process and leverages *arrogance* in software engineering—the tendency to break other team members' code. The valuable outcome was that enforcing security evaluation, as a concrete process activity, came with no cost. That is, we reutilized our resources by changing the flow of the engineering process while capitalizing on arrogance as a motivating stimulus yielding a cost-effective vulnerability assessment for each software release. We describe our process, provide the case for the benefit of arrogant engineers, and conclude with a report of incidents in which arrogance came to our rescue.**

***Keywords*-Security management, Risk analysis, Business process management, Process modeling**

I. Introduction

In 2012, we started a mobile payments startup (Next Payments[1]) far away from Silicon Valley, CA. Our startup's goal was to develop a robust and scalable mobile payment system. At the time, a number of startup companies (such as Square and LevelUp) were gaining tremendous attention and the idea of mobile payments was promising. We started with a relatively small team of engineers, business developers, and social media specialists (a total of 11 team members) focusing on coding, business development, and marketing. A group of four engineers, including a lead senior engineer, developed two platform client applications, a large backend set of RESTful API micro services for client applications, and a log and monitoring component for auditing transactions. The engineers were focused on single platforms with specific task allocations. The engineering team consistently met to brainstorm and exchange ideas and resolve possible problems.

Being remote to Silicon Valley without access to vast amounts of funding available to startups, we were limited to a seed fund of $300,000 and a tight delivery deadline. We could not afford hiring engineers with special skills or dedicate engineers to specific quality testings. Our engineers were mostly recent university graduates that had to translate requirements to code and perform testing on their own. Thus, during the early stages of our development, we were faced with a serious problem: we had a very limited understanding of our current state of security. We were not confident whether a release of the software was secure against powerful client or server-side attacks. While we had proper security policies and specifications for our transactions, the uncertainty about transactions correctness was a growing problem. There was an inevitable need for an efficient process to control the software security quality while maintaining efficient use of our financial resources. In fact, we wondered whether a security evaluation with near zero cost was possible.

Unfortunately, the research literature has a narrow view of methodologies and guidelines for security evaluation of software developed in startup environments. Also, for a startup teams, searching, learning, and correctly applying a theoretical process such as Microsoft's Security Development Lifecycle [5] may not be feasible. Thus, to develop a clear understanding of the security of our system, our ideas was to developed an *internal security-centric process* that is simple and fluid and has a security evaluation phase ensuring vigorous system security testing before every release. This security evaluation phase relied on the passion and desire of arrogant team members to lead aggressive security evaluations with the intention to break code written by others. Our hypothesis was that an arrogant engineer with a technically negative attitude towards colleagues can be an asset instead of a burden. That is, the engineer's desire to find vulnerability in the release, perhaps to prove a point against others, has a valuable and positive outcome. Thus, our process reutilizes the capabilities of existing team members by leveraging human factors (largely supported by previous studies [4]) with minimum training and adaptation efforts.

This paper will provide the details of our internal process that uncovered a number of serious vulnerabilities either before the final commit of the software or early after its release into production. This paper does not provide the architecture of our mobile payment system or an experimental evaluation of our internal process. Instead, we'll discuss the motivation and the design of the process and a number of cases in which it proved financially efficient as well as effective in preventing

[1] https://nextkw.com

vulnerabilities.

II. The Security-Centric Process

To develop the system in a timely manner, Agile processes, especially Extreme Programming (XP) and Scrum, were the strongest candidates for us partly because they require less formal documentation and emphasize more on effectiveness of teamwork. That said, as a study by Reifer finds out [6], a number of Agile-related practices in participating companies were merely new ways to manage classical software projects. For instance, customer collaboration, product demos, and full stake-holder participation were among the list of effective Agile practices that contributed to improved results in quality, delivery time, and some minimal cost reduction.

In terms of security, Scrum and XP do not provide specific recommendations on handling security-related evaluation. Although evaluation, testing, verification and validation are integral parts of any software engineering process, there are limited guidelines on effective security evaluation especially in small startup teams. We needed to rethink and extend an Agile process to fit our security evaluation goals. With a very limited budget of nearly $300,000 and a long list of features for the minimum viable product, we could not afford fancy process requirements that required extra time and effort. However, as we discuss in Section II-B, we reutilized our human resources relying on their passion to lead thorough security evaluation at the end of every release of the system.

A. The Early Stages Process

We started developing the first four releases of our system with a simple process that was inspired by Extreme Programming and included planning and feedback loops. After each release, once the business team compiled the feedback and concluded that changes were needed or new features had to be added, a joint meeting would plan the next release. The software engineering team was responsible for developing clear tasks that were assigned to individual team members.

In the first year of our startup, we noticed a lack of a comprehensive understanding of the current state of security in the system. None of our engineers was confident whether our system releases were secure, at minimum, against vulnerabilities such as denial of service, memory corruption, plaintext storage of data, and other vulnerabilities that will have result violate our system's integrity, confidentiality, and availability. The intensity of the development, the rapid changes in the requirements, and the pressure on the team to spend the minimum time for developing a component, were among the reasons that delayed a thorough security testing of both the applications and the backend services. Further, there was a lack of interest in security in most other software engineers. This lack of interest in them was a potential problem that could leave severe vulnerabilities in the code.

Thus, we started further enhancing our process to include security evaluation steps. The high-level requirements for the new process were as follows.

1) The enhanced process should continue to be simple and should not require significant training for the team members.
2) The enhanced process should ensure the final product was tested against a set of fundamental security vulnerabilities and should include security features that, given the current technology, guaranteed reliability of transactions.
3) The enhanced process should not require additional dedicated team members.

While the requirements were clear for enhancing the software engineering process, we also needed a mechanism to produce releases with an acceptable level of security, which was defined as the extent of our knowledge and research of attacks and defenses related to our platforms. Thus, we decided to utilize human factors to accomplish our goal. Considering the behavior and attitude of our engineers, we found a rather undesirable personality: one of our engineers enjoyed harshly criticizing others' code and their design decisions. On the bright side, however, he had interests in software and network security and would constantly research topics in this area.

We realized this arrogance towards other engineers as an opportunity. The hypothesis was that given the aggressive and critical behavior of the engineer, testing others' code for security would be a natural stimuli for intrinsic motivation. Although the arrogant engineer's experience in software development did not exceed three years of prior work, compared to other team members, he had consistently showed personal desire in understanding security attacks and how software often fails in maintaining minimum security. The arrogant engineer could not be characterized as a security expert. However, we can summarize his belief of security in that

1) he did not trust anyone's code for having minimum security even if the software was a more experienced engineer's work, and
2) he was never convinced that accepted standards, such as transport layer security (TLS) were sufficient for ensuring integrity of data reaching the backend servers. For example, he always preferred an end-to-end encryption layer on top of TLS.

The arrogant engineer's belief constituted a useful ground for utilizing his beliefs and personal desires. Based on the hypothesis that his attitude and beliefs will result in finding security flaws in our work, we extended our software engineering process by rearranging our resources to realize the team's potential to deliver secure software. The result was a Scrum-inspired process (Figure 1) which emphasized on the review stage of a sprint. Breaking down our hypothesis, our new process had to test the following ideas.

1) A personal desire for proving a security failure in an engineer with a basic understanding of security will be an intrinsic stimuli for the engineer. This stimuli will motivate the engineer for effective research in security advancing his/her capabilities.
2) Adapting a central role of security without the need for

hiring new engineers will assist in maintaining a low *burn-rate* (the negative cash-flow of a company) of our seed capital.

As we will describe in Section II-C, we could indeed provide enough motivation for an arrogant engineer to both pursue state-of-the-art security practice and also to attack others code. In fact, we split the security evaluation task between an arrogant engineer and an external ethical attacker to compare their results. As suggested in the results of Section III, the arrogant in engineer was more successful in terms of finding the higher number of vulnerabilities in the system.

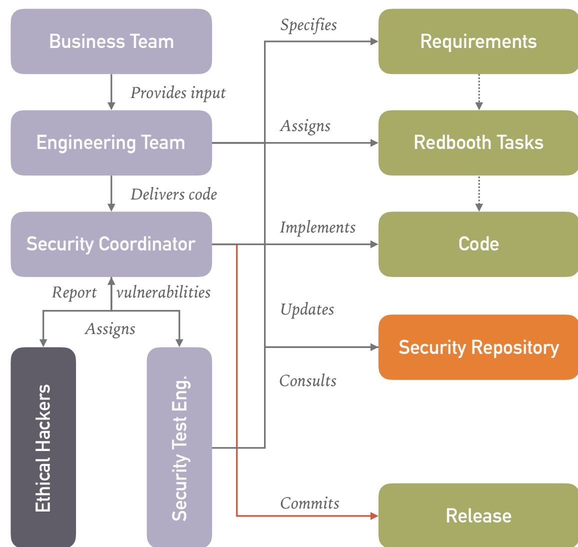

Fig. 1. An overall flow of our software engineering process with a central focus on security engineering. This flow demonstrates a single release lifecycle that starts with a joint meeting of the teams and ends after a thorough security testing. Purple boxes represent actors, arcs represent the direction of interaction, arc labels represent actions, green boxes represent artifacts, the orange box represents data, and dotted arcs represent influence. The last action is on the red arc.

B. Characteristics of the Security-Centric Process

In Figure 1, we demonstrate the main stakeholders, the artifacts, and the flow of tasks in the security-centric process. The light purple boxes represent the main stakeholders, the green boxes represent the artifacts, the dark purple box represents external ethical attackers recruited for security testing, and the orange box represents data. The arcs represent the direction of interaction and the arc labels represent actions.

The security-centric process receives input from three sources: the business team as the primary provider of system requirements, the engineering team, and the security coordinator, with the latter two influencing the requirements mainly by introducing technical changes to the main business requirements. Only the security coordinator commits the final release after all the necessary security tests pass. In our case the security coordinator was the senior engineer who was in charge of the engineering team.

The process includes a release lifecycle (similar to a sprint in Scrum) that starts with a joint meeting of engineering and business teams. In the joint meeting, the teams (which are the main stakeholders) discuss, determine, and agree on the requirements for the current release lifecycle. The engineering team prepares a rough draft of the requirements during the meeting. In the hours following the joint meeting, the engineering team translates the requirements into tasks that are posted on an online collaboration application, in our case Redbooth [2]. Redbooth played a central role in the delivery of tasks as it featured automated reminders and deadline notifications.

As components were developed and passed initial tests, they were delivered to the security coordinator who would in turn (i) hire external ethical hackers and (ii) specify security tests and properties for the *arrogant engineer*, which we will refer to as our security test engineer. Recall that our main hypothesis was that our security test engineer had an intrinsic motivation to break others' code. To communicate testing goals to the security test engineer, the security coordinator updates the security repository which includes the desired security properties and in some cases a specification of the test. For instance, when adding a new publicly accessible server to our cluster, we would ask our external ethical hackers to attempt an intrusion attack and report the results. The security test engineer, however, mostly focused on the code developed by other engineers. A security test specification example was to test a number of SSL implementations in our client applications, which eventually unveiled serious flaws (discussed in Section III).

When external ethical hackers and the security test engineer submitted their reports to the security coordinator, the engineering team would meet to discuss and plan to remedy the vulnerabilities. When all the fixes in the code, policies, and system parameters were discussed, only the security coordinator had the permission to commit code or approve a release by engineers. As the security coordinator was responsible for the final release, our process guaranteed that at least the specification in the security repository was evaluated.

In summary, a complete release lifecycle must go through the following steps.

1) A joint meeting of business and engineering team to determine and agree upon the requirements for the release.
2) The engineering team prepares a draft of requirements and translate the requirements into tasks that are posted on an online collaboration system.
3) The engineers start coding and delivering components to the security coordinator.
4) The security coordinator dispatches security tasks to external ethical hackers and the security test engineer.
5) The security coordinator receives a report of potential vulnerabilities in the draft release and calls for a meeting with the engineering team.
6) The engineering team decides on a plan to remedy the vulnerabilities and immediately starts improving the release.
7) The security coordinator constantly monitors the im-

provements in the release with the engineering team.
8) The security coordinator commits the release (or approves a commit) when the release reaches a *good security* level.

This process was a natural fit for all the team members. With small and focused business and engineering team, we had all the components needed to effectively realize new features in software while avoiding security pitfalls. As to our knowledge of security we had a higher level of confidence and understanding of the state of security in a release compared to our simplistic earlier extreme programming-based process. Although, processes such as Microsoft's security development lifecycle [5] are beneficial, as a small startup team with tight resources we could not afford consuming resources for training and executing these processes.

C. The Case for Arrogant Evaluation

Our engineering team included a single engineer (i.e., the security test engineer) which enjoyed breaking the code written by other engineers. He relentlessly tried to break the code, mainly in our client applications, even if he had to stay for long hours. He had a passion to find state-of-the-art attack techniques and static analysis tools to test the code at the end of every release. He was willing to perform black box testing for hours to test the functionality of the release while at work or out of the office. Thus, we could discover vulnerabilities without consuming extra resources except for a few occasions where we hired external ethical hackers. Despite his efforts in finding vulnerabilities, the security test engineer could balance between his own tasks and his testing tasks as he was willing to work for extra hours.

The code written by the security test engineer himself was mostly tested by himself. This was a limitation of our work as we could not find the same passion in other team members. However, with larger teams the possibility of having more people interested in the task will increase. Also, while this arrogance might not be immediately a desired quality when interviewing engineers, we argue that arrogant engineers could provide this opportunity to thoroughly test the code.

In our experience, the team was small and already included one arrogant engineer, which responded positively to the requests for evaluating others' code. Scaling this experience to large teams could face challenges. On one extreme, large teams with a large number of arrogant engineers can potentially result in costly conflicts that may not improve the security at all. On the opposite extreme, large teams with no arrogant engineer will not benefit from our model. Thus, to have a deterministic model for using security evaluation by arrogance, we believe team leaders must identify and assign (or even recruit when one does not exist in the startup) a single arrogant engineer to each team of *reasonable* size, which we define as a team that delivers software releases that are sized to the capabilities of the arrogant engineer for a timely evaluation. The exact parameters of security evaluation by arrogance for large teams requires further investigation that is out of the scope of this work.

III. RESULTS

Table I presents a number of statistics about our development efforts, which will shade a light on a number of cases where our vigorous security evaluation revealed vulnerabilities in our system.

	Web Services	iOS App	Android App
Lines of code	89K	28K	20K
Human Resources (HR)	1	1	1
HR cost	$50K	$60K	$60K
Major vulnerabilities	2	3	2

TABLE I
STATISTICS OF OUR DEVELOPMENT EFFORTS FOR A PERIOD OF 24 MONTHS. APPROXIMATELY 15% OF THE CODE WAS DEVELOPED BEFORE USING THE SECURITY-CENTRIC MODEL IN A PERIOD OF SIX MONTHS. ALL THE VULNERABILITIES WERE DISCOVERED AFTER WE ADAPTED OUR NEW PROCESS. OUR SECURITY EVALUATION COST IN FINDING THE MAJOR VULNERABILITIES WAS NEGLIGIBLE.

Vulnerabilities in SSL. In the first release lifecycle, the security test engineer used a number of software tools to perform security testings on the iOS app. His goal was to find out whether our SSL implementation can be broken. After a block box test using a proxy software, he pointed out that the order of calls to `NSURLConnection` [1] API was flawed. Our iOS application's SSL connection could be completely compromised by a man-in-the-middle that could control the wireless router. Our Android application, on the other hand, did not suffer from improper use of SSL. On the server side, the security test engineer was the first to inform of us the existence of the Heartbleed vulnerability [3] in the openssl version on two servers that handled payment transactions.

Authentication policy vulnerabilities. An authentication vulnerability allowed a large number of users to log into both iOS and Android applications without providing a correct combination of username and passwords. Our application only required a username and password, which, due to a simple bug in the backend code, allowed for wrong credentials to be accepted. This finding was also due to the efforts of our security test engineer who did several authentication testing with every release.

Input validation vulnerability. A web-based static analysis tool by a third party company reported that one of our micro services suffered from input an validation vulnerability in a PHP script. The software did not provide further details and this report was included as a task for the security test engineer. After several test, the engineer was unable to find the vulnerability partly because of the large code base and the incomplete report that did not specify the location of the vulnerability.

Intrusion in a database server. The only major vulnerability found by external ethical hackers was their ability to gain access to one of the database servers on Amazon Web Services. We used the input provided by the hackers to patch vulnerabilities and update software on the server. We recruited external ethical hackers only in two more releases for testing our server configurations. They did not find further

vulnerabilities.

Other cases of minor vulnerabilities in various parts of our system were discovered during several releases. Due to lack of proper documentation of those cases, we refrain from providing further details on those case. While, the major vulnerabilities were not very frequent, the impact of each individual vulnerability was high.

IV. Conclusions

Our experience provides a hint for the possibility of using human factors and intrinsic motivation of team members to perform security evaluation within a harsh software startup environment. While our process may seem simplistic, it did provide a smooth and flexible platform that accomplished our goals. An important outcome of our process was the negligible cost of the security evaluation process which heavily relied on human factors and reutilized resources already allocated for other purposes. We do acknowledge that the discovery of the vulnerabilities in our code could have been the result of maturity of engineers in general or the increasing size of the code. While we did observe the passion and the personal desire to perform security evaluation, the effectiveness of human factors and intrinsic motivation of our team remembers remain to be scientifically established.

The problem we faced in executing a rigorous security evaluation using existing methods was the lack of proper training, insufficient funds to attend conferences and workshops, tight delivery schedules, and above all, limited applicability of generic models in a startup environment. We hope our experience ignites further development in utilizing and balancing human factors aiming for efficient and effective enhancement of software quality in startups.

References

[1] Making HTTP and HTTPS requests. https://developer.apple.com/. Accessed: 2017-01-17.

[2] Redbooth, online team collaboration software, tools. https://redbooth.com. Accessed: 2017-01-17.

[3] The Heartbleed Bug. http://heartbleed.com/. Accessed: 2017-01-17.

[4] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp. Motivation in software engineering: A systematic literature review. *Information and Software Technology*, 50(910):860 – 878, 2008.

[5] M. Howard and S. Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[6] D. J. Reifer. How good are agile methods? *IEEE Software*, 19(4):16–18, July 2002.