Global Information Assurance Certification Paper

SECURE SOFTWARE DEVELOPMENT FRAMEWORK: PRINCIPLES AND PRACTICES

GIAC (GSEC) Gold Certification

Author: Michael H. Matthee, michael.h.matthee@gmail.com
Advisor: Richard Carbone

Accepted: 11 November, 2014

Abstract

While larger, more resourceful organizations such as Microsoft's Secure Development Life-cycle may have sufficient funds and resources to develop their own customized security processes and routines, less affluent ones usually do not. Application security controls are best applied in context of the underlying development environment. Different environments demand different objectives; the large disparity between free open-source projects like OpenSSL to commercially motivated endeavors like Microsoft is simply one of many. Although it is generally accepted that unique circumstances demand uniquely tailored solutions, no framework exists within the software development industry to demarcate the principles and practices of application security to unique organizational needs and circumstances. Drawing from new developments in software engineering, a secure application development framework is developed in this paper that is sensitive to the culture, team, stakeholders, infrastructure, and technological characteristics of an organization. The framework enables management to apply software security controls that are fit for purpose, in the right manner, at the right time, and in light of their unique organizational circumstances.

1. Introduction

A common approach is that software delivery is realized through a set of sequential deliverables in a phased and systematic manner. The software process model of the IEEE attempts to bring order to the delivery process by identifying a set of universal artefacts and activities in software construction (Gustafson, Melton, Chen, Baker, & Bieman, 1988). The hypothesis states that the building blocks of this model can be arranged and re-arranged to describe and measure a software development approach practiced within the industry. This assertion is questionable. Scacchi (2001) argues that in the advent of the Internet, open-source software construction, geographically distributed work efforts, and new models of development are arising that are more sensitive to social and organizational circumstances. Great technological shifts have happened since 1988. Yet, the information security community still uses this model to communicate, analyze, measure, and understand secure software practices. The Defense-in-Phase (or Phase) strategy is depicted in Figure 1 below.

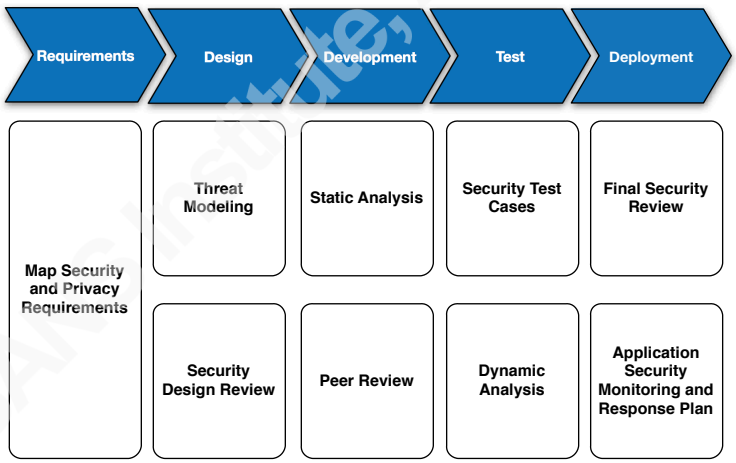

Figure 1: The conventional building blocks of a secure development process. Image was redrawn from Merkow & Breithaupt (2014).

Secure practices such as code review, vulnerability assessments, and requirements analysis are carefully applied during each phase of development. A summary of the Phased approach is given by Merkow & Breithaupt (2014) and are listed below:

Michael Matthee: michael.h.matthee@gmail.com

- Requirements phase: Non-functional requirements that involve application resilience and security are mapped to critical security goals during this phase of construction. The security goals involve the confidentiality and privacy, integrity, availability, non-repudiation, and auditing of information.

- Design phase: Security subject matters are involved during the design and architecture of the system to ensure that good design decisions are made. During this phase the system is decomposed into functional blocks (e.g. using data flow diagrams). This decomposition allows the team to assign and establish a level of trust for each component within the system. Firstly, security threats and their potential impact are identified and ranked for each component. Thereafter, mitigation strategies are drawn up for each threat and then integrated into the design. Examples of threats include spoofing a user's identity, tampering with system data, a user denying that he/she purchased a specific product or elevating his/her privilege levels, an attacker gaining sensitive information such as credit card numbers, or an attacker that executes a denial of service against a software system. Threats can be ranked by their potential damage, how easy it will be to reproduce or launch a specific attack, the number of customers that will be affected by such an attack, and how easy it would be to find a vulnerability that may assist such an attack. A peer review practice may be used to leverage specialist expertise from across the organization to mitigate these potential threats and vulnerabilities within the software design.

- Development phase: A static analysis tool such as Checkmarx (2014) or Sonar (2014) can be used to detect potential code defects, perform stylistic and type checks, as well as perform security vulnerability reviews. Performing code inspection and reviews from time-to-time and writing unit tests help to produce a more robust system that is less prone to exploits. The validation of boundary conditions and the prevention of buffer overflows and underflows are some of the benefits from implementing such secure coding practices.

- Testing phase: Software defects and vulnerabilities that are not discovered during the development phase would hopefully be found by a security testing tiger team. Potential exploits such as SQL injection, cross site scripting, and SSL hijacking are discovered during the testing phase. Automated testing tools such as SoapUI Smartbear (2014) can be used to automate or replay many of the tasks involved. Such an automation practice is also known as dynamic code analysis.

- Deployment phase: The deployment phase involves the coordination between several teams with a strong presence of managerial oversight. Teams from release management, change management, testing, production, and operations engage with one another to ensure that the best possible version is deployed. Ongoing monitoring and periodic testing is conducted to ensure that the software remains resilient in light of changing infrastructure. Infrastructural changes could involve a change in a server's hardware or a software patch to an operating system.

It would be reasonable to think that such an approach will prevent vulnerabilities and produce secure software that is resilient to attack. However, this is only partially true. The truth is that a greater attack surface is at play during software construction.

Whereas secure development practices and activities remain useful, the approach outlined by Merkow & Breithaupt (2014) is constrained to the building blocks of Gustafson, Melton, Chen, Baker, & Bieman (1988). As a consequence, the aforementioned Phased approach fails to:

- Track the progress and maturity toward a secure development environment holistically. For example, it does not reflect the side effects or vulnerabilities of shipping a product with incomplete functionality, neither the use of bleeding edge technologies and/or development platforms. A mechanism is required to trace the maturity of interrelated security assets such as the requirements, technology, team dynamics, stakeholders, and strategic opportunities during an incremental and iterative software delivery process.

Michael Matthee: michael.h.matthee@gmail.com

- Establish greater trust between the producers and consumers of software within a tightly integrated software industry. Consider a vulnerable operating system patch on a specific Android handset or a software dependency on a vulnerable version of OpenSSL. Although a team may be producing secure software internally, credible assurance of third party dependencies remains unknown while confined to the boundaries of the Phased model. A complimentary and vendor neutral security index (similar to CCMI) is needed that ranks an external dependency according to its risk and potential vulnerability profile.

- Focus on establishing and maintaining a loyal, honest and dependable workforce. The Phased approach does not detect and prevent despondent team members from introducing malicious activities or code during the development process. Although control checks may be in place to prevent malicious code from being shipped, in theory, factors such as team synergy, trust, honesty and openness amongst the team members are not addressed explicitly within the model. The Edward Snowden incident, as an example, questions the boundaries of a member's loyalty to secretive programs, personal ethics, and the satisfaction of public interest (Schneier, 2013). Regardless of the verdict, team dynamics play a vital part in forming secure applications that are free from loopholes and prying backdoors.

- Permit the extendibility, flexibility and adaptability of software processes to include future enhancements and the tailoring of it to unique organizational circumstances. The changing application vulnerability landscape mandates that the security framework should be flexible enough to introduce newly developed practices and methods over time in order to remain resilient to new attack vectors (Vinod, Anoop, Firosh, Sachin, Sangit, & Siddharth, 2008). Moreover, security testing and software deployment is best conducted as a continuous activity in order to capture problems as early as possible during the software construction and delivery process.

Michael Matthee: michael.h.matthee@gmail.com

Unlike the Phased model that delineates a software construction process into blocks of engineering activity, the ISO27034 standard (2014) recommends that secure development should be approached more holistically. The software production and consumption eco-system is a tightly knit production line where a vulnerability in one entity affects another. Software companies, outsourced development houses, software toolkits, hardware providers, cloud providers, and supply chains all play a role in producing a safe and secure software product in the end. A number of "secure" development processes claim to address these security concerns. However, none of them frame the essence of secure development principles and practices for general applicability, nor do they fully address application security from a holistic perspective. Two secure development processes that are in common use today are outlined below:

- The OWASP Comprehensive, Lightweight Application Security Process (CLASP) lists a collection of 24 security-related practices that may be reused in a different context. The CLASP methodology has a heavy reliance upon team organization and roles of responsibility (Information Resources Management Association, 2013). The methodology consists of several perspectives:

  - The Concepts perspective sets its focus on the authorization, confidentiality, authentication, availability, accountability, and non-repudiation of organizational resources. This is accomplished by spurring on a cultural affinity toward secure principles and practices within the team through the induction of awareness programs and the monitoring of security metrics; the implementation of practices through the capturing of security requirements, implementing them and constructing vulnerability remediation procedures; and guiding those practices by publishing operational security guidelines.

  - The Role-based perspective assigns roles and responsibility to the various team members. Whereas designers, architects and project managers are

trained with overall security in mind, developers need only focus on the intricacies of secure coding by following guidelines, policies and standards.

- The Activity-assessment perspective is a collection of activities (or security controls) to be implemented during the project endeavor. Each control is mapped and assigned to one or more roles defined in the Role-based perspective.

- The Activity-implementation and vulnerability perspectives detail the intricacies of implementing the various security controls and categorizing anticipated risk exposure, respectively.

Although the CLASP methodology seems to cover most areas within a development environment, it is not always contextually relevant (Win, Scandariato, Buyens, Grégoire, & Joosen, 2007). The activities and practices of CLASP can be successfully reused in medium-to-large organizations and be integrated into existing processes that are mature, plan-driven and systematic. However, difficulties arise when attempting to integrate the principles and practices of CLASP to existing agile processes like eXtreme Programming (Win, Scandariato, Buyens, Grégoire, & Joosen, 2007).

- The Microsoft Secure Software Development Lifecycle (or SSDL) claims to be the first process to comply with ISO27034 prerequisites (Microsoft, 2010). According to the Information Resources Management Association (2013) Microsoft SSDL's guiding principles are to:

  - Produce secure software designs using the practices listed under the Phased approach.

  - Assume that something is not required until asked for by applying the principle of least privilege, avoiding risky default changes and enforcing strong security controls by default.

- Perform secure deployments by producing deployment guides for others to follow and by installing analysis and patch management tools to continuously monitor events on a production system.

- Outline incident response procedures by engaging the consumer community.

In summary many of the practices established by the Microsoft SSDL are simply elaborations and improvements to the guidelines set out by the Phased approach previously mentioned. However, Microsoft does make the process more flexible by defining a collection of security activities that are interchangeable across the entire project endeavor. Microsoft admits that its process is not a solution for all environmental characteristics and situations (Microsoft, 2010). Instead, it advocates that its process contains a set of practices that can be reused in different contexts if desired. Whereas CLASP addresses security from a much broader perspective, Microsoft SSDL is narrowly focused on a set of practices while some parts of the process are characterized as guidelines rather than specific process activities (Win, Scandariato, Buyens, Grégoire, & Joosen, 2007). This makes practical applicability within specific situations difficult as no framework exists to port the principles and practices of Microsoft SSDL to other project endeavors.

Both CLASP and Microsoft SSDL have limited means to make quantitative measurements and they both provide low visibility on any improvements made while in use (Win, Scandariato, Buyens, Grégoire, & Joosen, 2007). Moreover, both processes lack contextual relevance for distinct and unique organizational needs. According to the United States Department of Homeland Security, nearly 50% of all traditional software security assurance activities are not compatible with Agile methods while less than 10% are natural fits to the Agile cause (Noopur, 2014). A framework is needed that would encourage and enforce secure software practices irrespective of the selected development approach or contextual nuances.

Michael Matthee: michael.h.matthee@gmail.com

Instead of advocating a set formula for achieving information security in an organization, the ISO27001 standard suggests that a tailored solution be used in order to satisfy the unique needs of each and every organizational environment (ISO/IEC 27001, 2013). Unfortunately, the standard does not indicate how such tailoring is to occur, nor does it mention mechanisms that may be used to analyze the development environment analytically. A recent webcast held by SANS reiterated this concern (SANS, 2014). From this discussion, it was understood that developmental context plays a crucial role in mitigating software vulnerabilities such as the OpenSSL Heartbleed attack from re-occurring.

Is there a way to classify an organizational environment categorically? Moreover, can such a classification be used to tailor a software development process to deliver the right security controls at the right time?

Instead of advocating a silver bullet in answer to security breaches within applications, a set of reusable principles and practices (collectively called a software security kernel) is suggested that works for any working environment and enables management to tailor a solution to their unique needs and situation. Furthermore, situational characteristics are identified and are tied to the possible styles of applying security controls within an organization. Together, the principles and practices of the security kernel forms a Defense-in-Depth strategy that can be used to mitigate incidents in an ever-evolving application vulnerability landscape.

2. Secure principles and practices: forming the software security kernel

According to dictionary.com (Dictionary.com, 2014):

> A *principle* is a general and fundamental truth that may be used in deciding conduct or choice: *to adhere to principle.*

Whereas principles guide our actions, a practice puts them into action:

> A *practice* is the act of doing something: *he put his plans into practice* (Dictionary.com, 2014)

Together, principles and practices form decisions that people make on a daily basis. Software practices range from the rigor of highly structured discipline-based approaches to others that are more creative and affluent in nature. Empirical and creative security practices involve exploration and experimentation, such as TDD, penetration testing and malware analysis. Such practices are useful whenever a high degree of uncertainty, ambiguity or obscurity is present within a work endeavor. These practices are classified as *crafted* practices. Security practices that commonly demand more discipline and managerial oversight are well-defined tasks with step-wise enforcement procedures, such as executing a disaster recovery plan or a business continuity plan. These practices are classified as *plan-driven* practices. The selection of one practice over another is driven by factors both inside and outside of a team. Whether the end goal is a speedy delivery of a website launch or the meticulous planning of a successful moon landing, different objectives demand different approaches to achieving them. During a project endeavor security controls may be applied in order to align engineering practices to security principles and objectives, such as the confidentiality, integrity and availability of information.

The term security kernel is used to highlight the maturing aspect of security within an organization. Some of the practices that are good today are not necessarily the best option for tomorrow. As Lee Copeland (2014) put it:

> There are no best practices - there are, however, good practices in specific contexts.

By employing a specific practice during a software project endeavor, management can influence its successful outcome. An obstacle is knowing when and how a particular practice should be employed.

2.1. Employing security practices: controlled or crafted?

According to Myburgh (2014) software engineering should be viewed as a complex adaptive system. Whereas other engineering disciplines have clearly defined steps and outcomes (e.g. the laws of Newton that remain consistent over time), software engineering is somewhat more idiosyncratic in nature with a high degree of additional complexity. This additional complexity is attributed to software's dependence upon the ingenuity of people and their interactions with one another.

Two primary forces are present in software engineering as a complex adaptive system. The first force is the strategy for producing the system (a result of selected engineering practices). The second force is the manner in which controls are enforced (a consequence of the selected management style).

From an information security perspective, application security controls are enabled by secure software development practices. The security practice can either be plan-driven or crafted. This forms the first force within the situational applicability model of Myburgh (2014).

Management and control processes together form a second force within software engineering. Management style can be categorized as either formal or informal. A formal management style is followed when management maintains close supervision over production processes. In this situation, management seeks detailed visibility within the development process, which can only be attained through close monitoring and planning of the software construction process. An informal management style means that management is less involved in the production of software and relinquishes discretionary power to others working in the field (e.g. a technical lead or an architect that may drive the technical architecture and design of a software system). In such instances, technical

staff has greater autonomy in their *way of working* while management is less involved in the finer details.

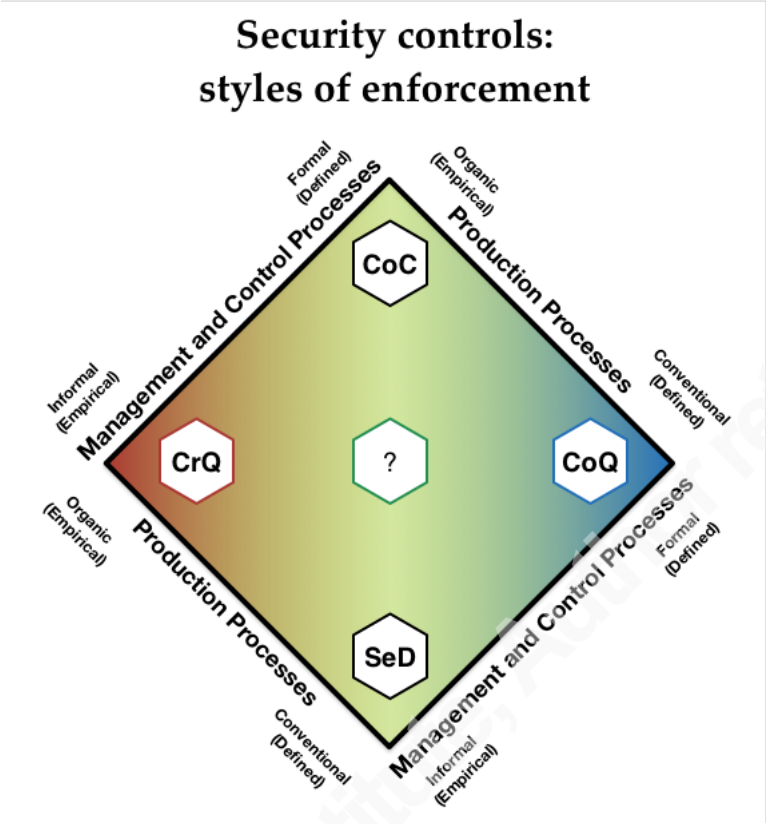

Figure 2: Ways to control and enforce security during software development. Image was adapted from the situational process model (Myburgh, Towards Understanding The Relationship Between Process Capability And Enterprise Flexibility, 2009). The left corner is colored red to denote a high degree (or frequency) of reflection while the color blue (on the right) correlates with slowly changing and systematic regulatory demands.

However, making the impact of engineering practices visible (as in Section 2.4) provides management with greater visibility in the defensive posture of a specific development environment and the software that is being produced within it. This enables management to make decisions that are more informed. Sometimes more managerial oversight (or formal management) is required to achieve and maintain security compliance. Compliance defines the minimum standard for ensuring that a secure development environment is formed. Other times, however, it may be beneficial for management to stand back (informal management) in order to allow the team to push the conventional boundaries of application security to new frontiers.

Michael Matthee: michael.h.matthee@gmail.com

The combination of the chosen management style and the selected engineering practices form four distinct areas of operation as depicted in Figure 2. The combination of informal management with organic engineering, which predominantly involves the use of crafted practices, forms the crafted quality (CrQ) domain. A formal management style combined with a well-defined software development effort and a dominance of plan-driven practices forms the controlled quality (CoQ) domain. This range of CrQ to CoQ domains forms a band of software engineering best practices (Myburgh, 2009). From Figure 2, this range of good engineering practices would constitute the horizontal region between the CrQ domain on the left and the CoQ domain on the right.

Two other combinations exist within this model, the controlled costs domain (CoC) and the self-directed domain (SeD). The CoC domain is the result of voracious demands by management on practices that require sufficient experimentation, exploration and innovation to be completed successfully. An example would be the strict enforcement of testing procedures, methods and steps during a penetration testing exercise. The combination of crafted security practices (like penetration testing) with formalized expectations (e.g. the number of vulnerabilities or exploits that should be found) is likely to produce inconclusive results. This may foster politicking, cover-ups and blame shifting within the development environment when things go wrong (Myburgh, 2009). It would be more effective to encourage a culture that propels security awareness and reward crafted practices such as defensive coding and penetration testing using an informal management style. Informal rewards could be employed through the instigation of employee prestige, team celebration and honorary respect. This stands in contrast to an enforcement style that may utilize strict schedules and procedures while mandating pre-determined outcomes.

At the other end of the spectrum (bottom corner of Figure 2) resides another undesired enforcement style. The self-directed domain (SeD) is the result of little managerial oversight against well-defined tasks and procedures. An example of such step-wise security practices is the planning and execution of a disaster recovery plan or even a

business continuity plan. Consider the following emergency: a tornado hits the premises. In preparation for such an incident, it is important that all software repositories are backed-up to a safe and secure remote facility. Furthermore, attention should be taken not to leave sensitive software assets (such as requirements, blueprints or design documents) while evacuating the facility. Failure to do so may compromise the defensive posture of the software product. If employees are left to their own devices (and not supervised during the evacuation procedure), they are likely to succumb to their primeval instincts and overlook the protection of sensitive information. This would personify a self-directed (SeD) enforcement style where workers are relied upon to save the day while management secedes responsibility for any failure. During emergencies, it is better not to succumb to emotional decision-making processes. In such situations, a well thought-out and planned checklist procedure that is enforced in a controlled manner (CoQ) would be more effective.

Security risk within a software product is the product of potential threats and the vulnerabilities that it may expose. These threats and vulnerabilities are not restricted to coding practices alone, but may be introduced via its environment, sometimes unbeknownst to its creators. Thankfully, information security controls exist to mitigate such security risks and their potential attacks against the confidentiality, integrity and availability of software products and the information that it holds.

Determining when and how a particular security control should be applied is depended upon the context of the environment as well as the risk exposure of the software artifact that is to be secured.

2.2. A development environment in context

In essence security controls are not limited to the engineering intricacies of software construction (e.g. addressing cross site scripting and SQL injection vulnerabilities) or managerial governance (e.g. a risk management or a business continuity plan), but are also subject to unique situational characteristics that are present during the development effort. For example, the development environment (whether cubicles or war rooms are

used), the supporting tools or build systems that are used, the software's intended purpose and function, and the stakeholders that are involved are all situational elements that needs to be accounted for during the enforcement of security controls. Failure to do so may jeopardize the successful outcome of a project. An effective software development process (SDP) should enable management to enforce software security controls at the right time and in the right manner. It is therefore important that a project's situational characteristics (such as the developer's working environment) are adequately understood during the software development life-cycle (SDLC).

From a high-level perspective, Boehm & Turner (2003) outline five environmental variables one could use to capture the unique situational characteristics of a specific software project endeavor. These variables – as viewed from an information security perspective - are examined below:

- Size: The size of a project determines how much management and control is required during the software delivery process. A large software project endeavor (like implementing the software control systems of a Mars expedition) requires a significant amount of coordination and well-documented communication channels. Consequently, interception of communications (e.g. corporate or nation state espionage) or the spreading of misinformation (e.g. the malicious actions of delinquent employees) becomes easier and the risk of compromise is higher. A large-sized project would therefore require additional plan-driven practices in order to remain secure. A smaller project, like the construction of a small web site, can do away with most of the heaviness involved in stringent control and planning measures.

- Criticality: The sensitivity and purpose of a particular software product determines how critical its various deliverables are to the utility, brand and reputation of an organization. For example, the exact, precise and secure implementation of control software for a nuclear power facility is vital. Not only does an imprecise implementation run the risk of system failure, moreover, the effects of an exploited system by terrorist organizations would be detrimental. For

example, the software of critical infrastructure in a nation-state (such as power plants and telecommunications) demands more plan-driven practices. In contrast, the compromise of a personal blog would not be devastating to the general population.

- Personnel: The demands of creative and innovative software production require the involvement of more talented and skilled individuals. A well-defined working environment with systematic work routines and procedures can succeed with less skilled employees using plan-driven practices, for example factory workers that perform routine packing tasks. In contrast, a creative and innovative creation demands well-trained and highly skilled individuals with greater degrees of autonomy. For example, an unskilled and inexperienced workforce tasked with the design and development of a complex and mathematically inextricable cryptographic function would likely produce a solution that is error-prone, vulnerable and insecure.

- Dynamism: Detailed planning and big-design up front would work best in a stable environment where requirements are less likely to change. A highly dynamic environment that requires rapid response and feedback to implementation details is best met with greater flexibility and adaptability. For example, constructing a new software product requires more adaptability and change than maintaining an old code-base. Constraining an upstarting development exercise to tightly, fixed plans and procedures may inhibit the team's ability to be ingenious and build in new, innovative and sound protection mechanisms.

- Culture: According to Dyer (2013) a culture is not only constrained to geographical regions like continents or countries, but is formed whenever artifacts, norms, values and assumptions are shared collectively within a group. A team culture that prefers clear policies and procedures is a team that thrives on order (Boehm & Turner, 2003). Such a team would thrive when more systematic and plan-driven practices are implemented. This is synonymous to a production-

line environment where each member's tasks are well defined and pre-determined. In contrast, a team culture that prefers to be empowered by many degrees of freedom is a team that prefers to thrive on greater levels of autonomy (Boehm & Turner, 2003). Such a team would likely appreciate the presence of more crafted and creative practices that lend themselves to greater liberties for innovation and industriousness. For example, the efficiency of researchers at a research institution would be stifled if their creative freedoms were inhibited through bureaucratic task lists. Applying security controls in a way that is out of touch with the team's culture and makeup is likely to produce despondent team members that could foster malicious activity and behavior within the organization.

Together, these five environmental factors characterize and contextualize a specific environment. In some situations, a greater measure of systematic, planned and well-defined steps, tasks and activities would be appropriate to achieve security compliance. In other situations, the incubation of creative, innovative and fresh ideas would be more useful in attaining excellence.

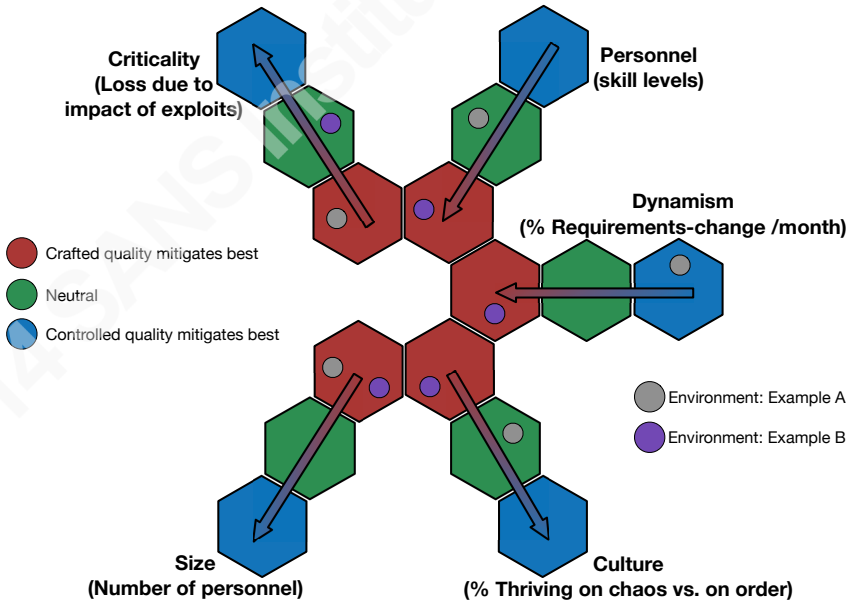

Figure 3: Five critical factors that indicate whether more plan-driven or more crafted practices are appropriate to mitigate software security risk. The concepts were adapted from (Boehm & Turner, 2003) for information security purposes.

Michael Matthee: michael.h.matthee@gmail.com

By analyzing the development environment, a manager can approximate how many plan-driven or crafted practices are needed within a particular setting. Different environmental conditions demand different control mechanisms. Within Figure 3, these distinctive environmental characteristics (size, culture, dynamism, personnel and criticality) are depicted against the two categories of software security control (CrQ and CoQ). Two fictitious environments are also depicted in Figure 3 in order to highlight the impact different situational elements have upon effective software security controls.

Consider the first example (indicated as environment A in Figure 3). A few friends have decided to launch a new company and their team has just been formed. The software product is a simple company website hosted off-site. The requirements of the web site are well understood and agreed upon by all the team members. An initial environmental analysis for this scenario concludes the following results:

- It is a small project endeavor with few team members. From a security standpoint, the number of communication points is few. Therefore, the use of predominantly crafted practices is feasible.

- The team is still in the formation stage and its culture is still forming. As such, it is still too early to determine if the team culture favors autonomy or systematic procedures. From a cultural perspective, neither plan-driven nor crafted security practices would negatively affect the team's security posture. Later on, however, the team could evolve to attain either a thriving plan-driven or a thriving craft-driven culture.

- The engineering team is encountering a low frequency of requirements changes within the project endeavor. Therefore, the risk of work efforts becoming obsolete is low. This permits the team to create detailed work breakdown charts and conduct big upfront designs sparingly. Having well-defined plans and consistent

ways of working produce more resilient architectural designs and software. This characteristic would encourage the use of more plan-driven practices.

- The team consists of both highly skilled employees as well as more junior, less experienced workers. While junior developers may find a place doing smaller, more mundane tasks, experienced professionals could be innovating new products, designs and solutions. The hybrid nature of the team make-up allows it to operate well using both plan-driven and crafted practices.

- The software solution produced in this case is a company website that is hosted off-site. Although deformations to the site could tarnish the company image and brand name, such an incident would not necessarily be catastrophic. As a remedial measure, the web site can be taken offline for a lock-down procedure without adversely affecting the core business. An excessive investment in securing this website is best invested elsewhere, on more critical software products produced by the company. For that reason, the low criticality aspect for launching the company website does not necessitate excessive penetration testing and vulnerability analysis exercises. A basic security conscious implementation would be sufficient in this instance.

In this example environment, an environmental analysis indicates that a greater proportion of crafted development practices should be employed. Crafted practices are best enforced using an informal management style. It is therefore recommended that management employ a more crafted quality (CrQ) response overall in order to mitigate software security risk for this project endeavor. In contrast, a different setting may warrant routines that are more systematic. Such an environment would benefit from a greater proportion of plan-driven practices and hence a greater measure of CoQ.

2.3. Risks in selecting a particular set of practices

While environmental characteristics help to determine the best way for employing and managing security practices, which practices should one use? Furthermore, are there any risks involved when selecting one particular set of security practices over another?

A set of engineering practices are often packaged together and then sold as a popular development process. Every few years a new development process is created with devoted followers, often claiming it to be the silver bullet for project success. Whatever the devised process is, it would reside somewhere between either a dominant plan-driven or a dominant crafted or "agile" spectrum (Boehm & Turner, 2003). Plan-driven approaches are more meticulous, systematic, structured and planned. Examples of plan-driven methodologies are: Cleanroom, Personal Software Process (PSP), Team Software Process (TSP) and the Waterfall model. These methods run the danger of falling into debilitating bureaucracy (Myburgh, 2010) or stifled creativity (Vliet, 2008) if not employed in the right circumstances. In contrast, agile-driven approaches are more experimental, self-reflective and adjusting in nature. Examples of such methods include: eXtreme Programming (XP), Adaptive Software Development, Crystal, Scrum and Feature-driven Development. These approaches may fall into controlled chaos and/or politicking if not employed in the right circumstances (Myburgh, 2010).

Sometimes the discipline and tight control of a plan-driven development strategy would be better suited than an agile approach. The United States military favors plan-driven approaches and standards such as DoD-STD-2167 (a document-driven standard for defining the data item descriptions of deliverables), MIL-STD-499B (which defines the contents of a systems engineering plan) and CMMI (which integrates software and systems engineering capability maturity models) for their structured precision (Boehm & Turner, 2003).

Consider eXtreme Programming (XP), a light-weight agile methodology. XP seeks to find the simplest technical solution for a problem tries to anticipate requirement changes and encourages continuous experimentation through the rapid deployment of software for

customer feedback. Experimenting to see if a software product works in a nuclear facility would prove fatal, tragic and cause unnecessary human casualties. In the nuclear facility's context a plan-driven approach would fit better. However, using XP to produce a small website may be very valuable since it's a quick, creative way to get the team to capitalize on early time-to-market opportunities. In the latter case, the business value of aligning development efforts to economic demands far outweigh the financial risks of software vulnerability and compromise. In the former case, the security risks are fatal, therefore demanding greater control mechanisms using a more plan-driven approach is necessary.

In summary, practices should not be constrained to a fixed and pre-determined list for the sake of conformity. Different organizations require different practices, methods and ways of doing things in order to produce secure software products. If the right practices are not employed at the right time, vulnerable software solutions might be constructed and then be shipped to customers. A secure software engineering framework is needed that can be tuned to the unique needs and circumstances of an organization.

2.4. Building blocks of a secure software framework

Instead of applying a set formula for software development blindly, adequate visibility during the construction process is required to ensure that the software produced is in fact safe to use and free from vulnerabilities. Three building blocks ensure that a secure software product is delivered, namely the customer's involvement, the solution that is provided and the manner in which the solution is provided (involving resources, a team and infrastructure). In turn, a number of metrics can be associated with each building block. The metrics defined by Jacobson, Ng, McMahon, Spence, & Lidman (2013) aims to establish maturity in the software development community with regards to the selection and adoption of new or existing software engineering methodologies and practices..

These metrics are depicted graphically in Figure 4. The *team* performs and plans some *work* and completes it through a set *way of working*. The *work* satisfies a set of

requirements needed by *stakeholders* and forms a *software system* when implemented. In turn, the software product (or system) capitalizes on some business *opportunity* enabled by the *stakeholders*. Together, the team, work, way of working, requirements, software system, opportunity and stakeholders all play a part in producing a secure software product. Overlooking any one of these assets when attempting to produce a secure software product is sub-optimal; communication is key. Consider a nuclear development program. If a resourceful team member were to be captured by a terrorist organization, he or she might leak sensitive information, or vulnerabilities of the software product, during interrogation. The consequences of such an information leak in the *team* aspect (e.g. leaking information about a SCADA system that controls a nuclear enrichment process) could be dire. In 2010 a similar incident occurred in Iran when the Stuxnet worm (2014) was introduced to Iran's nuclear facility through a USB flash disk. The worm contained a number of zero-day exploits that were targeted specifically and precisely for Iran's computer systems. The number of zero-day exploits used was unusually lavish, suggesting that team members from software vendors could have participated in the attack. On the contrary, however, if the *way of working* in constructing the system does not involve security-testing exercises then the product may be prone to vulnerable exploits.

The objective of the software security framework is to incrementally increase the reliability, safety, dependability and security of each building block. Tracking the maturity of these building blocks enables management to establish a secure, trusting and dependable working environment.

Figure 5 depicts high-level activities during a project endeavor that should mature over time. The states depicted are called activity spaces since they represent a group of concrete and practical activities that may be pursued by the team. The first activity space within the solution domain is called "understand the requirements". This activity space can be sub-divided into a number of sub-activities to make it more concrete. One such sub-activity could be to understand the requirements of a mobile testing tool. In turn, activities involve concrete decisions and actions on the floor. These activities involve a

number of practices that could increase the confidence in the software security endeavor and its supporting environment. Referring to the requirements of a mobile testing tool again, a set of requirement metrics for it may be:

- R1: The tool should expose the mobile application as a consumable service.
- R2: The tool should be non-invasive and require no changes to the application itself.
- R3: The tool should be platform independent.

Each of these requirements has an associated state of maturity, moving from conception (as the first level of maturity) to fulfillment (as the last level of maturity). The maturity process for all software metrics is shown in Figure 6. In order to accomplish the maturing process, each software artifact may be associated with a set of security controls and tasks (such as vulnerability analysis, penetration testing, code reviews and test driven development). More importantly, the maturity of one artifact (such as requirement R1 from above) is dependent on the maturity of other artifacts (such as the team, stakeholder or opportunity assets). Referring to requirement asset R1, the following additional assets may improve its maturity:

- *Stakeholder* involvement: The potential security risks of the exposed service can be communicated to the stakeholders involved. From such a discussion, the necessity for exposing the service publicly can be reconsidered, therefore minimizing its attack surface. Misuse cases of the service can also be identified and refined into the set of requirement assets.

- *Customer* involvement: The consumers may not be ready to consume and use the mobile testing tool appropriately. The consumers should be trained and prepared in the correct usage of the tool. If used incorrectly, the mobile testing tool could inadvertently expose sensitive information to the outside world and become a liability instead.

- *Team* involvement: The team might still be forming and not be collaborating well enough to build a new, innovative and groundbreaking mobile testing tool. Additional team building exercises could lift morale, improve collaboration and increase overall trust levels in the organization.

Whereas a given requirement may be *acceptable* for implementation, the team could still be in a *formed* state while stakeholder involvement is suspended in the *represented* state. The underdevelopment of the latter may restrain the *integrity*, *availability* and *non-repudiation* of the exposed service (requirement asset R1).

Each software security metric (like the *team* or *way of working*) can be associated with a number of tickets or tasks to complete. Together, the progression of the security tasks and activities can be displayed on a big screen to foster effective team communication, mutual understanding and team synergy. One popular tool for displaying the project status to team members during team gatherings is the JIRA project management tool from Atlassian (Atlassian, 2014).

In summary, continuous measurement of the entire project environment makes software production efforts more visible and traceable. As a result, the team is made aware of fragilities within the larger project environment that may foster security vulnerabilities and defects in software.

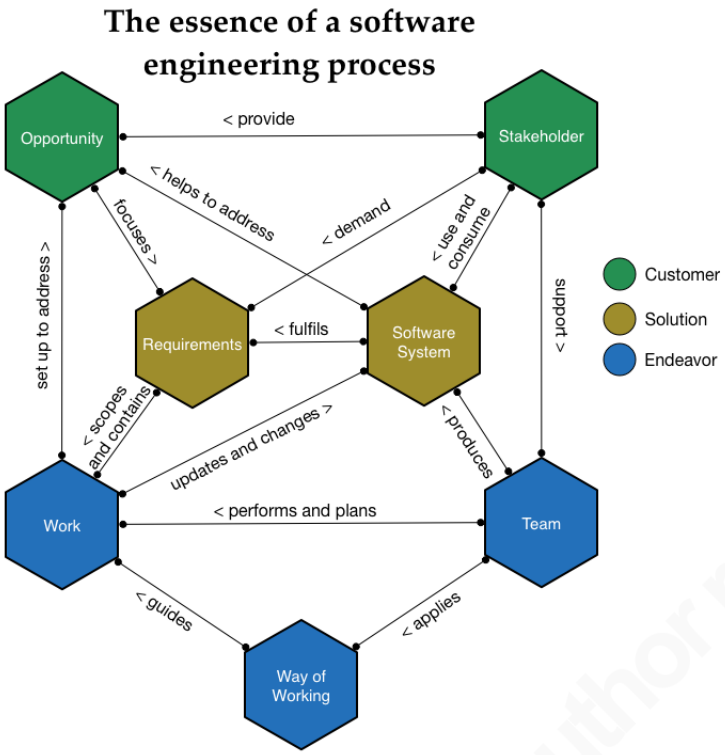

Figure 4: Building blocks of a software development process. Redrawn from (Jacobson, Ng, McMahon, Spence, & Lidman, 2013).

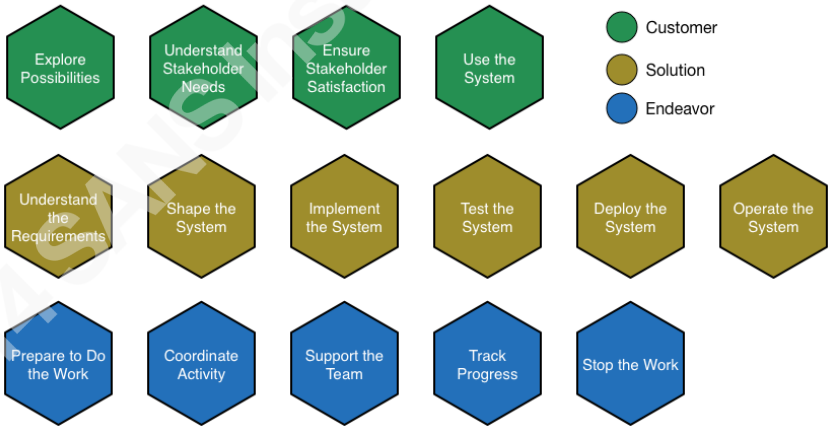

Figure 5: Depiction of high-level activity spaces in order of increasing maturity from left to right. Redrawn from (Jacobson, Ng, McMahon, Spence, & Lidman, 2013).

Michael Matthee: michael.h.matthee@gmail.com

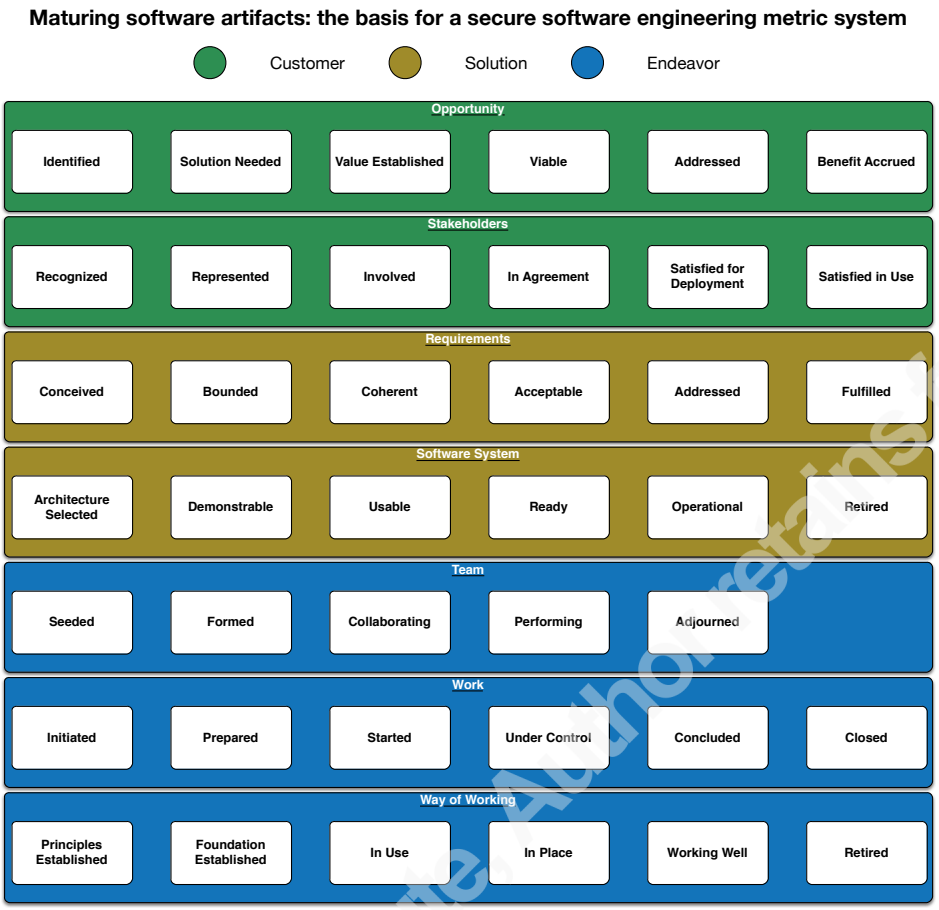

Figure 6: Available states for software metrics (or alphas) arranged in increasing levels of maturity from left to right. Redrawn from (Jacobson, Ng, McMahon, Spence, & Lidman, 2013).

2.5. How do existing processes fit in to this framework?

Every software development process and methodology consists of a selection of engineering practices and methods. While one process may value the use of a particular practice, another could regard it superfluous at best. A set of practices that works well for one delivery team does not necessarily translate well to another. There is no silver bullet for the successful production of secure software. Environmental characteristics like team structure, culture and operation; project size, criticality and make-up; as well as stakeholder involvement, business opportunities, and the nature of requirements all play a part in the delivery of secure software. Rather than enforce a uniform way of working across all software project endeavors, a process and methodology is best tailored to the unique circumstances of each situation. According to the ISO27034 standard, the

tailoring process should be conducted from a holistic perspective and include factors that are both external and internal to the organization (ISO/IEC 27034, 2014). The tailoring can be conducted by first performing an analysis of the environment (outlined in Section 2.2) where after a set of practices are selected in line with the business needs and priorities. The software security assets defined in Section 2.4 serve as a measurable criterion for such a tailoring exercise. If the process is contextually relevant then the organization's software security assets will mature collectively and in tandem with one another as envisioned by Jacobson (2013).

Unfortunately, popular software development processes do not secure the underlying building blocks of a software engineering endeavor very well. The building blocks for three software development processes are illustrated in Figure 7 below.

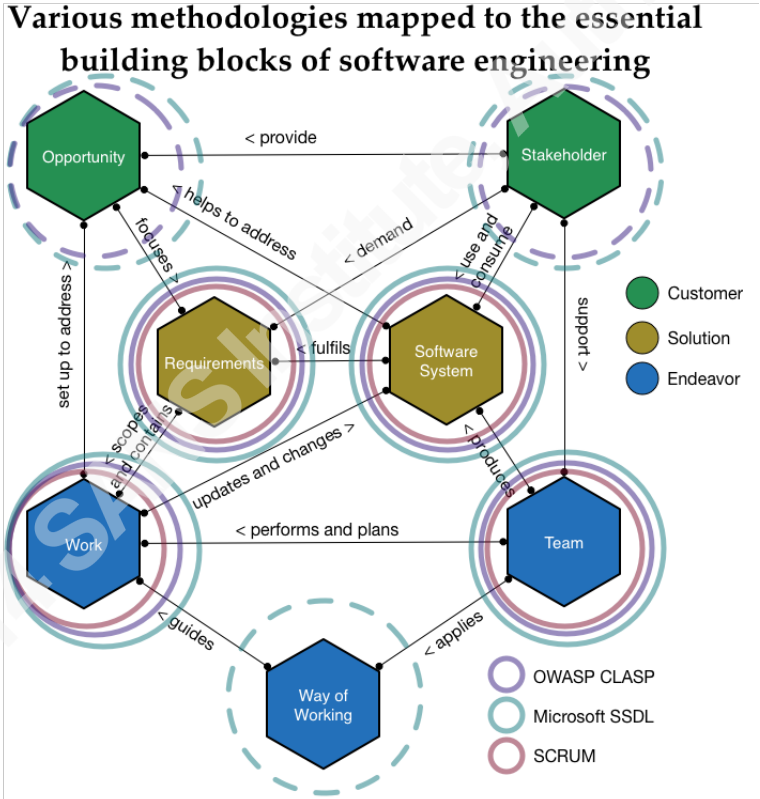

Figure 7: Mapping various methodologies onto the building blocks of secure software engineering. Areas that are covered somewhat by a process but not intently are encircled with dotted lines.

Michael Matthee: michael.h.matthee@gmail.com

According to Jacobson (2012) the SCRUM methodology only addresses four out of the seven software engineering building blocks of Section 2.4. The four areas that are covered are the *requirements* that are defined, the *software system* that is built, and the *team* that conducts the *work*. These asset areas are marked with red circles in Figure 9. The defensive posture of these assets is critical to the formation of secure software constructs. Supporting practices such as the way of working (which includes static and dynamic analysis, code inspections and continuous integration) is crucial to developing secure code. Ironically, although such practices are often perceived to be reminiscent of an "Agile" process, neither the Agile manifesto (agilemanifesto, 2014) nor the official SCRUM guide (Scrum.org, 2014) mentions or encourages the use of supporting production processes such as continuous integration or code reviews. Nevertheless, a strict and rigid implementation of the SCRUM methodology cannot guarantee the delivery of secure software, because its set of practices fail to focus on all the essential building blocks of a software engineering endeavor.

The Microsoft SSDL methodology relies heavily upon threat modeling. According to the Information Resources Management Association (2013), this may be due to the fact that a large percentage of flaws in Microsoft's software have been design related in the past. The methodology secures the *software system* through the promotion of a secure architecture and software design. It also addresses some aspects of the *team* aspect by mandating that members undergo core training in secure development practices, assigning team roles and responsibilities. The *requirements*, *opportunities* and *stakeholder* aspects are addressed, although mostly from a vulnerability and threats perspective. The *way of working* aspect is covered by the use of static and dynamic code analysis as well as an optional manual code review. Altogether, the Microsoft SSDL process secures every essential building block, albeit at a lower intensity for some. However, the threat analysis within the *opportunity* and *stakeholder* domains could be made more explicit in this process. Such a clarification would allow management to track the progress toward maturity with greater clarity. According to Erwin, Magnuson, Parsons, & Tadjdeh (2014) people and relationships are central to security. Instead of

merely detecting threats within the *opportunity* and *stakeholder* domains, fostering greater levels of trust and collaboration in these relationships would be more proactive.

Similar to Microsoft's secure process the OWASP CLASP methodology also covers most areas from a threat and vulnerability perspective. It has a strong focus on *team* roles and responsibilities. It also has a good focus on *requirements* and coding guidelines on the *work* that is produced. However, very little is mentioned concerning the *way of working*. The *way of working* involves concerns such as the security of the code repository, the selected code editors and tools, continuous integration as well as branching and merging policies.

The software development processes analyzed above (two of which claim to focus intently on the delivery of secure software) reveal areas for improvement. This questions their appropriateness and efficacy as potential silver bullets in the software industry. It would be more sensible to conclude that the delivery of secure software can only be achieved through the amalgamation of selected principles and practices that are tailored for a particular environment and its priorities.

2.6. Tailor and manage your own secure process

Metrics are cornerstone to effective management. Harrington (1991) puts it succinctly: "If you cannot measure it, you cannot control it. If you cannot control it, you cannot manage it. If you cannot manage it, you cannot improve it."

The critical information assets of a software product can only be secured if they are measured accurately and matured appropriately. Accurate measurements are attained from a suitable metric system and capturing one's progress continuously. The software security framework developed previously can be used as a metric system. The maturation process is achieved through the application of security controls and tasks, bearing in mind the implications of enforcement style and methods for exercising such controls. Decisions frame how and when these controls are applied. These decisions not only impact the resulting quality and security of the software products being produced, but also the entire working environment. A system of systems engineering perspective of such decisions is

depicted in Figure 8. In this perspective, decisions are categorized into eight levels of abstraction, ranging from the visionary and strategic to the more tangible and concrete.

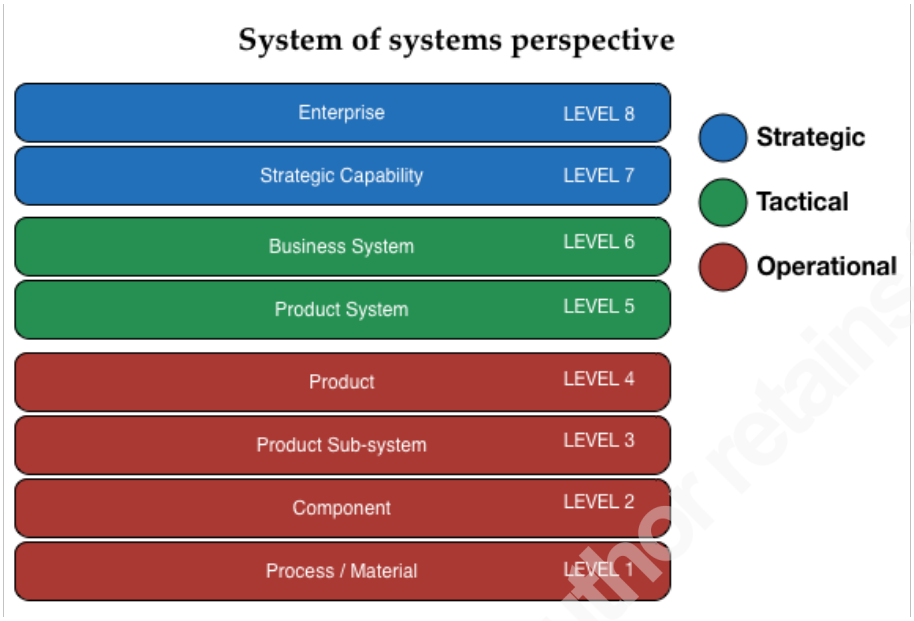

Figure 8: A system of systems perspective on an organization. The color blue is used to depict a slower, more strategic, decision making process (low frequency), while the color red is used to indicate a more rapid decision making process (high frequency). While strategic decisions may take a long-term view, day-to-day operational decisions would occur in rapid succession as the need arise.

Decisions are enacted through a set of controls. Quality controls (including security controls) can be enforced in two ways. They can either be enforced using well-defined plans and formalized ways of working (controlled quality), or using crafted quality that is highly adaptive, responsive and lends more freedom to tactical improvisations (Myburgh, 2009).

Decisions at the top of the hierarchy are strategic in nature. Such decisions would typically involve senior executives of the organization. A control quality approach would be better suited for these decisions, as long-term planning and strategic intent is emphasized. In contrast, decisions at the bottom level require rapid response and action from employees. A mixture of control quality and crafted quality would be better suited since quick turnaround times are emphasized.

For example, assume that a traditional mail-based post office is to be secured and defended from cyber-attack. In light of the predominance of e-mail and the emergence of Internet cafés, the post office may advocate that more technology should be introduced within its service offering. From a system-of-systems perspective, the enterprise (see Level 8 in Figure 8) would designate the post office as a business. Variables that would influence business decisions on this level include market base (or customers), profit, turnover, intellectual property and asset specificity (for example, geographical location, specialist practitioners, patents and confidential research for strategic purposes). In turn, the post office, as an enterprise, cannot exist without a number of strategic units of capability (indicated as Level 7 in Figure 8). Strategic capabilities allow the post office to differentiate itself from its competitors both in the short and long term. One such capability may be the use of information technology infrastructure to establish an integrated e-business service; thus providing a hybrid service between a physical post and a digital world. To accomplish this, various business systems (marked as Level 6 in Figure 8) are needed to establish and further operate the e-business as a unit. One such unit may be electronic messaging, such as fax, email and hybrid mail. In turn, the product system (see Level 5 in Figure 8) details how management, policies and procedures, facilities and employees would enable electronic messaging to take place. A product system may contain a number of Level 4 based products, such as a router, server, firewall, intrusion detection system, intrusion prevention system and/or a printer. A firewall consists of a number of Level 3 based product sub-systems (such as a network monitor), which in turn would comprise a number of Level 2 software components (such as software packages or libraries) and the Level 1 processes or materials that created them.

The higher-level decisions are more influential and take a longer time to realize. These decisions form the company's overall vision, strategy and policies. For example, deciding to move an organization over to IPv6 is a big decision that requires many changes, updates and testing. Such a decision is strategic and consumes a considerable amount of time and resources to implement correctly. Referring to the software delivery building blocks of Section 2.4, this decision would involve a change in the organization's *way of*

working. An incorrect or an incomplete implementation of IPv6 could expose the organization to unnecessary risk. In turn, an exploit in this area could compromise software products that are produced for customers as well as any software products that are consumed internally. Likewise, envisioning a new software product as a strategic decision for the company will affect its long-term profitability. From Section 2.4, this decision would involve the *opportunity*, *stakeholder* and *team* assets. During this stage, initial proposals and designs are drawn up. A compromise during this stage of the project endeavor would have long-term consequences and require a greater amount of control quality to ensure its success. Another example where the discipline of a controlled quality approach is more appropriate is when forming a new disaster recovery plan. Such a plan requires meticulous planning which should not be developed spontaneously in the spur of the moment.

Lower levels within the system-of-systems view demand a more rapid and quick decision-making process. These decisions are more inclined to daily operations. One example would be the investigation of an incident alert flagged by an intrusion detection system. The investigation is performed by a technically inclined individual, lower down in the organizational hierarchy. In order to handle the incident effectively quick and rapid response remediation is required. There is no time to draw up documents or plan mitigation strategies during the incident handling process. A crafted quality control would be more appropriate in such a scenario since the specialist skills of the incident handler are depended upon.

Coincidentally these decisions are inter-related when viewed as a complex adaptive system. Decisions that are made at a lower tier could affect the decisions that are made at a higher tier, and vice versa. Myburgh (2014) affirms that varying amounts of both controlled quality (CoQ) and crafted quality (CrQ) may be necessary to ensure a successful project outcome. Whereas CrQ is required to apply quick and flexible decision making processes, CoQ is required when introducing more stringent governance requirements. The premise, however, is that effective decision-making processes cannot be attained through rudimentary adoption of a “secure software development process”.

Michael Matthee: michael.h.matthee@gmail.com

Boehm's (2003) risk-based approach together with the environmental analysis of Section 2.2 forms a good start for establishing a good development atmosphere. However, thereafter the security framework and metric system of Section 2.4 is required to keep the development procedure in line with changing circumstances and concerns in the workplace.

According to (Brotby & Hinson, 2013), metrics and measurements should be driven by business needs. The ease at which a measurement can be made should not be a prerogative for selecting a particular metric. Consider for example the attacks stopped by a firewall. The number of attempted intrusions would be easy to measure; however, it does very little in maturing and re-aligning the strategic decisions required by senior management. The metrics outlined in Section 2.4 are quintessential to the formation of secure software products.

3. Conclusion

In conclusion, security is best enforced in a controlled manner at times, using CoQ, and empirically using CrQ on other occasions. Care should be taken not to enforce a security practice or control in abstention of its surrounding environment and the unique situational circumstances at play. Influential forces that may affect the decision making process and consequentially deviate the software construction effort from effective practice are:

> Economic: Security controls can be applied under strict time constraints in order to get it done quickly and gain an early time-to-market. To enable quick and rapid development, a more crafted or dominant-agile development process is likely to be favored. If a practice is enforced inappropriately then it could result in a self-directed (SeD) or controlled costs (CoC) quality control that is best avoided. At the one extreme, an oversupply of time and resources without reasonable time constraints may result in misappropriated and over-engineered solutions with little business value; a bearing of self-directed (SeD) quality. At the other extreme, demanding impervious solutions with insufficient time and resources is likely to

produce weak, fragile and vulnerable products. This is a repercussion of combining tight budgetary controls and managerial oversight with unreasonably high expectations (a bearing of a CoC enforcement style).

Engineering: The engineering team may end up over-engineering a solution if they are not managed with deadlines and managerial oversight. This may skew security controls into the self-directed (SeD) territory, which would reduce valuable organizational output. Security measures of little to no value for the organization could be implemented in such situations. At the other extreme, not giving engineering practitioners room to conduct research and innovate may hamper their ability to produce software solutions that are relevant and up to date security-wise.

Political: The intricate nuances of adhering to strict organizational policies may encourage those with sufficient power to either circumvent or alter applicable security controls. This may be for the sake of convenience or to be on par with a larger political objective. This would make security controls ineffective and push controls into the self-directed domain (if controls are nullified) or controlled costs territory (if debilitating control mechanisms are strictly enforced while speedy delivery is demanded). Both of these domains, the controlled costs domain and the self-directed domain are not sustainable in the end, and may result in despondent team members, poor security implementations and/or malicious activity.

In order to avoid such unwelcome eventualities, the entire software construction endeavor is best tied to a security metric system as outlined in Section 2.4. Data-driven decisions using an appropriate information security metrics system can mitigate such deviances. For example, a business often needs to balance application security enhancements against new revenue generating features. The lack of visibility within the development environment makes such decisions difficult, very often favoring the immediate financial returns of the latter. However, by making

software security assets visible and tangible allows management to better quantify the financial benefits of investing into security tools and practices.

The style of enforcing a particular practice should not be downplayed either. Misappropriation of security controls and practices may steer an otherwise positive environment into debilitating bureaucracy, stifled creativity, politicking or disruptive chaos. It is best to enforce plan-driven practices formally (using a controlled quality enforcement style) while standing back and encouraging the quality of crafted and empirical practices informally (using a crafted quality enforcement style). At times, a controlled quality decision would be appropriate in order to gain long-term strategic value within the business. Other times it is best to rely upon the expertise of security specialists and grant them more autonomy to be effective. Software security controls and practices can now be applied that are fit for purpose, in the right manner, at the right time and under the best of circumstances.

Michael Matthee: michael.h.matthee@gmail.com

4. References

agilemanifesto. (2014, June 3). Manifesto for Agile Software Development. Retrieved June 3, 2014, from agilemanifesto: http://agilemanifesto.org/

Atlassian. (2014, May 1). JIRA. Retrieved 2014, from atlassian: https://www.atlassian.com/software/jira

Boehm, B., & Turner, R. (2003). Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Professional.

Brotby, W., & Hinson, G. (2013). PRAGMATIC Security Metrics: Applying Metametrics to Information Security. Auerbach Publications.

checkmarx. (2014, May 8). checkmarx. Retrieved May 8, 2014, from checkmarx: http://www.checkmarx.com/

Copeland, L. (2014, May 8). Cloud_Event_Polteq_Lee_Copeland_Testing_Trends_and_Innovations. Retrieved May 8, 2014, from polteq.com: http://www.polteq.com/wp-content/uploads/2012/09/Cloud_Event_Polteq_Lee_Copeland_Testing_Trends_and_Innovations.pdf

Dictionary.com. (2014, May 7). dictionary.reference.com. Retrieved May 7, 2014, from dictionary.reference.com: http://dictionary.reference.com/

Dyer Jr., G. W., Dyer, J. H., & Dyer, W. G. (2013). Team Building: Proven Strategies for Improving Team Performance, 5th Edition. Jossey-Bass.

Erwin, S. I., Magnuson, S., Parsons, D., & Tadjdeh, Y. (2014, May 31). TopFiveThreatstoNationalSecurityintheComingDecade. Retrieved May 31, 2014, from nationaldefensemagazine: http://www.nationaldefensemagazine.org/archive/2012/November/Pages/TopFiveThreatstoNationalSecurityintheComingDecade.aspx

Gustafson, D. A., Melton, A. C., Chen, Y. C., Baker, A. L., & Bieman, J. M. (1988). The software process model. Computer Software and Applications Conference (pp. 3-9). Chicago: IEEE.

Harrington, H. J. (1991). Business process improvement: The breakthrough strategy for total quality, productivity, and competitiveness. McGraw-Hill.

Information Resources Management Association. (2013). Software Design and Development. In J. Fonseca, & M. Vieira, A Survey on Secure Software Development Lifecycles. Portugal: IGI Global.

Michael Matthee: michael.h.matthee@gmail.com

ISO/IEC 27001. (2013). Information security management. Geneva: ISO/IEC.

ISO/IEC 27034. (2014). Information technology — Security techniques — Application security. Geneva: ISO/IEC 27034.

Jacobson, I. (2012). Refounding software engineering: The Semat initiative (Invited presentation). Software Engineering (ICSE), 2012 34th International Conference on. IEEE.

Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I., & Lidman, S. (2013). The Essence of Software Engineering: Applying the SEMAT Kernel. Addison-Wesley Professional.

Merkow, M. S., & Breithaupt, J. (2014). Information Security: Principles and Practices, Second Edition. Pearson Certification.

Merkow, M., & Raghavan, L. (2010). Secure and Resilient Software Development. Auerbach Publications.

Microsoft. (2010). Microsoft Security Development Lifecycle. Simplified Implementation of the Microsoft SDL . Microsoft.

Myburgh, B. (2009, January 21). Towards Understanding The Relationship Between Process Capability And Enterprise Flexibility. Insyte Information Systems Engineering .

Myburgh, B. (2010). Can project management survive in the information age ? Insyte Information Systems Engineering.

Myburgh, B. (2014). Situational Software Engineering. Federated Conference on Computer Science and Information Systems , 2, 841–850.

Noopur, D. (2014, June 3). secure-software-development-life-cycle-processes. (D. o. Security, Producer) Retrieved June 3, 2014, from buildsecurityin.us-cert.gov: https://buildsecurityin.us-cert.gov/articles/knowledge/sdlc-process/secure-software-development-life-cycle-processes#agile

OpenSSL. (2014, May 24). about. Retrieved May 24, 2014, from openssl.org: http://www.openssl.org/about/

SANS. (2014, May 8). closing-book-heartbleed-avoiding-future-sad-stories-98210. Retrieved May 8, 2014, from www.sans.org/webcasts: https://www.sans.org/webcasts/closing-book-heartbleed-avoiding-future-sad-stories-98210/success

SANS. (2014, May 8). critical-security-controls. Retrieved May 8, 2014, from sans.org: http://www.sans.org/critical-security-controls/

Michael Matthee: michael.h.matthee@gmail.com

Scacchi, W. (2001). Process Models in Software Engineering. In J. J. Marciniak, Encyclopedia of Software Engineering, 2nd Edition. New York: Wiley-Interscience.

Schneier, B. (2013). Carry On: Sound Advice from Schneier on Security. In B. Schneier, Carry On: Sound Advice from Schneier on Security. John Wiley & Sons.

Scrum.org. (2014, May 5). SCRUM. Retrieved May 5, 2014, from https://www.scrum.org/

smartbear. (2014, May 8). soapui. Retrieved May 8, 2014, from soapui: http://www.soapui.org/

sonarsource. (2014, May 8). sonarsource. Retrieved May 8, 2014, from sonarsource: http://www.sonarsource.com/

Vinod, V., Anoop, M., Firosh, U., Sachin, S., Sangit, P., & Siddharth, A. (2008). Application Security in the ISO27001 Environment. IT Governance Ltd.

Vliet, H. v. (2008). Software Engineering: Principles and Practice. John Wiley & Sons.

Wikipedia. (2014, October 16). http://en.wikipedia.org/wiki/Stuxnet. Retrieved October 17, 2014, from http://en.wikipedia.org/wiki/Stuxnet: http://en.wikipedia.org/wiki/Stuxnet

Win, B. D., Scandariato, R., Buyens, K., Grégoire, J., & Joosen, W. (2007, May 20-26). On the Secure Software Development Process: CLASP and SDL Compared. Software Engineering for Secure Systems .

Michael Matthee: michael.h.matthee@gmail.com

5. Glossary

Activity	Defines one or more kinds of work items and gives guidance on how to perform these.
Activity space	A placeholder for something to be done in the software engineering endeavor.
Alpha	An element (an attribute) of a software engineering endeavor, which has a state relevant to assess the progress and health of the endeavor. Aspiration Led Progress and Health Attribute is the mnemonic.
CLASP	The Comprehensive, Lightweight Application Security Process is a software development process that instills security practices during an application development endeavor.
Controlled costs	Application of a strictly enforced formal management style on tasks that are unclear or empirical in nature.
Controlled quality	Enforcing the security and quality of well-defined and systematic software construction through formal management methods.
Crafted quality	Encouraging the security and quality of empirical, experimental and innovative software artifacts using an informal management style.
ISO27001	The ISO27001 standard is a systematic approach to managing sensitive company information (including software related artifacts) so that it remains secure.
ISO27034	The ISO27034 standard provides guidance on the specification, design, selection and implementation of information security within software applications.
Method	A composition of practices that describes a team's way of working.
Practice	A repeatable approach to doing something with a specific purpose in mind (to address a specific challenge).
Self-directed quality	Application of a loosely defined informal management style on tasks that are systematic and well defined in nature. Delegating the responsibility of task management and completion to individual subordinates.
TDD	Test-driven development is a development practice where a small test case (or unit test) is written before the code is implemented. This increases code reliability, malleability and instills good software design structures.

Michael Matthee: michael.h.matthee@gmail.com

XP	Extreme programming is a lightweight software development methodology that favors frequent releases and short development cycles, programming in pairs and is characterized by the extensive use of code reviews and unit testing.