Information Security Journal: A Global Perspective, 18:8–25, 2009

ISSN: 1939-3555 print / 1939-3547 online
DOI: 10.1080/19393550802623206

Taylor & Francis
Taylor & Francis Group

Secure Software Engineering: Learning from the Past to Address Future Challenges

Daniel Hein[1] and Hossein Saiedian[2]
[1]Garmin International, Inc., Olathe, Kansas, USA
[2]EECS, The University of Kansas, Lawrence, Kansas, USA

ABSTRACT This paper provides a taxonomy of secure software systems engineering (SSE) by surveying and organizing relevant SSE research and presents current trends in SSE, on-going challenges, and models for reasoning about threats and vulnerabilities. Several challenging questions related to risk assessment/mitigation (e.g., "what is the likelihood of attack") as well as practical questions (e.g., "where do vulnerabilities originate" and "how can vulnerabilities be prevented") are addressed.

KEYWORDS metrics for software security, secure software systems engineering, security threats, software security vulnerabilities

1. INTRODUCTION

Consider the case of a manager or technical lead responsible for developing a new "connected" software product, perhaps a Web-based server application. Being connected, this product will use Internet infrastructure and/or technologies to meet one or more of its requirements. Early in the development cycle, perhaps during requirements definition, the subject of security arises. The manager has been concentrating on functional requirements and features but has given little thought to the subject of security. This hypothetical scenario sparks several practical questions relating directly to the field of secure software engineering (SSE):

- Will we really be attacked? Could the software really be at risk for attack?
- Isn't security something handled by network administrators? How does the software we're developing open vulnerabilities?
- Aren't there already solutions we can add on to provide the needed protections?
- Should we be concerned? How would a security breach affect our organization, product, and/or customers?
- As the development team, how should we mitigate security threats? How do we go about building secure software?
- How do we know we're doing a good job of making our software more secure?

Address correspondence to Prof. Hossein Saiedian, EECS, 2001 Eaton Hall, 1520 West 15th St., The University of Kansas, Lawrence, KS 66045. E-mail: Saiedian@ku.edu

The scenario is fabricated, but the questions are based on real-world conversations the author has had with managers and other developers. In addition, papers such as (Hoglund & McGraw, 2002; vanWyk & McGraw, 2005; vanWyk & Steven, 2006) support the legitimacy of the above questions and their industry-wide generalizations. After reading this paper, a software engineering manager or lead developer should be better equipped to answer the above questions. The primary contribution of this paper is to consolidate and interpret SSE research, making it more widely accessible to a practicing industry audience, which (judging by the current state of affairs) is overdue.

The current state of affairs, at least in the commercial sector, finds a feature-centered development culture where security concerns often get little attention (McGraw, 1999). One needs only to look at the non-stop stream of news headlines publicizing the latest exploits testifying that security has yet again been overlooked in the software development life cycle (SDLC) (Viega & McGraw, 2001). It's not that the commercial sector doesn't care about security; rather, it's more the case that managers and developers do not understand how software development activities relate to end-product security. The current majority of commercial development community is largely, as Wyk and Steven state, ". . . completely blind to how the software their building or maintaining today could be exploited tomorrow" (2006).

1.1. Fundamental Question and Motivation

A fundamental question is, "Will we really be attacked? Or, could the software really be at risk for attack?" First, this question should not be dismissed lightly, since at least theoretically *any* money, time, or other resources devoted to security will be completely *wasted* if you are *never* attacked. Second, this question is not trivial; it's difficult to quantitatively determine attack "attractiveness." Intuitively, factors such as connectivity, popularity (Alhazmi, Malaiya, & Ray, (2005), attack difficulty, and market penetration play a role in estimating how attractive your company or product is to would-be attackers, but it's difficult to tangibly relate these factors to an attack probability (i.e., used in risk analysis). With that said, the attack probability is most likely not zero. Historical statistics, for example, NIST National Vulnerability Database (NIST, 2007), US-CERT vulnerability notes (CERT, 2007, and other anecdotal evidence (e.g., news headlines) *strongly* suggest that the correct question is not "*Will* I be attacked" but rather "*How will* I be attacked?"

If we've learned anything from the past, it should be that software products we produce will be deployed into hostile environments, often beyond our control, where our software will be pushed to the limits and used in unintended ways to accomplish intentionally malicious objectives. This has never been truer than the present time, and the situation isn't getting any better – especially for software running in a connected environment (Hoglund & McGraw, 2004). While many improvements have been made in firewall technologies, operating systems, and secure/encrypted sockets, the root cause of many security vulnerabilities has been the software itself (Hoglund & McGraw, 2002). Moreover, due to improvements made in other areas such as operating systems, firewalls, and secure communications, and because attackers will choose the path of least resistance, experts predict attackers will increasingly attack the application layer directly (Ahmad, 2007).

Although the concept of SSE is not new, the field has received increased attention in the last decade. Part of this increased attention is from the realization that the current so-called "penetrate and patch" approach to security is inadequate Essafi, Labed, & Ghezala, 2006) because the applied patches often fix similar vulnerabilities that frequently reappear in software (Hoglund & McGraw, 2002; McGraw, 1999). For example, at the time of this writing, 13 of the 20 most severe US-CERT vulnerability notes documented over the last 10 years were buffer overflow vulnerabilities (US-CERT, 2007). The buffer overflow vulnerability is a classic vulnerability that has been widely discussed in software security literature, (e.g., Landwehr, 1994; Krsul, 1998; McGraw, 2003, Viega et al., 2002; Viega et al., 2000; Zitser, Lippmann, & Leek, 2004). Note that (Landwehr et al., 1994) is a widely cited classic flaw taxonomy dating back more than 10 years, demonstrating that the buffer overflow vulnerability has been repeated for at least as long. In addition, note that the buffer overflow vulnerability in particular is widespread. According to Zitser, Lippmann, and Leek (2004), buffer overflows "account for roughly one-third of all the severe remotely exploitable vulnerabilities listed in the NIST ICAT vulnerability database."

Another factor contributing to the increased attention in SSE is connectedness and technology convergence in embedded devices (Hoglund & McGraw, 2002) and, in particular, wireless phones. Many desktop applications such as Web browsers, email, media players, and instant messaging are converging with the mobile domain (Garcia & Horowitz, 2007). Many of these applications have longstanding, widely publicized security issues. The number of wireless phone customers, combined with the convergence of longstanding security-plagued desktop features, implies that these features must be specified, designed, implemented, and tested better to avoid the widespread security issues that currently plague desktop systems.

Finally, interest in secure software engineering is increasing due to its financial benefits. There is financial justification for applying SSE processes, principals, and best practices throughout the SDLC, especially early on, rather than applying patches later in maintenance. Software engineering circles have long recognized that software development/maintenance costs increase over time and that the most effective cost savings are realized with early corrective action; "often a 100 times more cost effective" (Bocehmand & Basili, 2001). Hoo, Sudbury, and Jaquith (2001) examined the software development and maintenance costs (e.g., patch development costs) associated with security in particular and concluded that return on investment (ROI) was "from 12–21%, with the highest rate of return occurring when analysis is performed during application design." Likewise reinforcing the notion of early consideration, the final security assurance attained by software products has also been shown to improve when security is considered early in design (Sachitano, Chapman, & Hamilton, 2004). Security improvements indirectly translate into operational cost savings since companies and/or users are less likely to incur losses and down-time when using a more secure software product; less time and money is wasted recovering from attacks enabled by software security vulnerabilities.

1.2. Intended Audience

This paper is primarily targeted toward software project managers and software developers in the commercial sector, although the contained information and concepts are widely applicable. More specifically, this paper is written with the consumer electronics and consumer software industries in mind.

Commercial and consumer-centered organizations are important for several reasons. First, these organizations represent technologies and products that have become deeply integrated within the fabric of our lives (e.g., mobile phones and Internet), facilitating expanding avenues of communication and commerce. Second, commercial organizations develop these technologies within a competitive atmosphere where time-to-market pressures are more likely to cause security oversights. Finally, the commercial sector faces much less government regulation when compared to such sectors as health care, military, and financial (Garcia & Horowitz, 2007). Time-to-market pressures combined with the absence of regulations allows commercial organizations to be largely self-policed. Therefore, barring any "altruistic intentions", the primary motivations in the commercial sector are financial (Anderson, 2001).

Organizations operating in the commercial sector are primarily motivated by financial considerations (Anderson, 2001). Consequently, the security improvements in and security assurance provided by products built by such organizations are motivated by financial considerations; this can be contrasted with a nonprofit government contractor who might be motivated to comply with Department of Defense (DOD) Common Criteria (CC Technical Report, 2007) for contract bidding/award eligibility. Commercial entities are motivated to improve the security of consumer products in order to meet customer demands and to satisfy competitive pressures. Perhaps the key motivation to improve product security, whether for a mobile phone, Web server, operating system, or shrink-wrapped software, is to maintain or increase market share (Anderson, 2001). If a competitor creates a more secure product, and your product suffers a publicized security breach, you face the danger of loosing your customers to your competitor. Depending on the importance of the product to your business, such a loss could be devastating.

1.3. Paper Objectives

This paper attempts to help the reader answer the previously mentioned questions by providing a better understanding of SSE. Understanding is provided by interpreting the relevant literature, consolidating past and present research in one place. A secondary goal of this paper is to highlight some of the difficult to answer questions and limitations within the SSE field.

These questions likely represent areas where more research and improvement is needed; areas that academia might work to address.

1.4. Paper Organization

This paper is organized as follows. A high level overview of the SSE field and its associated challenges is presented in Section 2. Section 3 follows, attempting to precisely define vulnerabilities, describes their origins and their relationship to threats, and discusses current efforts to classify vulnerabilities. Section 4 blends the discussion of threats and vulnerabilities into the topic of risk analysis for computer/network security in general. Section 4.1 demonstrates how threat/vulnerability relationships can be used to specialize a standard operational risk assessment calculation for software. Finally, Section 5 outlines several best practices for improving and monitoring software security.

2. OVERVIEW

This section describes what SSE is and is not. Simply put, secure software engineering is not necessarily engineering security software. SSE seeks to apply processes, principals, and methods to build vulnerability-free software, software that remains in a secure state under attack and continues to provide service to authorized users (Avizienis et al., 2004). Key questions (listed in the introduction) addressed by this section include the following:

- Isn't security something handled by network administrators? How does the software we're developing open vulnerabilities?
- Aren't there already solutions we can add on to provide the needed protections?

As stated above, SSE is not necessarily engineering security software. SSE can certainly be applied to build security software, but SSE techniques can be applied to building *any* type of software. Furthermore, as will be discussed shortly, it is becoming increasingly important to look at applying SSE techniques to applications software.

There is a tendency among many software developers (even those with many years of experience) to view computer and network security as an operational subject that the IT department or network administrators handle (van Wyk & McGraw, 2005). In the IT context, security assurance is obtained by installing/configuring firewalls, keeping virus definitions up to date, and applying the latest patches. Note that firewalls and patches are actually "band-aid" solutions that compensate for software vulnerabilities (Hoglund & McGraw, 2002). The correct context to address the root cause of most computer security failures is in the software development context (McGraw, 2004).

When the topic of security is raised in the development context, an almost reflexive reaction is to think about specialized security features such as cryptography, authentication, and copy-protection and how the development team might add on or integrate software components providing these security features (McGraw, 1999). The tendency to apply add-on security components results from the fact that many of these components 1) often implement sophisticated and proven encryption and 2) are often already prepackaged or provided by the underlying operating system (OS) platform. It simply makes good sense, both in terms of security assurance and costs, to reuse such components (Peine, 2005). However, the development team must realize that such add-on components are not a security panacea. Security assurance cannot be achieved by simply "bolting-on" security software. The requirements, design, code, and selected implementation language may all impact security.

In a complete system, a Web server, for example, there are several categories of software working together, packaged in different logical components and often organized in layers (i.e., application, network, OS/kernel/driver). A vulnerability in any component at any layer provides opportunity for an attacker. Any layer of the system may be attacked, and often an attacker will target the weakest layer (Barnum & Sethi, 2006; Kim et al., 2007). In fact, since much research has led to security improvements in the operating system, networking, and cryptography layers (Popek and Kline, 1979), attackers are increasingly targeting the application layer (Panko, 2003).

Secure software engineering is concerned with engineering software (all types) such that the end product provides some level of security assurance. SSE is predicated on fact that attackers frequently exploit vulnerabilities originating within the software development life cycle (SDLC); during requirements, design, implementation, or are missed during verification (McGraw, 2003). Section 3 more deeply explores and defines the term *vulnerability* and associated high-level concepts.

Understanding exactly what is meant by "vulnerability" and how attackers exploit vulnerabilities is essential for understanding SSE and its associated challenges. Section 2.1 provides an overview of some of these challenges.

2.1. Current Challenges

Current ongoing research in the secure subfield of software engineering focus on security centered:

1. Developer education,
2. Software development processes,
3. Best practices,
4. Threat enumeration and classification,
5. Requirements and Abuse cases,
6. Design and Architectures,
7. Testing, and
8. Metrics.

In terms of academic research accomplished and research needed, it seems that many of our future technical challenges will come from the requirements, design, and metrics areas. Challenges in the commercial sector entail effectively leveraging existing research accomplishments (primarily related to reducing implementation level defects) from academia in practical ways.

As an industry/science, we are currently capable of addressing many common and high frequency implementation level defects (Wing, 2003). Such implementation level defects include the notorious buffer overflow and its more general class of input-validation-related defects (Tsipenyuk, Chess, & McGraw, 2005). According to Wing, "We have the technical solutions in hand to detect or prevent these attacks; so it is a matter of deploying them in an effective, scalable, and practical way." However, based on personal experience, the author would argue that deployment of those tools is ongoing.

There is still a wide gap between progress made on the academic front and the state of practice in industry across all areas, and implementation in particular. Implementation level defects account for approximately half of all exploited vulnerabilities (McGraw, 2006); with the remaining half stemming from defects in requirements and design. Even if the available tools for implementation were more widely deployed, there would be a scant number of developers (when compared to the majority) that would know how to use these tools (McGraw, 2004). For example, ignorant-minded application of static analysis tools is likely to result in many false positives (Viega et al., 2000), possibly distracting developers from discovering real defects.

Aside from deploying technical solutions in the form of tools, probably the single most important factor in reducing implementation level defects before introduction would stem directly from improvements in security-centric developer education. Developer education is critical because exploit-enabling defects unknowingly introduced into software will continue to be unknowingly introduced into software unless the development populace better understands attacks and the vulnerabilities that enable them. Moreover, education as mentioned above should not only provide the answers to *what* threatens software and *how* those threats are avoided, but also *why* it is advantageous to address the security problem within the software rather than looking to operational solutions (e.g., firewalls). To be effective, education about *why* it is advantageous to address the security problem within software needs to crossover into management and propagate to the top. Management must lead by example, showing real commitment to security improvement by allocating time and money to properly institute the processes and practices of SSE. Without management backing, any software security "initiative" is likely to be hype and lip-service (van Wyk & Steven, 2006).

Other areas of focus showing the most promise for academic research and advancement are requirements, design, and metrics, with metrics being a key area (Geer, Hoo, & Jaquith, 2003). Requirements and design are promising because they are early activities in the SDLC, and as indicated previously, earlier SDLC investments/improvements are most likely to translate into the most significant cost savings. Often, however, what is really desired are methods/metrics to evaluate requirement and design artifacts with respect to their impact on the emergent software security attributes; this points to development of early life-cycle metrics that can be used as indicators of security assurance.

Note that process, practices, and testing (SQA) are also all important, but with the exception of testing, may not represent current "hot spots" for research. However, processes, practices, and testing also are likely to build on requirements, design, and metrics. For example, testing is likely to directly benefit from advances in testable security requirements. Also,

processes in particular, while likely to be critical for industry success, are fairly well understood at this point, being able to leverage standard-issue software engineering process research in a fairly straightforward manner. Key information yet-to-be determined for processes include the results of process comparison as in Gregoire et al. (2007). Identifying and establishing metrics for process comparison as they relate to end-product security assurance seems a fundamental prerequisite. Processes can tell us how to organize our requirements, design, implementation, review, and testing activities, but it is difficult to quantitatively analyze process effectiveness (especially when comparing several processes) without the proper metrics.

Metrics are important to know whether or not we're actually improving security assurance (Geer, Hoo, & Jaquith, 2003). Common techniques for determining if one software product is "more secure" than another involve comparing the number of security advisories logged against each version in a post-mortem fashion, with vulnerabilities identified *after* release. For example, although looking at completely different things, both Alhazmi, Malaiya, and Ray (2005) and Sachitano, Chapman, and Hamilton (2004) use this post-mortem approach for source data and comparison. Furthermore, this post-mortem advisory counting technique only works on products from the same product line or that perform identical functions. In the case of Sachitano et al. (2004), the security attributes of two mail programs were being compared. In the case of Alhazmi et al. (2005), a model for vulnerability discovery rate in subsequent operating system versions was being validated.

One of the reasons metrics continue to be actively researched is that the academic community is still getting a handle on cognitive security models (e.g., threat and attack models) to reason about software and a standardized taxonomy for vulnerability classification(McGraw, 2006). That is, the research community is still figuring out exactly what data should be measured, how the data should be characterized/classified, and by what names the data should be known (Geer, Hoo, & Jaquith, 2003).

The other issue that makes developing security metrics difficult is the nature of the security problem. The effort required by an attacker to violate software security defenses is linear, requiring an attacker to find a single weakness to exploit. As Bellovin (2006) states, "Whatever the defense, a single well-placed blow can shatter it." On the other hand, the effort required to *assure* that software is secure is exponential, requiring exhaustive and comprehensive knowledge of the software and all its possible interactions with its environment; this simply isn't tractable (Bellovin, 2006).

3. VULNERABILITIES

This section is intended to more rigorously discuss the term *vulnerability* and comment on its various related dimensions. Section 3.1 provides a brief primer on the origins of vulnerabilities in software. In Section 3.2, the relationships among vulnerabilities and other common terms such as threat and attack will be discussed. Threats, vulnerabilities, and their relationship to risk analysis will also be briefly addressed in 3.2. Finally, Section 3.3 expands on the relationship between threats and vulnerabilities, suggesting ways in which the noted relationship could be leveraged within current vulnerability taxonomy efforts to provide additional information; information that could be used to facilitate risk analysis and/or drum-up interest in adopting SSE processes and practices. For now, a better understanding of the term vulnerability is required before the motivating question, "How does the software we're developing open vulnerabilities?" can be answered accurately.

Krsul (1998) defines vulnerability as "an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate the [implicit or explicit] security policy." While the definition is comprehensive, capturing the essence of the term vulnerability, some substitutions will help when relating to contemporary SSE literature. In particular, this paper substitutes *defect* for *fault* and *security requirements* for *security policy*.

The use of *fault* follows directly from its standard use in dependability literature such as Avizienis et al. (2004). The term *defect*, identical to fault, is used more extensively in risk analysis literature such as (Alhazmi, Malaiya, and Ray (2005) and Anderson (2002). More recent SSE literature (e.g., McGraw, 1999, 2003; McGraw & Potter, 2004) also favors the use of *defect,* and so this paper will also use the term defect.

In the larger computer security context, a policy is an official statement about the roles and responsibilities of various individuals with respect to information and computational assets. Generally speaking, the term policy is usually framed in the larger context of

computer/network security, not just software security. In the overall computer/network security context, a policy *specifies* acceptable and unacceptable use, actions constituting abuse, and punishments for violators (Panko, 2003). In terms of software, such high level specification translates into the software's implicit and explicit *security requirements*.

The important realization to take away from the previous substitution discussion is that software security assurance is ultimately a subset of software quality assurance (Davis et al., 2004). Software quality assurance is concerned with assuring that the software conforms to its requirements; defects prevent the software from meeting those requirements. In the more general field of standard-issue software engineering, approaches to improving software quality focus on preventing and/or removing defects throughout SDLC stages. SSE is concerned with eliminating (or at least reducing) software defects/faults threatening security; such defects are commonly called vulnerabilities. In this regard, SSE seeks to improve the quality of security attributes by preventing and/or removing vulnerabilities throughout the SDLC. Section 3.1 will now answer the question, "How does the software we're developing open vulnerabilities?" by detailing vulnerability origination within the SDLC.

3.1. Vulnerability Origin

Vulnerabilities result from defects. The defects may be easily identifiable, code level bugs resulting from implementation, or may result from more deeply seated issues/oversights (i.e., flaws) in the design or requirements (McGraw, 2003). The following list enumerates various phases of the SDLC and briefly highlights the ways in which vulnerabilities manifest:

1. *Requirements definition phase*, resulting from inadequate or often completely absent (Firesmith, 2007) security-centric requirements. Requirements may fail to consider unintended and malicious uses (Sindre & Opdahl, 2005). Security requirements are often stated as nonfunctional requirements, often specifying a particular technology to use rather than identifying threats and characterizing the threat environment. Coincidentally, later stages in the software development life-cycle may fail to address unintended/malicious uses of the software.
2. *Design phase*, resulting from failure to adequately account for security-related cross cutting concerns such as error handling, policy enforcement, and component composition (Tsipenyuk, Chess, & McGraw, 2005). The complete absence of considering security in design is also a possibility.
3. *Implementation phase,* resulting from use of nontype safe languages, insecure application programming interfaces (APIs), improper use of secure APIs, seeding by malicious developer, lack of secure coding practices, and/or simply inadequate developer education with respect to code-level defects traditionally exploited by attackers (Tsipenyuk et al., 2005).
4. *Verification/testing phase*, resulting failure to catch vulnerabilities introduced in requirements, design, and implementation phases. Testing personnel may lack architectural knowledge and/or malicious skills needed to probe security weaknesses. Testing tools, even those developed to test security, are currently limited to black-box attacks (McGraw, 2005) and may not integrate well with the system under test.
5. *Maintenance phase,* resulting from incorrect deployment configuration, or defects introduced by applying updates and bug fixes. Vulnerability origin in maintenance shares much in common with origin in previous phases since patches are typically created in maintenance. Since the patch itself is a software product, vulnerabilities can originate in many of the same places that they occurred in previous SDLC phases.

Vulnerabilities originate in a number of ways and can enter the software at various phases. Generally speaking, the earlier in the SDLC the vulnerability is introduced (or rooted in), the more difficult (and more costly) the corresponding defect would be to patch. For example, it might be fairly straightforward to patch a one-line coding error, whereas a "patch" in the traditional sense would be near impossible to "apply" to a flaw rooted deep within the design. In such a case, a complete re-design might be required to address the design flaw.

3.2. Vulnerabilities, Threats, and Attacks

Vulnerabilities, threats, and attacks are interrelated. Vulnerabilities enable certain attacks. The danger that an attacker will attempt to exploit a vulnerability

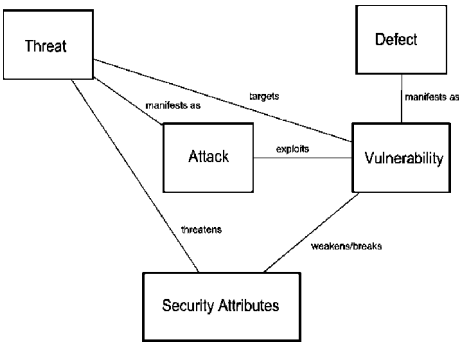

FIGURE 1 **Defect, Threat, Vulnerability Relationship.**

represents a threat. The relationship among vulnerabilities, threats, and attacks is shown in Figure 1.

We plan against threats and we defend against attacks. Specific threats represent the danger from particular attack types. In software, there is a strong relationship between particular attack types and the vulnerabilities they seek to exploit. This paper points out that threats, although existing independently from the vulnerabilities they target, are of little concern if said vulnerabilities do not exist in the first place. For example, a whole class of common threats enabled by C/C++ may be significantly reduced by simply switching from C/C++ to a more strongly typed language. A biological metaphor to germ threats may help relating the conceptual framework.

A Germ Metaphor

If a person has not been vaccinated for chicken pox, he or she is *vulnerable* to this disease. We might say that person is *threatened* by chicken pox germs, or vulnerable to a chicken pox *attack*; the disease *targets* vulnerable individuals. If the person gets a vaccine (assuming the vaccine provides immunity), the person is no longer threatened by chicken pox. A key difference between the germ metaphor and threats in the security context is *intent*. Germ threats naturally happen to exploit genetic vulnerabilities whereas attackers maliciously seek to exploit system vulnerabilities. Germ variants evolve and adapt by random mutations; attackers intelligently vary attack strategies.

The important analogous point illustrated by the germ metaphor is that there is a close relationship between individual threats and the vulnerabilities they would exploit. Although chicken pox exists independently in the threat environment, it may not threaten a vaccinated individual. If the individual's DNA were perfect and not susceptible to the germ threat in the first place, the person would not have needed a vaccine. Note that threats represent the danger of particular types of attacks. Also, threats exist independently from the vulnerability; simply because the software doesn't have a vulnerability susceptible to a particular attack doesn't mean that an attacker won't try that attack.

Threats and Vulnerabilities for Software

As demonstrated by the germ metaphor, threats are relative to their target vulnerabilities. Threats represent the danger that an attacker will seek to exploit an existing vulnerability in the target. Figure 2 illustrates a threat, *t*, that targets an exploitable vulnerability, *V*.

Notice that the illustration places *V* near the boundary, signifying that exploited vulnerabilities are often localized around input/output interfaces and the software/system boundary (e.g., files, communication channels, and system resources such as memory and CPU; Manadhata & Wing, 2005). For software, threats are only significant to consider when enabling vulnerabilities are present (or are likely to be present) in the target software. The danger threats represent is linked to the consequences of exploited vulnerabilities. If the corresponding vulnerabilities are removed, the threats that target them will no longer be of concern; those particular threats no longer *threaten* the software. This relationship is shown in Figure 3. Bear in mind that the vulnerability may be a latent design flaw (possibly an oversight propagated from requirements) or the common code-level defect (e.g., buffer overflow).

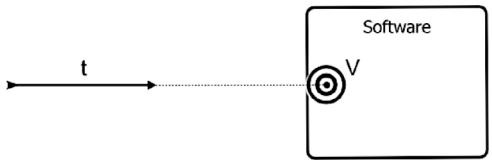

FIGURE 2 **Threat *t* targets vulnerability *V*.**

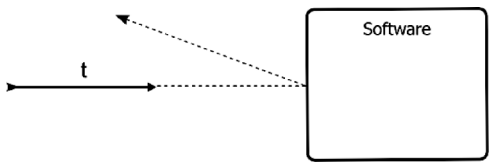

FIGURE 3 **Software immune to threat *t*.**

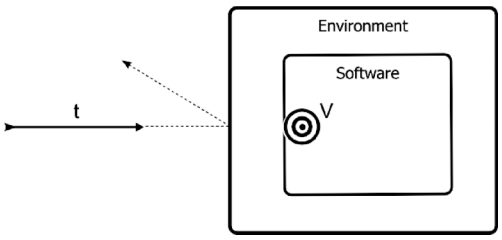

FIGURE 4 **Environment shields vulnerability *V* from threat *t*.**

Conversely, if the threat environment does not contain a threat that would exploit a given defect, then that defect can't legitimately be classified as a vulnerability. On a practical level, however, the information we have about the threat environment is historical. At any given point in time, the threat environment is characterized by *known* threats. That is, simply because no known threat currently exploits a defect, one cannot generally predict whether or not the defect could be exploited in the future. Incidentally, SSE is equally concerned with standard-issue software quality improvement and defect removal. Due to earlier insights, however, SSE is more likely to be concerned with detecting/removing defects at the input/output and software/system boundaries and defects similar in nature to those already known to be actively exploited. Also, some defects, say for example a spelling error in a UI field label, are extremely unlikely to have any security implications.

The relationships discussed between threats and vulnerabilities have important implications on traditional security-based risk analysis. At its heart, security-centered risk analysis for software (addressed in Section 4) should take a holistic view of the threat environment and how it relates to the deployment environment. Consequently, when transferring concepts of classical risk analysis and actually performing risk analysis for software, it is important to continue considering operational mitigation options offered by the deployment environment (Hoglund & McGraw, 2004).

For example, consider the case where a new Web application is being developed. It is quite possible that an existing firewall may already mitigate a threat to the Web application and do this in a cost-effective manner. That is, the application of a firewall mitigates a known threat (e.g., a LAND attack where source and destination IP addresses are both set equal), effectively removing consideration for the root-level software vulnerability (e.g., networking software enters unspecified state or doesn't properly check reply-to-self case). Because the firewall provides immunity from threat *t*, the LAND attack, it may not be necessary (outside defense-in-depth considerations) to incur the cost associated with mitigating what might be a design flaw in an externally provided component based protocol stack. Generally speaking, however, as threats move further up the protocol stack to the application level, there is less protection available from operational solutions such as firewalls. Operational solutions are also not common on the client side. In the absence of operational mitigation options, the software development team must mitigate the threat directly, hopefully by keeping the code free of threat enabling vulnerabilities, or as common practice, by providing a patch.

3.3. Vulnerability Taxonomy and Augmentation

This section suggests ways in which the threat/vulnerability relationship discussed in Section 3.2 can be leveraged within current vulnerability taxonomy efforts to provide meaningful recommendations for value-added metadata. This section also provides a brief introduction to vulnerability taxonomy research and suggests metadata recommendations for the current draft taxonomy of the Common Weakness Enumeration (CWE), sponsored by the National Cyber Security Division (NCSD) of the U.S. Department of Homeland Security (DHS). The proposed metadata would greatly facilitate early-stage risk analysis and could also potentially provide support for secure software development initiatives.

As mentioned previously, due to the close-knit relationship between threats and vulnerabilities, security software engineers are likely to be interested keeping the software free from defects which are similar in nature to those already known to be actively exploited "in the wild." We might call such vulnerabilities "like-vulnerabilities." As mentioned previously, developer education centered on currently exploited vulnerabilities is a key element to avoiding the re-introduction of like-vulnerabilities. An organizational framework, or taxonomy, for organizing and classifying vulnerabilities is an essential step toward this end.

Taxonomy Efforts and Rationale

A recent noteworthy work in the taxonomy area is Fortify's "Kingdoms" taxonomy (Tsipenyuk et al.,

2005). In fact, the earlier treatment of vulnerability origin in Section 3.1 was greatly aided and guided by the "Kingdoms" work. Tailored specifically to be easily remembered and to help transfer expert vulnerability knowledge to practicing developers (Tsipenyuk et al. (2005) defines the following categorization scheme (in order of importance to software security):

- validation and representation,
- API abuse,
- security features,
- time and state,
- errors,
- code quality,
- encapsulation, and
- environment

In both McGraw (2006) and Tsipenyuk et al. (2005), McGraw shows how the above classification can be applied to classify OWASP's Top Ten (2007) vulnerabilities list and the "19 Sins" list presented in the book titled *19 Deadly Sins of Software Security* (Howard, LeBlanc, & Viega, 2005). However, the very simplicity that makes the "Kingdoms" work so powerful is at the same time its shortcoming: While providing a straightforward classification scheme for developers to use in their day-to-day work, it doesn't meet the needs of management types or include the breadth and detailed granularity required by security researchers. Several important reasons enumerated by Martin, Christey, and Jarzombek (2005) motivate the need for a comprehensive and standardized vulnerability taxonomy, discussing how such a taxonomy would, in their words,

1. Provide a common language of discourse for discussing, finding, and dealing with the causes of software security vulnerabilities as they are manifested in code.
2. Allow software security tool vendors and service providers to make clear and consistent claims of the security vulnerability causes that they cover to their potential user communities in terms of the CWEs that they look for in a particular code language. In addition, a new type of CVE compatibility will be developed to allow security tool and service providers to publicly declare their capability's coverage of CWEs.
3. Allow purchasers to compare, evaluate, and select software security tools and services that are most appropriate to their needs including having some level of assurance of the level of CWEs that a given tool would find. Software purchasers would be able to compare coverage of tool and service offerings against the list of CWEs and the programming languages that are used in the software they are acquiring.
4. Enable the verification of coverage claims made by software security tool vendors and service providers. Verification is supported through CWE metadata and alignment with the software assurance and metrics tool evaluation (SAMATE) reference dataset.
5. Enable government and industry to leverage this standardization in the contractual terms and conditions.

Vulnerability taxonomy research has been ongoing since the mid-1970s (McGraw, 2006). Factors complicating the creation of a standard vulnerability taxonomy are the various users of the taxonomy, the level of information required by those users, and the technical difficulty inherent in classifying defects spanning multiple technologies, platforms, environments, and SDLC phases (e.g., requirements, design, and implementation). For example, a taxonomy with too many classification categories begins to approach a flat listing, where meaningful categorization is not provided, and vulnerabilities are therefore likely to be classified ambiguously (McGraw, 2006). On the other hand, without enough fine-grain attribution within the classification, it may be difficult for researchers to identify the particular "species" of vulnerability, which may depend on the type of software (e.g. application, systems, driver, protocol) and/or the programming language (e.g., vulnerabilities particular to C/C++ string functions).

The Common Weakness Enumeration (CWE)

After more than three decades of research, the DHS-sponsored (CWE) stands as a remarkable work-in-progress, combining several key ideas from past taxonomy research (Martin & Barnum, 2008), as the CWE Web site states

> ". . . we leveraged MITRE's [preliminary list of vulnerability examples for researchers] (PLOVER) effort as a starting point for the creation of the formal Common Weakness Enumeration. Not only does CWE encompass a large portion of the CVE List's 15,000 CVE names, but it also includes detail, breadth and classification structure from a diverse set of other

industry and academic sources and examples including the McGraw/Fortify "Kingdoms" taxonomy; Howard, LeBlanc and Viega's 19 Deadly Sins; and Secure Software's CLASP project; among others." (MITRE Corporation CWE List, 2007)

The CWE exists as a hierarchical tree structure, applying a Kingdoms-style classification near the mid level and then branching down into the detailed species of particular vulnerabilities found in the wild. The CWE draft Web site, in draft 7 as of this writing, provides an expanding/contracting tree view which in itself is a useful educational tool. As a developer himself, the author found it very useful and elucidating to browse the various branches, expanding them into more detailed form, examining the myriad ways, across multiple technologies, in which software can be exploited. One can only imagine how clever front-ends could be applied to real data organized by the CWE hierarchy at various levels and slices to create different views specialized for a multitude of disparate interests.

The CWE is a currently evolving community standard. According to Martin and Barnum (2008), required entity attributes for individual definitions is still in flux. A cursory review of various definitions revealed that some definitions included a "likelihood of exploit" while others did not. Based on the body of SSE literature, this author would recommend that this field be required for all definitions at the leaf nodes or final tier of the hierarchy. The next section discusses the importance of this metadata and argues for its required inclusion.

Recommendations for the CWE

This author recommends that the "likelihood of exploit" field be required for all leaf-node definitions. If possible, listing attacks, or attack patterns in a separate field which are known to exploit the vulnerability would also be extremely useful. These recommendations are based primarily on the end-goals of more easily assessing the threat environment, providing input into risk analysis calculations, and eliciting support for secure development initiatives.

One of the more difficult tasks in risk assessment is determining reasonable inputs for attack or threat probability. Currently, threat models must be developed and analyzed to provide "threat likelihood" inputs to risk analysis frameworks such as DREAD and NIST SP800-30 (Buyens, DeWin, & Joosen, 2007). However, based on the threat/vulnerability relationship discussed in Section 2 and the supporting data provided by Alhazmi, Malaiya, and Ray (2005), this author suggests that a close approximation for threat probability/likelihood would be the "likelihood of exploit" offered by certain CWE definitions. The key to this notion is that the discovery of vulnerabilities implies that in order to be discovered, those vulnerabilities are also being (or have been) actively attacked/exploited. Evidence of this relationship is reflected in some earlier taxonomy category names that incorrectly named the software defect with the type of attack or exploit it enabled.

Were the individual CVE entries in the NIST NVD database (NIST, 2007) categorized according to CWE definitions, front-end tools could be provided to quickly obtain situational awareness (e.g., what are we up against and do we need to be concerned?). Augmenting the "likelihood" of exploit with the types of attacks it enables, managers and architects could more easily determine if an in-development software product would likely face a similar threat environment once deployed. Questions such as those introduced at the beginning of this paper could more easily be answered in meaningful ways, assuming appropriate filtering can be done to obtain slices by technology and platform that most closely mirror the in-development software product. Recall such questions included

- Will we really be attacked? Could the software really be at risk for attack?
- Should we be concerned? How would a security breach affect our organization, product, and/or customers?

As stated previously, a number of tools could leverage the CWE taxonomy to provide interesting views on CVE data for different information consumers. In order for tools to provide these views, new and existing CVE data must first be classified according to the categories defined by the CWE.

4. THREATS AND RISK ANALYSIS

This section explains the role of threats and vulnerabilities in risk assessment. As mentioned before, while conceptually simple, risk assessment is not trivial in practice due to the difficulty of quantifying traditionally qualitative and intangible factors (e.g., target

attractiveness and loss of reputation). The definition and discussion of threats, attacks, and vulnerabilities, as well as the treatment of risk assessment, should help the reader answer the following questions:

- Will we really be attacked? Could the software really be at risk for attack?
- Should we be concerned? How would a security breach affect our organization, product, and customers?

Risk analysis is ultimately performed to determine how money should be spent. A question closely related to "How do we go about making our software more secure?" is "How do we go about spending money and decide where to place emphasis to reach our security goals?" Standard-issue software engineering performs risk assessment to determine how to allocate spending throughout the SDLC, within budget, such that the resultant software expresses the desired quality attributes. Schneidewind (2002) elegantly provides the software-based context for risk analysis with the question, "What is the cost of achieving quality goals and the risk of not achieving them?". The risk of not achieving security goals relates directly to the question "How would a security breach affect our organization, product, and customers?" In terms of SSE, much of the work entails accurately estimating the cost or *impact* resulting from the "risk of not achieving" the desired security qualities in the software. Note that knowing what security qualities (i.e., confidentiality, integrity, and availability) are actually *desired* depends on high level business objectives. For a commercial organization, specific security qualities are likely to be driven by market analysis, competition, and perhaps a corporate mission statement.

Different organizations will likely choose different relative priorities for threats with different high-level security implications, such as *confidentiality*, *integrity*, and *availability*; note that this also might suggest additional metadata required for CWE definitions discussed in Section 3.3. For example, an Internet Service Provider (ISP), or vendor providing software for the ISP sector, would likely place higher relative priority on threats affecting *availability*. Conversely, a bank or other financial institution would likely place higher relative priority on *integrity*. In the ISP versus bank example, the different emphasis on threat categories follows directly from the difference in market forces and how those forces weight the threat cost. An ISP stands the most to loose when customers become dissatisfied with poor service *availability* and sign up with a competitor (Garcia & Horowitz, 2007). Although a bank will have many unhappy customers if they can't access their accounts for a day, it stands to lose more if customer account balances are modified due to an *integrity* breach. Essentially, one size does not fit all when it comes to prioritizing threats, although particular domains may have similar relative priorities for generalized threat categories.

The risk of not achieving the desired security qualities is evaluated with respect to a given threat environment. Computer/network security researchers traditionally have analyzed risk by enumerating threats, determining the value of threatened assets, and making cost comparisons weighing potential loss against mitigation costs (Panko, 2003).

When looking at security risk analysis for software, many of the concepts from the classic computer/network security context are transferable. In fact, Verndon and McGraw (2004) state, "What separates a great software risk assessment from a merely mediocre one is its ability to apply classic risk definitions to software design and then generate accurate mitigation requirements." In the same paper, the authors outline three basic approaches to risk analysis as follows:

1. Financial loss methodologies that seek to provide a loss figure to balance against the cost of implementing various controls;
2. Mathematically derived "risk ratings" that equate risk with arbitrary ratings for threat, probability, and impact; and
3. Qualitative assessment techniques that base risk assessment on anecdotal or knowledge-driven factors.

In terms of traditional (i.e., nonsoftware-centered) security-based risk analysis, organizations typically attempt to determine how they should best allocate their resources (usually financial) in order to mitigate known threats. Security-based risk analysis involves threat identification, threat prioritization, and the development of a mitigation strategy (Panko, 2003; Mead et al., 2004). Threats are prioritized on a "bang for the buck" basis with top spots reserved for those threats most cheaply mitigated that also pose the highest loss potential.

4.1. Vulnerability-Based Risk Analysis for Software

Determining priority for software involves calculating a variant of the "value of protection" (Panko, 2003) formula. The "value of protection" formula represents a classical computer/network security risk assessment framework, classified as a "financial loss methodology" (Verdon & McGraw, 2004). The original formula is expressed as:

$$VoP = R - \mathrm{M}, \text{ or}$$
$$VoP = (Ap \times L) - M$$

where *VoP*, is the value of protection (aka priority), *R*, is the risk, calculated as $(A_p \times L)$, A_p, is the probability of a successful attack, *L*, is the loss resulting from a successful attack, and *M*, is the cost of the mitigation countermeasure.

Applying the relationship among attack, threat, and vulnerability (discussed in section 3.2, the equation can be better adapted for software security risk analysis. Realizing that A_p can be rephrased as the probability that an attacker will successfully exploit vulnerability *V* to realize a given threat, and that threats target vulnerabilities, the formula has been modified to approximate threat probability, A_p, with *P(V)*, which is defined below:

$$VoP = (P(V) \times L) - M$$

where *VoP*, is the value of protection (aka priority), *P(V)*, is the probability of a successful attack on vulnerability *V*, *L*, is the loss resulting from a successful attack, and *M*, is the cost of the mitigation countermeasure.

In the above equation, an assumption is that there exists a threat, *t*, in the threat environment, **T**, which threatens vulnerability *V*, or $\exists t \in$□ and *threatens (t, V)*. Threats are then ranked by *VoP*, with larger *VoP* receiving priority consideration and financial expenditure. In other words, if the financial loss *L* resulting from a security breech is large and the cost of mitigation, *M*, is small then the threat will be ranked near the top of the priority list (depending on its probability of occurrence *P(V)*). Understanding that a successful attack on vulnerability *V* could result in loss *L*, an organization is often willing to consider paying less than $P(V) \times L$ for a mitigating countermeasure. Countermeasure tactics may include SSE-related strategies for vulnerability removal, or may take a more traditional role if possible. As this existing framework is extended to software, it should include the methods traditionally used in security based risk analysis.

As expressed earlier, the root cause of many vulnerabilities is in the software itself. A popular mitigation countermeasure frequently purchased is a firewall. Often, companies buy firewalls to screen packets that seek to exploit software running on the internal corporate network. As already discussed, were the software built from the ground up to be immune to such attacks (focus of SSE), investment in firewalls wouldn't be as popular. However, software is far from perfect and it is doubtful we will ever be able to make defect or vulnerability free software (firewalls are likely here to stay). In addition, the cost required to make software more secure may exceed the cost of what might otherwise be considered a band-aid solution (Hoglund & McGraw, 2002) for the root problem.

5. TOWARD A BETTER TOMORROW

Working toward a better, more secure tomorrow starts with taking action today. This section provides a high level overview of specific actions that must be taken to actually build more secure software. The key questions addressed by this section are shown below:

1. As the development team, how should we mitigate security threats? How do we go about building secure software?
2. How do we know we're doing a good job of making our software more secure?

In general, increasing software security (topic of question 1) comes down to eliminating defects. As mentioned previously, defects originate in various phases of software development. Therefore, to improve security, key defects must be prevented in each phase. To evaluate security improvements (topic of question 2), appropriate metrics must be used in each phase to evaluate the relative assurance level as the software is developed. Question 1 is answered by Section 5.1, which discusses current life cycle activities for vulnerability prevention. Question 2 is then addressed by Section 5.2 which briefly discusses current metrics. Finally, this section ends with Section 5.3, which discusses practical considerations for

effectively executing techniques presented in the preceding sections.

5.1. Best Practices

Several best practices (i.e., touch points; McGraw, 2006) have been proposed for each phase in an effort to reduce defect introduction throughout the SDLC. A graphical representation of a typical software development phase timeline is shown in Figure 5.

Note that the graphic in Figure 5 closely mirrors a traditional waterfall development process, which may not fit the needs of many organizations favoring alternative development approaches (e.g., iterative; McGraw, 2004). Recent work such as Beznosov and Kruchten (2004) explores how the security-centered best practices illustrated in Figure 5 can be worked into agile processes. A brief description of each best practice follows:

Abuse cases - The negative of standard-issue use cases, abuse cases describe the malicious misuse and abuse of the system. Abuse cases help elicit functional, testable requirements for the software. Such requirements facilitate testing that the software behaves as specified when confronted with known, anticipated attack scenarios. Note that compelling abuse cases, in conjunction with preliminary risk analysis, can provide motivation for adopting other SSE best practices or spurring new secure software initiatives.

Security requirements - With the help of abuse cases, specific and testable security related requirements can be defined which should prove much more useful than general "hand waiving" goal declarations (e.g., software must remain secure). In additional, more obvious specifications such as when/how to authenticate, when/how to apply encryption, and particular detailed security technologies should also be defined.

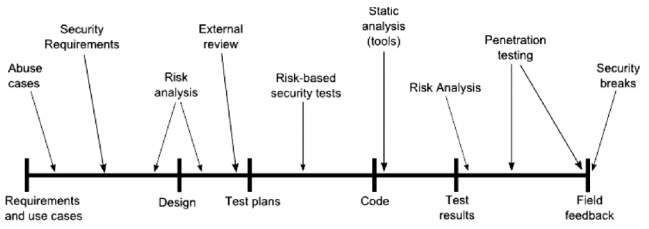

FIGURE 5 **Best practices applied to software artifacts throughout the SDLC (McGraw, 2004).**

Risk analysis - Risk analysis should permeate the SDLC. Risk analysis has already been described throughout this paper. Risk analysis attempts to answer the fundamental questions "how concerned should we be" and "how should we allocate resources to address our concerns?" Both questions largely depend on what threats exist, threat likelihood, and resultant impact/damage. Early-stage risk analysis compliments abuse case efforts to enumerate known/expected threats. Understanding the ways the software is threatened is paramount to (1) declaring behavioral policies (specified in requirements) that describe how the software should respond under specific attacks; (2) tailoring architectures (specified by design) that employ security principles (e.g., least-privilege) to counter threats; and (3) weighing potential damages/loss against investments/changes in process, training, external review, technology (e.g., language selection and static analysis tools), and any other conceivable effort applied in any other SDLC phase for the sake of attaining better security assurance.

External review - As security is a specialized field, the group responsible for design and implementation likely lack sufficient experience and expertise to formulate quality security requirements and properly specify a secure design. Review external to the development group is often necessary to illuminate oversights such as absent misuse cases, missing threats, flawed requirements, and unsafe design approaches. Expert knowledge at early stages can also help fine-tune the risk analysis input parameters.

Risk-based security tests - Security test plans should be developed that cover the security requirements, and are based on enumerated threats and attack models. Test plans should provide coverage for cases representing the highest risk threats. Test plans should attempt to violate secure design principals in order to verify proper implementation. Note that quality test plans require comprehensive and complete threat enumeration, quality requirements, and accurate risk assessment. As an interesting aside, test driven development techniques could utilize the test plans to formulate tests for secure code.

Static analysis - Static analysis is heavily focused on detecting/removing defects in the implementation. Static analysis can consist of peer-based code review, external code review, and static analysis

tools. Tool based static analysis is most directly applicable to detecting code-level bugs. Higher level, more abstract defects may be found by human inspectors, where discrepancies between the design and the implementation may be uncovered. As security often involves specialized expertise, it may be beneficial to enlist expert reviewers. As an aside, an interesting technique might augment test driven development practices with static analysis tools for feedback. The idea here would be to help more developers (without expert knowledge) to produce code with fewer vulnerabilities by leveraging the expertise embodied in tools.

Penetration testing - Penetration testing is fundamentally limited because it tests a negative; any degree of assurance requires infinite testing (Bellovin, 2006; McGraw, 2006). In his book on software security, McGraw (2006) further points out that the common practice of using automated attack tools can only show degrees of "badness" with respect to security; absolutely nothing can be said about assurance when using only canned black-box tests. However, penetration testing can at least verify that the software remains secure under the set of known and anticipated attacks. Alluded to earlier, good penetration testing should leverage white-box knowledge relating to specific requirements, software design, and likely threats enumerated via misuse cases and risk analysis. As McGraw states, ". . . any black-box penetration testing that doesn't take the software architecture into account probably won't uncover anything deeply interesting" (2004).

Security breaks - Security breaks represent successful vulnerability exploitation. Note that since penetration testing cannot test all future attack scenarios, a skilled attacker(s) bent on breaking in will break in eventually. Detection of security breaks provides an opportunity to measure/evaluate security (relative to a single product) by comparing discovered vulnerabilities to subsequent versions. Gathering data after detecting the break-in is also valuable for refining misuse cases and risk analysis. Because a breach is always possible, Requirements/design activities should specify mechanisms to facilitate patching.

5.2. Metrics for Software Security

As discussed near the end of Section 2.1, evaluating the security assurance level of software is often an exercise in comparing the number of discovered vulnerabilities *after* release. As discussed previously, these numbers only make sense when comparing subsequent versions of the same product, or products that perform the same functions (e.g., such as email handlers in Sachitano, Chapman, and Hamilton, 2004). Other postmortem metrics such as daily vulnerability exposure, or daily vulnerability exposure (DVE; Jones, 2007) have been noted which aim to indicate exposure by tracking counts of known vulnerabilities over time.

While post-mortem metrics will continue to provide a straightforward basis for comparison and monitoring, it is arguably more desirable to know whether or not security is improving *before* release. To this end, Nichols and Peterson (2007) suggest several concrete metrics that can be used prior to release to construct an overall "scorecard" indicative of security. Some metrics they present could be used as part of an overall risk management/project tracking strategy, prior to release, to gauge the relative improvement in the software's security. Theyr metrics, as well as (McGraw, 2006), applicable during design, implementation, and verification phases are listed below (and slightly reinterpreted) for easy and generic reference:

- *Percent validated input (PVI)* - A metric computed as $(I \times V) / I$, where V represents the number of validated input interfaces, and I represents the total counts of these input interfaces. The example given in Nichols and Peterson (2007) discusses a concrete application where I represents the count of HTML form POSTS and V represents the number of those form POSTS that are validated.
- *Throw-away error count (C_{tae})* - A metric directly after the "instance per application" count presented in Nichols and Peterson (2007), C_{tae} is simply a count of functions returning errors where the errors are not used or checked. These instances of occurrence can by identified by static analysis tools.
- *Attack-based vulnerability count* (CV_a) - A simple count, CV_a counts the number of vulnerabilities uncovered by launching a particular attack type, a, against the software. Let all attack types be represented by A, where $a \in A$. Then the total, TC_a, of all such discovered vulnerabilities is $TC_a = \sum_{a \in A} CV_a$. This paper is presenting a generic metric based on the "cross-site scripting (XSS)" and "Injection flaw"

examples given in Nichols and Peterson, where a known attack (such as XSS or bad input) can be used in penetration testing (see Section 5.1) to discover vulnerabilities. Increasing CV_a is bad and decreasing CV_a is better.

- *Lines of code (LOC) and LOC^2 - LOC* is a simple count of the lines of code in a given language. McGraw (2006) states that, "more code, more bugs, more security problems." There is a strong empirical correlation between *LOC* squared, LOC^2, and the number of incidents logged against fielded software (McGraw, 2006). An organization can use LOC^2 to get a feel for attack-ability. Note that defect density (*number-of-defects/LOC*) combined with vulnerability discovery models presented by Alhazmi et al. (2005) could provide estimates of future exploitation. To make such future estimates, the development team would have to track the number of vulnerabilities discovered and fixed via the penetration test/fix cycle.

Note that on further consideration, these metrics spark additional practical implications for design and implementation. For example, consider the *PVI* metric. The *PVI* metric points to a design architecture in which all input passes through a shared validation filter. Such an architecture could make the task of carrying out the actual counting much easier as each input handler could be checked for the presence of (or connection to) the shared validation filter. In addition, Throw-away error count (C_{tae}), suggests the importance of a coding standard and the application of code reviews during implementation.

With the collection and use of each metric, users must be careful about what the metric actually indicates. Fairly obvious, the above metrics are useful for relative comparison. Less obvious is what they assure or don't assure. Most of the metrics indicate that the security posture of the software is getting more or less "worse." For example, note the use of the word "better" instead of "good" for the CV_a metric. This word choice is made under the assumption that canned black-box tests will be used to launch the attack, discover the vulnerability, and subsequently increment CV_a. The "better" versus "good" word choice intentionally reflects the notion that canned black-box tests can only indicate relative levels of decreasing "badness," after McGraw (2006).

5.3. Culture Shift via Top-down Education

A practical assumption with respect to the application of the afore-mentioned SSE best practices and measurement collection is that the organization's culture is security conscious. If trying to launch a new SSE initiative, the first task is to gauge the organization's position within the "culture spectrum" as it relates to security and take actions to shift the culture in a more security conscience direction if needed. Increasing team member and management awareness through education and the presentation of statistical data serves as the primary vehicle enabling such a culture shift. Pointing key people at information such as that contained in Section 2 (Overview) and Section 3.1 (Vulnerability Origin) would provide a starting point for further discussion. As discussed in section 2, the whole notion of SSE, achieving better security assurance by keeping the software free of exploit-enabling vulnerabilities may be a new concept to many. Also, depending on corporate culture and typical software attributes desired, it may be that security is treated as a second-class citizen. That is, under time-to-market pressures to deliver feature complete software, it may be standard practice to relegate (often nonfunctional) security attributes to the wayside. Relegating security attributes to the wayside most likely represents the worst end of the culture spectrum. No matter where on the security-culture-spectrum an organization falls, any change must start with top management. Ultimately, top management sets the organizational tone and provides the go-ahead to spend time and money on security related development activities. Therefore, top management needs to be convinced that it is in the best interests of the organization to work SSE practices into the current SDLC.

To elicit top management support for secure development initiatives, one should present the financial benefits (e.g., relative cost savings and return on investment) as mentioned earlier in Section 1.1. Any data that can be gathered on similar products actively being attacked or known threats to the in-development software should also be presented to top management for their consideration.

6. CONCLUSIONS

This paper has presented a high-level overview of SSE in an effort to answer practical questions relating to the field. Preventing the introduction of vulnerabilities prior to release, rather than patching vulnerabilities

afterwards is THE challenge SSE rises to meet. Outside the context of formal methods and automated theorem provers (which tend to require significant investment and expertise), proving that a typical software product is secure requires testing for negative (e.g., there are no vulnerabilities). Since testing for a negative can entail infinite testing time, it is important to build security in from the beginning and to make cost-effective decisions regarding when to stop testing.

A key question of overarching practical consideration is how to go about the vulnerability prevention task in a cost-effective manner. This consideration is not new to SSE and is addressed by risk analysis in general, which is predicated on the fact that there is some reasonable level of preplanning and up-front expenditure required to obtain security assurance. Generally speaking, risk analysis must weigh the up-front costs of protections for vulnerabilities against the probability that attackers will target those vulnerabilities. The more accurate the risk analysis, the better will be the contingent business decisions. This paper discussed aspects of risk analysis from pre-existing fields of study, namely software engineering and computer/network security, and gave an example showing how the concepts translate to software.

Performing both vulnerability prevention and risk assessment better relies on fundamentally agreeing on classifications and models tailored for the SSE domain. Efforts such as the CWE provide the groundwork and common language needed to share information and compare testing coverage of automated testing and static analysis tools. Although the current focus of the CWE is on supporting tools for vulnerability prevention, the CWE can also be used to support risk assessment efforts. In particular, the concept of using real vulnerability data from fielded products with similar features to more quickly carry out risk assessment was discussed. Additional research, tailored for software, into models for better expressing and reasoning about threats, attacks, and vulnerabilities should likewise benefit prerelease efforts. While much work remains to be done, there are current approaches and metrics that can be used starting today as we work toward a better tomorrow.

REFERENCES

Ahmad, D. (2007). The contemporary software security landscape. *IEEE Security and Privacy*, 5(3), 75–77.

Ihazmi, O., Malaiya, Y., and Ray, I. (2005, August). Security vulnerabilities in software systems: A quantitative perspective. *Data and Applications Security XIX*, 3654, 281–294.

Anderson, R. (2001). Why information security is hard—An economic perspective. In *17th Annual Computer Security Applications Conference (ACSAC'01)*, pages 358–365, Los Alamitos, CA.

Anderson, R. (2002). Security in open versus closed systems — The dance of Boltzmann, Coase and Moore. Technical report, Cambridge University, England.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions Dependable and Secure Computing*, 1(1), 11–33.

Barnum, S. and Sethi, A. (2006). Introduction to attack patterns. US-CERT: Build security in Web site. Citigal. Available at https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/585.html

Bellovin, S. S. (2006). On the brittleness of software and the infeasibility of security metrics. *IEEE Security and Privacy*, 4(4), 96.

Beznosov, K. and Kruchten, P. (2004). Towards agile security assurance. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, pages 47–54, New York: ACM Press.

Boehmand, B. W. and Basili, V. R. (2001). Software defect reduction top 10 list. *IEEE Computer*, 34(1),135–137.

Buyens, K., De Win, B., and Joosen, W. (2007). Empirical and statistical analysis of risk analysis-driven techniques for threat management. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 1034–1041, Los Alamitos, CA. IEEE Computer Society.

MITRE Corporation. (2007, September). About CWE. Technical report. Available at http://cwe.mitre.org/about/index.html

MITRE Corporation. (2007). CWE List (Draft 7). Technical report. Available at http://cwe.mitre.org/data/index.html

Geer, D., Hoo, K.-S., and Jaquith, A. (2003). Information security: Why the future belongs to the quants. *IEEE Security and Privacy*, 1(4), 24–32.

Davis, N., Humphrey, W., Redwine,S. T., Jr., Zibulski, G., and McGraw. G. (2004). Processes for producing secure software: Summary of us national cyber-security summit subgroup report. *IEEE Security and Privacy*, 2(3), 18–25.

Essafi, M., Labed, L., and Ghezala, H. B. (2006). ASASI: An environment for addressing software application security issues. In *International Conference on Systems and Networks Communications*, p. 19, Los Alamitos, CA. IEEE Computer Society.

Firesmith, D. G. (2007). Engineering safety and security related requirements for software intensive systems. In *29th International Conference on Software Engineering (ICSE'07Companion)*, p. 169, Los Alamitos, CA. IEEE Computer Society.

Garcia, A. and Horowitz, B. (2007, February). The potential for underinvestment in Internet security: Implications for regulatory policy. *Journal of Regulatory Economics*, 31(1), 37–55.

Gregoire, J., Buyens, K., De Win, B., Scandariato, R., and Joosen, W. (2007). On the secure software development process: CLASP and SDL compared. In *SESS'07: Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, p. 1, Washington, DC, IEEE Computer Society.

Hoglund, G. and McGraw, G. (2002). Point/counterpoint. *IEEE Software*, 19(6), 56–59.

Hoglund, G. and McGraw, G. (2004). *Exploiting software: How to break code*. Boston, MA: Addison-Wesley, pp. 1–23, 37–70.

Hoo, K.-S., Sudbury, A. W., and Jaquith, A. R. (2001, Fourth Quarter). Tangible ROI through secure software engineering. *Secure Business Quarterly*, 1(2).

Howard, M., LeBlanc, D., and Viega, J. (2005). *19 deadly sins of software security*. New York: McGraw-Hill/Osborne.

Jones, J. R. (2007). Estimating software vulnerabilities. *IEEE Security and Privacy*, 5(4), 28–32.

Krsul, I. V. (1998, May). *Software vulnerability analysis*. PhD thesis, Purdue University, West Lafayette, IN.

Landwehr, C. E., Bull, A. R., McDermott, J. P., and Choi, W. S. (1994). A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3), 211–254.

Leavitt, N. (2005). Will proposed standards make mobile phones more secure? *Computer*, 38(12), 20–22.

Manadhata, P. and Wing, J. M. (2005, July). An attack surface metric. Technical Report CMU–CS–05–155, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Martin, R. A. and Barnum, S. (2008). A status update: The common weaknesses enumeration. Technical report, MITRE Corporation. Available at http://cwe.mitre.org/documents/cwe_update.pdf

Martin, R. A., Christey, S. M., and Jarzombek, J. (2005, November). The case for common flaw enumeration. *NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics*.

McGraw, G. (1999). Software assurance for security. *Computer*, 32(4), 103–105.

McGraw, G. (2003). From the ground up: The DIMACS software security workshop. *IEEE Security and Privacy*, 1(2),59–66.

McGraw, G. (2004). Software security. *IEEE Security and Privacy*, 2(2), 80–83.

McGraw, G. (2006). *Software security: Building security. In*: Addison-Wesley Software Security Series. Upper Saddle River, NJ: Addison-Wesley, pp.13–37, 277–298.

McGraw, G. and Potter, B. (2004). Software security testing. *IEEE Security and Privacy*, 2(5), 81–85,.

Nichols, E. A. and Peterson, G. (2007). A metrics framework to drive application security improvement. *IEEE Security and Privacy*, 5(2), 88–91.

NIST. National Vulnerability Database (NVD). (2007). Technical report, National Institute of Standards and Technology. Retrieved January 12, 2009 from http://nvd.nist. gov/statistics.cfm

OWASP - The Open Web Application Security Project. (2007). *OWASP top ten: The ten most critical Web application security vulnerabilities*, 2007 update edition.

Panko, R. R. (2003). *Corporate computer and network security*. Upper Saddle River, NJ: Prentice Hall, pp. 324–330.

Peine, H. (2005). Rules of thumb for secure software engineering. In: *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*. ACM Press: New York, pp. 702–703.

Popek, G. P. and Kline, C. S. (1979). Encryption and secure computer networks. *ACM Computing Surveys*, 11(4), 331–356.

Common Criteria Portal. CC-Common Criteria. (2007). Technical report. Available at http://www.commoncriteriaportal.org/public/developer/index.php?menu=2

Sachitano, A., Chapman, R. O., and Hamilton, J. A. (2004). Security in software architecture: A case study. In: *Information Assurance Workshop*, Proceedings of the Fifth Annual IEEE SMC, June, pages 370–376.

Schneidewind, N. F. (2002). Body of knowledge for software quality measurement. *Computer*, 35(2), 77–83.

Sindre, G., and Opdahl, A. L. (2005, January). Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1), 34–44.

Kim, Y., Park, G., Kim, T., and Lee, S. (2007). Security evaluation for information assurance. In: *The 2007 International Conference on Computational Science and its Applications*, pp 227–230, Los Alamitos, CA, IEEE Computer Society.

Tsipenyuk, K., Chess, B., and McGraw, G. (2005). Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security and Privacy*, 3(6), 81–84.

US-CERT. US-CERT Vulnerability Notes Database. (2007). Technical report, US-CERT: United States Computer Emergency Readiness Team. Retrieved January 12, 2009 from http://www.kb.cert.org/vuls/

van Wyk, K. R. and McGraw, G. (2005). Bridging the gap between software development and information security. *IEEE Security and Privacy*, 3(5), 75–79.

van Wyk, K. R. and Steven, J. (2006). Essential factors for successful software security awareness training. *IEEE Security and Privacy*, 4(5),80–83.

Verdon, D. and McGraw, G. (2004). Risk analysis in software design. *IEEE Security and Privacy*, 2(4), 79–84.

Viega, J., Bloch, J. T., Kohno, T., and McGraw, G. (2002). Token-based scanning of source code for security problems. *ACM Transactions on Information Systems Security*, 5(3), 238–261.

Viega, J., Bloch, J. T., Kohno, T., and McGraw, G. (2000). ITS4: A static vulnerability scanner for C and C++ code. Technical report, Citigal, Dulles, VA. Available at http://www.cigital.com/papers/download/its4.pdf

Viega, J. and McGraw, G. (2001, September 24). *Building secure software: How to avoid security problems the right way*. Addison-Wesley Professional, 1st edition. pp. 1–6.

Wing, J. M. (2003). A call to action: Look beyond the horizon. *IEEE Security and Privacy*, 1(6). 62–67.

Xie, N., Mead, N., Chen, P., Dean, M., Lopez, L., Ojoko-Adams, D., and Osman, H. (2004). Square project: Cost/bene?t analysis framework for information security improvement projects in small companies. Technical Report (CMU/SEI–2004–TN–045, ADA431118), Software Engineering Institute, Carnegie Mellon University Pittsburgh, PA. Available at http://www.sei.cmu.edu/publications/documents/04.reports/04tn045.html

Zitser, M., Lippmann, R., and Leek, T. (2004). Testing static analysis tools using exploitable buffer over?ows from open source code. In: *SIGSOFT '04/FSE–12: Proceedings of the 12th ACMSIGSOFT International Symposium on Foundations of Software Engineering*, ACM Press, New York, pp 97–106.