Software Security and Reliability in Financial Organizations: A Multi-layered Approach to Delivering Safe, Secure and Reliable Software

Parag Patel, Sotiris Skevoulis, Member IAENG

***Abstract*—Developing safe and reliable software is not a luxury but a necessity given our staggering level of dependency on it. Indeed, we have handed over our ability to function day to day to an electronic array or zeros and ones and we expect, demand and rely on these invisible bits to work without error or fault. However, the process of building such reliable software is far from simple. This paper will examine the practical aspects of the problem as well as solutions that are either used or in the process of being evaluated and/or adopted by an NYC based financial institution to produce safe and reliable software.**

***Index Terms*—Safe Software, Secure Software, Reliable Software**

I. Introduction

Over the last thirty years, as if in conformance with a software equivalent of Moore's law, software has surpassed frontier after frontier of human experience and pervaded almost every aspect of our lives. High level languages have made it possible for the masses to harness the power of the CPU and create innovations that have changed how we experience the world. Technology and the software that drives it is so interweaved into daily life that many may suffer an existential crisis if it were ever not to be there. Software is used to control everything from our transportation and shopping to our health and agriculture. And through all of these, our very lives. Businesses are using software on an unprecedented scale and any impact to technological driven business operations would be devastating.

However, in step with these developments, we must be absolutely cognizant of the safety nets we erect to protect against elements that should never go wrong. We routinely entrust our lives to software, to a string zero's and one's. As the application of software continues to evolve and frontier technologies such as robotics and artificial intelligence become standard patterns in the tapestry of daily existence, there must be a corresponding growth in the evolution and development of our safety nets. Failing to do so will incur a terrible price.

Manuscript received December 20, 2016; revised January 16, 2017.

P. A. Patel is a Pace University Alumni. He works as a cyber security security architect at an NYC based financial company (papatel108@gmail.com).

S. Skevoulis is a Professor and, Chair of the Software Engineering Program at the Seidenberg School of Computer Science and Information System at Pace Univerrch (sskevoulis@pace.edu).

Yet, producing software that is safe, secure and reliable is no simple task. We have learnt much from our past mistakes and one thing is certain; there is no "flick of a switch" that can create secure and reliable software. Rather, there are many switches that have to be flicked and in the correct order and in the right way. Secure and reliable software is the result of a process driven methodology that has been faithfully executed with care given to each step of the process. So, what are these steps, what problem does each of these steps solve and who is responsible for executing each step? These questions will be addressed in the remainder of this paper with a particular focus on the financial domain.

The Need for Solid Software in the Financial Organizations

In the modern era, the financial services industry would be unable to function without software technology. Software is used for everything from trading (i.e. placing buy and sell trade orders. Certain high frequency trading companies also use software to engage in algorithmic trading) to record keeping. Software that works incorrectly can cost a financial institution much in terms of reputation, trust and money. For example, imagine a customer placing an order to buy 100 shares of a certain stock and the software erroneously buying 1000 instead. This is a simple and improbable example but as we shall see, given the right sequence of events on untested software, it is a possible scenario. The practices outlined in this paper describe are not only applicable to the financial services industry but to every industry, company or organization that requires bulletproof software.

Before we proceed, let us understand the problem that we are trying to solve; creating secure and reliable software. There is an overlap in meaning between software that is secure and software that is reliable. Software that is reliable is software that functions in a consistent and deterministic manner. It can consistently be relied upon to perform its assigned task. Software that is secure builds upon the bedrock of reliable software but is enhanced with a level of robustness that is able to deflect attempts to infiltrate and modify its behavior. Hence, reliable software is an essential quality of secure software. Conversely, software that is built to withstand illegitimate penetration efforts but cannot be relied upon to meet its functional objectives in a consistent manner, cannot be deemed secure.

Having identified two essential qualities that we want to achieve, all important question, how is it achieved? There is a separate approach for each goal and some overlap.

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

II. Producing Reliable Software

A 2014 paper published by Usenix [1], identified that most catastrophic software failures are the result of incorrect error handling of *non-fatal* errors. They identified predominantly three ways in which developers mishandled coding exception blocks;

1. the exception block is defined but left empty
2. the exception block is too generalized; it catches multiple types of exceptions but does not distinguish in how it reacts to them, instead it performs the same response, which could be inappropriate depending on the exception
3. the exception block logs the exception but does not handle it

The analysis performed by the authors of this paper revealed some very interesting findings. One of the key findings was that almost three quarters of failures are deterministic, meaning that given the right inputs and right sequence of events, they can be recreated at will. Not only that, but the problem is widespread, affecting some of the most recognizable web sites in existence today. It states that:

"…in an outage that brought down facebook.com for approximately 2.5 hours, which at that time was "the worst outage Facebook have had in over four years", "the key flaw that caused the outage to be so severe was an unfortunate handling of an error condition" [5]. In the outage of Amazon Web Services in 2011 [6] that brought down Reddit, Quora, FourSquare, parts of the New York Times website, and about 70 other sites, the initial cause was a configuration change…"

Error handling is one of the most critical, and simultaneously, one of the most difficult areas of software design. Ironically, it is a subject that garners a low level of interest in the mind of the general software developer. This is not surprising; developers typically work within time restricted windows to design and implement core functional requirements. Much credit is given to meeting these requirements. Little is given to the implementation of great error handling.

According to the paper, failures are generally complex. They require a specific sequence and combination of no more than three input events to reproduce. And key, most times when an actual input is needed to generate failure, the value does not matter. What matters is the sequence of input events, rather than the value of the inputs. The paper identified key areas such as the starting of services, a file or database write from or to a client and an unreachable network node where the majority of these errors occur. This information provides actionable insight to a development team.

A. Creating Robust Error Handling Routines

In order to develop a robust and comprehensive error handling routine, a thoughtful analysis of the possible checked and unchecked exceptions must occur.

Figure 1 shows an (incomplete) example of the type of brain storming that should happen for each exception block in a codebase. Many more events and responses could and should be mapped out, however, the idea is to give an example of the thought process. Additionally, parsing the cause of an error may lead to the conclusion that the error is transient and the instruction can be retried. In this way the functional resiliency can also be increased. However, the chief benefit is that the code becomes highly deterministic upon entering an error state.

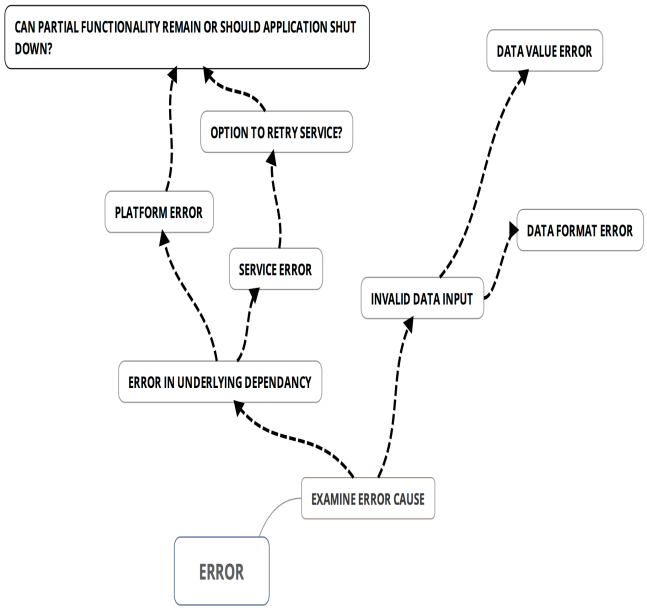

Figure 1: An (incomplete) mind map illustrating the brainstorming process employed to trace every possible cause of an error. Once an error cause has been determined, error handling options can be understood and choices can be made by the software. This leads to a deterministic outcome which is generally a safe outcome.

Developers should follow each type of error to its logical conclusion in order to gain an understanding of how their software will react in a failure scenario. It is a pretty safe bet to say that this not a common practice and so when software enters a significant failure state it leads directly to uncertainty and doubt. Coupled with the fact that developers tend to be fairly nomadic with no guarantees that the author of the code will be around, poorly designed error handling and poor documentation of expected error handling flows, significantly increases time to recovery, as well as decreasing customer confidence and the opportunity cost of developers spending time on root cause analysis. For critical systems, the upfront investment in taking the time to develop well thought out error handling scenarios can pay dividends many times over.

B. Understanding Dependency Hierarchies

Modern software architectures have been steadily migrating toward the service orientated architecture (SOA) model for many years. This architecture has provided many proven benefits, a few of which include an easy way to provide functional reuse, scalability and resolution of interoperability issues. SOA however does have the potential to create a complex set of interdependencies, with a service depending on one or more other services in order to function. This architecture is well established at both the

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

macro-level, with services depending on other services and at the micro-level, i.e. components within a single service depending on each other. Given this sometimes complex set of interdependencies, it is extremely helpful to map out the dependency hierarchy, in order to understand the real impact of a specific dependency being unavailable.

Consider Figure 2 below. Imagine that Service 1 provides three functions, function A, function B and function C. Imagine that each of these functions, depend on a sub-service with the same letter. So, function A would rely on sub-service A, function B, on sub-service B and function C on sub-service C. There are also other interdependencies as can be seen in the diagram.

Now, if sub-service G is unavailable, then we know that sub-service C would be directly impacted and that would have a direct impact to function C on service 1. We also can see that if sub-service F is unavailable, then, not only would sub-service C and function C be impacted but sub-service E would also be impacted which would sub-service B and ultimately function B on service 1.

Although this may seem obvious, it is because we have a clear view of the dependencies. Most of the time, these dependencies can only be ascertained by reading through source code or configuration files, which, is tedious and error prone.

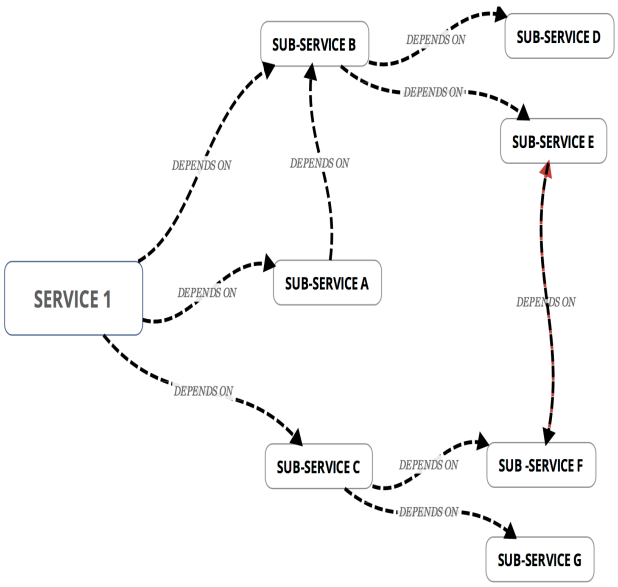

Figure 2: The complex state of interdependencies that exist in modern software architectures.

The value of having a dependency hierarchy can be understood by a common real word scenario; consider a change made to sub-service E due to a requirement in sub-service B. The developer may perform extensive testing on sub-service B and function B on service 1. However, unaware that sub-service F also has a dependency on sub-service E, no testing is performed and therefore sub-service F breaks, ultimately cause function C in Service 1 to break. Even worse, if a significant period of time elapses before function C is called, tracing back the root cause will be ridden with complexity, especially if no hierarchy map exists.

The type of operations described above do require an investment in time with no visible benefit to the end user, yet, depending on the function that the software serves, it could save reputation, large sums of money or most importantly, lives.

C. Order of Operations

Much like the order of mathematical operations, the order of logical operations matter. Let us consider a most basic example. Imagine we have four operations; Create-File Read-File, Write-File and Delete-File.

These operations will only work in a certain order; we cannot read from, write to or delete a file that does not exist. So the first operation must be Create-File. Now we can perform any of the three remaining operations but, if we delete, then the read and write operations will fail.
Of course, no one would create a file and delete it straight away but this may happen through poor implementation of exception handling processes, especially in the case of multiple processes working with a single share resource. In these types of scenarios, using exception blocks to reflect the state of a system such that other processes can be made aware of is vital to producing a stable system.

D. Proving Software Reliability

E. W. Dijkstra, the Dutch computer scientist, famously noted that "Testing can only show the presence of errors, not their absence" [7]. While there is no denying the logic of this statement, it is also true, that the only way to provide empirical assurance that software is reliable is through comprehensive testing. Of course there are multiple levels of testing, yet the key is to start at the grass roots level, closest to the code. Developers must write comprehensive unit tests for each functional unit of code that are inclusive of comprehensive failure tests, that is, tests to validate behavior of the functional code unit in a failure state. While it is common practice to for developers to write test cases that test the specific functionality of code, it is far less common to write test cases that provide coverage of code behavior when code reaches error states. However, such comprehensive unit testing of code in failure state is difficult to accomplish unless each possible failure is enumerated and its response is clearly identified (as shown above). Conversely, if these tests are well documented along with the expected resolution path then writing comprehensive tests become much easier.

Once the tests are written, they can be triggered at key timelines during the day. For example; the automated running of tests prior to checking code into a repository is becoming a standard across the development community. Automated tests are also run at the end of the day and prior to a daily build process to validate the consistency of the code base. Although these practices are being adopted, the point here, is that the success of these types of practices depend largely on the quality of the tests being run. Unless tests include comprehensive coverage of failure scenarios and recovery actions, they have limited value in terms of proving reliability.

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

III. PRODUCING SECURE SOFTWARE

Building secure software is a process that starts from the first day of a project. It demands a mindset that views every step of the software development life cycle (SDLC) through security shaded spectacles.

Every software application is trying to solve a problem. The functional requirements define the functionality required to solve the problem. Security requirements, which are classified as non-functional requirements, describe the security constraints required by the application. So the question becomes, how to determine the stringency of the security requirements? The answer is by determining the financial, reputational, legal and moral cost of compromise of the application or its data. The greater these costs, the more stringent the security controls need to be. So the first step in designing secure software is to define risk (based on the cost) of application or data compromise and thereby define the scope and stringency of the security controls.

Time and experience has distilled a foundational set of security requirements that must be met by software applications that are anything more that static HTML. These requirements can be viewed as architectural building blocks that will provide a certain level of assurance that software is secure. Let us examine each one of these blocks.

A. Authentication

Authentication is the process of establishing the identity a user. This is usually achieved by user name and password. Although far from ideal it remains the prevailing standard across most applications. Some U.S. states are toying with the idea of introducing legislation that would require the use of multifactor authentication for internet facing sites that allow the movement of money. Multifactor authentication creates a requirement for a user to identify themselves using at least two identification factors. Multifactor authentication consists of three possible authentication factors: something you know, something you have or something you are. A password fulfils the "something you know" requirement. The "something you have" requirement is commonly fulfilled by the use of an RSA token. The RSA token provides a user with a one-time password that changes every 60 seconds. Since only the user has the token, only the user knows the one-time password. The "something you are" requirement can be fulfilled by some sort of a biometric measurement such as a retina scanner or finger print reader. Multifactor authentication commonly requires that a user identify themselves using two of the factors and for this reason is sometimes called dual factor authentication. Multifactor authentication does provide authentication that is exponentially stronger than single factor authentication but comes at a cost. For example, imagine the governance and cost implications of providing RSA tokens to tens of thousands of application users. It is for this reason that wide spread user of multifactor authentication for public users of banking applications is limited. However, due to the emerging ubiquity of the smartphone, it may be possible to piggy back the distribution of smart tokens ("something you have") and so provide one-time passwords to users without the need to distribute hardware for RSA key tokens.

B. Authorization

Authorization is the process of identify what functionality a user is entitled to in a particular application. Standard methods of doing this involve using role based authorization. In this technique, specific entitlements are associated with specific roles. Application users are added to these roles depending on their required entitlements.

C. Data at Rest Protection

Protecting data at rest is usually achieved through symmetric key encryption. It is important that outdated algorithms are not used. The current standard is AES with a key size of 128, 192 or 256. The key greater the key size, the stronger the encryption at the cost of a marginal overhead in CPU time and actual time. Unless there is a time based constraint (such as a high frequency trading application) a 256-bit key should be used.

D. Data in Transit Protection

Protecting data in transit is usually achieved through TLS (Transport Layer Security) using Public Key Cryptography. Public key cryptography involves the use of public certificates. It is important to verify that industry standard key strengths are used such as 2048 bit or 3096 bit. Again, the larger the key size, the greater the processing overhead, although given the abundance of powerful low cost hardware today, performance due to stronger keys has become less of a problem. Also defunct ciphers (such as RC4) should be disabled and TLS should be configured only to use non-vulnerable versions (currently anything above TLS 1.0).

E. Ensure Compliance with Latest Security Standards

This point has been covered somewhat in the above sections but its importance cannot be overstated. Many preventable attacks occur because security components in software continue to use algorithms or protocols that are known to be vulnerable. An example of this is the 2014 Poodle attack, which was able to succeed because SSL version 3 has not been disabled. Other examples are the use of SHA1 instead of SHA-2. Although SHA1 has been known to be vulnerable for many years it is still used by developers not familiar with the security landscape and are instead focused on meeting functional requirements.

F. Review of Security Architecture Design Constructs

The design of the security architecture constructs (and their implementation roadmap) outlined above should be reviewed and validated by a security team before construction begins. Since application architects are not typically security experts, it makes sense to have the architecture design reviewed by a security focused team to ensure that relevant versions and constructs are used. If properly designed and implemented, these blocks will ensure the foundational security of the application.

G. Code Level Security

The above principles describe the security focused foundational building blocks of an application. However, by themselves, they are not enough. When building a house, it is essential that the structural components are solid so that the building can withstand natural elements such as wind and rain. However strong the foundations, if there are no locks on the windows or doors, the building remains

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

insecure. In software terms, the above building blocks are the foundational elements. The "locks" however, are designed and built at the code level.

The proliferation of attack vectors, aimed at exploiting software with no or very weak locks can be overwhelming. For those whose main goal is functionality and not security (as is the case with most application developers and architects) an approach that makes sense is to take advice from experts in the security community. The Open Web Application Security Project (OWASP) is a consortium that amongst other things provides a "Top 10" list of the most critical web application security risks, examples of vulnerabilities and guidance on how to avoid [2]. Simply understanding the most critical attack vectors and implementing appropriate code level "locks" is enough to make a huge difference in the security of the application. OWASP has published and Enterprise Security API (known as ESAPI) that can be used to thwart a number of the attacks on its "Top 10" list.
The key points here are that application security controls are *vital* and even non security focused developers can make their applications significantly more secure by understanding and following the guidance provided by OWASP.

IV. Testing for Security

As with reliability testing, the only way to empirically provide some degree of assurance of application security is to test for it. There are three types of security testing that are commonly performed for high risk applications; static and dynamic testing and penetration testing.

- Static testing analyzes code in a non-runtime environment. The idea is to search source code to identify exploitable vulnerabilities. Although static analysis can be done manually, for anything other than a small project, a manual analysis would be infeasible. Typically, enterprises will use tools to perform static analysis code scans. There are many vendors offering static analysis tools and one of the more widely used tools in HP Fortify.

- During dynamic testing is performed whilst the code is in operation. Wikipedia, defines dynamic testing as follows, "Dynamic testing (or dynamic analysis) is a term used in software engineering to describe the testing of the dynamic behavior of code. That is, dynamic analysis refers to the examination of the physical response from the system to variables that are not constant and change with time. In dynamic testing the software must actually be compiled and run. It involves working with the software, giving input values and checking if the output is as expected by executing specific test cases which can be done manually or with the use of an automated process."

- Penetration Testing

  In penetration testing ethical hackers to try and break into an application, in much the same way that malicious hackers would. They attempt this by using a variety of tools such as network and port scans and also by launching injection attacks, session attacks etc. They document any vulnerabilities found and provide to the application team for remediation measures.

Static and dynamic testing are essential. Traditionally, static testing is performed once full development on the application is completed. However, this is a costly and dangerous practice. It is costly because it requires potentially significant rework by developers to fix security flaws. It is dangerous because management, often in a rush to get their software product to market, defer security remediation's until a later date, and so push flawed code into a production environment. A better idea would be to "bake" security into the development lifecycle so that it is an organic an integral part of the finished product and that is what the Secure SDLC tries to address.

V. Secure Software Development Life Cycle

The concept of the secure software development life cycle emerged to address concerns surrounding the lack of security focused requirements and processes during the software development life cycle (SDLC). The secure SDLC addresses security concerns and mitigating processes at each step of the software development life cycle. Figure 3 below illustrates the security touch points that are addressed as part of a secure SDLC.

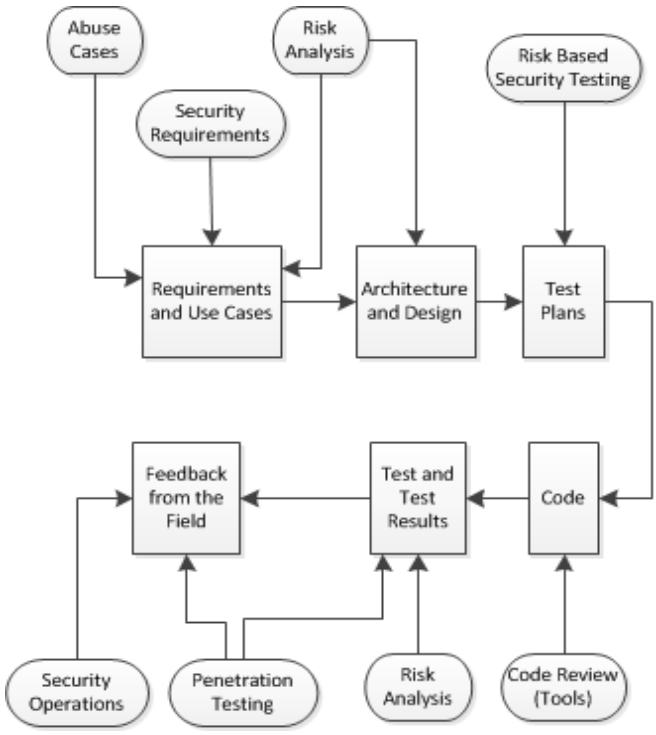

Figure 3: The Secure SDLC (Citigal)

According to Citigal, "In the past, it was common practice to perform security-related activities only as part of testing. This after-the-fact technique usually resulted in a high number of issues discovered too late (or not discovered at all). It is a far better practice to integrate activities across the SDLC to help discover and reduce vulnerabilities early, effectively building security in."

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

And again "Generally speaking, a Secure SDLC is set up by adding security related activities to an existing development process. For example, writing security requirements alongside the collection of functional requirements, or performing an architecture risk analysis during the design phase of the SDLC" (Citigal).

VI. BUILDING SECURITY IN MATURITY MODEL (BSIMM)

Recently a study was carried out that used the "Building Security Maturity Model" (BSIMM) to conduct a survey that analyzed data from 78 firms. BSIMM is a tool that allows a firm to directly compare its software security approach to the BSIMM community through 112 well-defined activities, organized in 12 practices. More information on the tool can be found at www.bsimm.com. Four "indisputable" findings about software security within corporations emerged from the study.

One of the key findings is that in order to build effect security into software, a software security group (SSG) must exist as an independent entity with in the organization. As the paper states:

"At the highest level of organization, SSGs have five major roles:

- Provide software security services
- Set Policy
- Mirror Business Unit Organizations
- Use a hybrid policy and services approach
- Manage a distributed network of those doing software security work" [9]

The make-up of the SSG (the subject of the second key finding) must include individuals from disparate backgrounds. This finding points out that in order for an SSG to be effective its members must have different skillsets. Someone well versed in code level vulnerabilities may draw a blank when reviewing for architectural vulnerabilities and vice versa.

However, it is the third finding that truly reveals a key structural difference among firms scoring the highest BSIMM scores, and that, is the use of satellites. The paper states that "one of the most commonly held myths of software security is that developers and development staff should 'take care of' software security". [9] However, the BSIMM studies have shown that this is not the case at all and that an SGG group is necessary. Having said this, the paper suggests that developers should be directly involved in security and serve as satellites to the SSG. As the paper says, "Each of the 10 firms with the highest BSIMM scores has a satellite (100 percent) with an average size of 131 people" [9]. Again, "In fact, satellites play a major role in executing software security activates among the most mature BSIMM community firms" [9].

The paper suggests a hub and spoke type structure with the SSG serving as the hub and satellites serving as spokes as an effective security model. This model provides the benefits and efficiencies of both a centralized and decentralized model and makes much sense. However, in order to achieve this, firms need to encourage developers to "get engaged" in security and to form satellites. There would also be an onus on the SSG to communicate, co-ordinate and drive interaction with all satellites in the firm. As can be ascertained with a little reflection, this is not something that would take place over night. It would require time and a certain level of maturity to arrive at but yet, the pay offs would be worth the effort. Using this model, many of the standards and best practices reviewed earlier in this paper could be advertised, promoted and guided by the SSG with satellites for each of the different units across the firm taking charge of driving home the implementation for their specific business units.

VII. CONCLUSION

As with almost everything worth having, there is a price to be paid for building secure software. However, more and more, corporations are starting to view insecure software as a business problem rather than as a pure technical problem. This makes sense, because insecure and unreliable software, has the potential to damage a business, either through an intangible vector such as reputation or through a tangible loss such as money stolen from a bank account. Software security, once viewed as a cost center by businesses is increasingly being viewed as an investment. This sea change in opinion and attitude has already resulted in the implementation of many of the best practices described above and yet, security, like technology is a continually evolving field that requires a persistent diligence and adjustment to be continually effective.

REFERENCES

[1] D Yuan, Y Luo, X Zhuang, G. Rodrigues, X. Zhao, Y. Zhang, P. Jain and M Stumm. (2014, 10) Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributes Data-Intensive Systems. Usenix [Online] Available: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

[2] OWASP Top Ten Project. OWASP [Online] Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[3] E. Mougoue (2016, 01) SSDLC 101: What is the Secure Software Development Life Cycle. Citigal. [Online] Available: https://www.cigital.com/blog/what-is-the-secure-software-development-lifecycle

[4] Wikipedia: Dynamic Testing. https://en.wikipedia.org/wiki/Dynamic_testing

[5] Facebook: More details on today's outage. https://www.facebook.com/note.php?note id= 431441338919&id=9445547199&ref=mf.

[6] Summary of the Amazon EC2 and RDS service disruption. http://aws.amazon.com/message /65648/

[7] Dijkstra (1970) "Notes On Structured Programming" (EWD249), Section 3 ("On The Reliability of Mechanisms"), corollary at the end.

[8] BSIMM: Software Security Framework Domains. https://www.bsimm.com/framework

[9] Gary McGraw (2016), "Four Security Findings". Available: https://www.computer.org/computer-magazine/

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)