
Application Security Fundamentals

By  **CONSCIERE**
ACT WITH KNOWLEDGE

Abstract: As the importance of security continues to dawn on the Software Industry 2.0, organizations of all sizes are trying to discover what constitutes software security “due care” for their customers. This brief paper will review key principles surrounding security in the development lifecycle (SDL), covering a prototypical SDL model, what we’ve seen work/not work with real-world SDL implementations, economic drivers, and industry benchmarks (or the lack thereof).

Few question that software occupies an increasingly central role in our everyday lives. From computer operating systems and applications, to mobile phones, television, the Internet, VoIP, GPS navigation, software-driven medical systems, air traffic control, the electric grid, and so on, human activity (and perhaps even human existence itself) has come to rely heavily on software.

But is that reliance justified? As more and more of our lives become digitized, and headlines have begun to trumpet the growth of malicious hacking and other incidents of cyber abuse, deep questions surrounding the confidentiality, integrity, and availability¹ of digital data have been raised. Are the risks underlying this brave new world greater than the rewards?

In parallel, software development organizations of all sizes are trying to discover how to protect their end users from unreasonable risks. Of course, this raises yet another question: what constitutes “reasonable”? Put another way, what is the standard of software security “due care”?

To date, the software industry has adopted the following fundamental approaches to establishing this standard:

1. Ignore
2. React
3. Prevent

Let’s examine each of these approaches briefly to set the stage for a deeper discussion of security in the software development lifecycle (SDL).

¹ Confidentiality, integrity, and availability (CIA) are often cited as the defining properties of information security. Some authorities also include a fourth “A” for “accountability,” typically understood to refer to the keeping of tamper-resistant activity logs to provide non-repudiation.

Ignorance is Bliss

The dirty little non-secret of the technology industry is that few software development-oriented companies are doing anything serious about security. Recent surveys suggest that, despite some uptake of outsourcing and tools, most firms do not allocate significant budget or headcount for application security outside of standard operational IT security processes.¹ Although some in the information security industry would bristle at the implication, the question remains: Is ignoring the problem simply good risk management?

Data on security incidents or breaches has historically not been tracked systematically (with some recent exceptions, albeit focused on operational breaches rather than purely software vulnerability-related²). There is good news and bad news in this recent data: while only 6 out of 90 confirmed breaches (derived from over 150 cases) “...resulted from an attack exploiting a patchable vulnerability... the bulk of attacks continues to target applications and services rather than the operating systems or platforms on which they run.” (ibid) So, if you’re a major operating system vendor, take heart, but if you’re writing custom application code, you’re increasingly the target of attack.

Another quasi-informative dataset is the National Vulnerability Database (NVD), which tracks advisories on software vendor bulletins.³ Figure 1 shows the raw count of vulnerabilities tracked in NVD, across all vendors and products, including software flaws (not configuration flaws), across all sources, from 1997 to 2009.

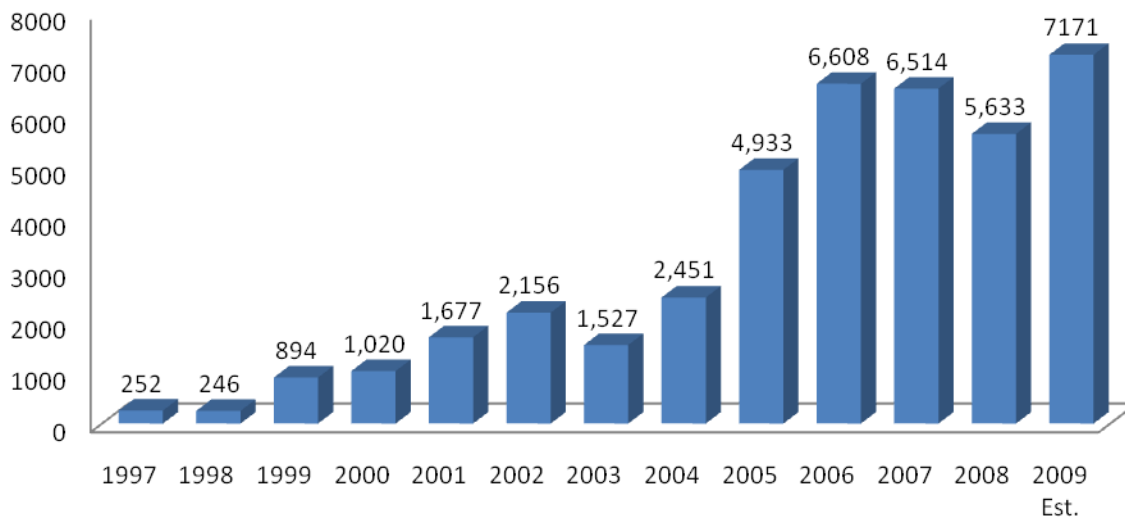


Figure 1: Software vulnerabilities released between 1997 and 2009 (extrapolated), courtesy of National Vulnerability Database

Clearly, the number of visible vulnerabilities is on the rise. Presumptively, based on unrelated studies showing software sales to be dominated by a handful of vendors, most of these vulnerabilities emanate from the same group of vendors. (We haven’t done research confirming this.) Given all this activity, does it make sense simply to “vanish in the noise” if you are a small or medium sized software shop, and simply invest minimally in, say, outsourced professional security review during the release cycle?

What is the return on investment (ROI) for security in the development lifecycle? We'll return to this question later, but will simply note at this point that there is a robust world-wide research community waiting for you out there.

To close this discussion of the merits of ignoring the software security problem, although it seems counterintuitive to assert positive outcomes from such a mindset, we've found minimal data to quantify the penalties of such an approach for small- to mid-sized software development efforts. Larger-scale development organizations with widely-deployed products are a different story, and anecdotal evidence exists to support investment in security, a topic we'll return to later. Next, we'll examine the other two postures, reactive and preventive.

React vs. Prevent

Many software firms have observed real-world security quality improvements resulting from external security review, and have hired penetration testers to assess their products, typically keeping the results private and selectively fixing some or all of the vulnerabilities. Although these practices can be laudable when performed in conjunction with other measures to be discussed momentarily, simply finding and fixing bugs iteratively between releases is not necessarily the most efficient way to increase code security quality. In fact, it's arguably less efficient than "learning to fish", in other words adapting a culture of prevention and the processes & technology to support it.

This is the heart of the justification for SDL. "Baking security in" rather than "bolting it on" in theory leads to better outcomes for all involved, including the development organization and its customers. Next, we'll describe in more detail the components of SDL and how it drives these outcomes.

SDL History and Philosophy

Of course, the notion of "baking security in" has been around for some time. Some of the "classic" antecedents of security in the development lifecycle include:

- NIST SP 800-64⁴
- BS7799/ISO17799/27001-2⁵
- OCTAVE⁶

In our opinion, ISO 17799 Sec 10 and ISO 27002 Sec 12 remain classics from an SDL policy perspective, including such fundamentals as separation of test and production environments, input/output validation, cryptography best practices, transaction integrity/non-repudiation, and so on.

More recent iterations of security in the development lifecycle frameworks include:

- Microsoft SDL⁷
- CLASP⁸
- BSI-MM⁹

- OpenSAMM¹⁰

Microsoft's SDL is among the most widely recognized currently, although there has been substantial recent attention for the other frameworks in this list (which share some overlaps in pedigree¹¹).

SDL Principles & Framework

Obviously, there are a number of approaches to security in the development lifecycle, going back several years. It is therefore more realistic to think of SDL as a framework, or set of principles, that specific organizations can adapt and customize to their own unique purposes. Even Microsoft's SDL documents refer to their "implementation" of SDL. Below we attempt to summarize some of the high-level principles of SDL that are common to many of the above-mentioned frameworks:

1. Distribute the work of assessment and remediation, especially to the development team
2. Independent (of the development team) reviews at key milestones
3. Provide relevant training and re-usable guidance (checklists)
4. Strive for quantitative risk management, and set thresholds
5. Leverage automation

The first point articulates the overall strategy of SDL: accountability for the security quality of software needs to reside primarily with the developers of the software. This creates incentives to make continuous improvements to security quality in the long term. Alternative accountability models, such as where the internal corporate security team takes responsibility for software security, don't scale well in our experience because of conflicting incentives between the business (release feature-rich software to customers) and risk management interests (ensure that security quality is high).

Of course, security assurance cannot be outsourced entirely to the development function, as that creates a "fox guarding the chicken coop" situation (i.e. lack of appropriate segregation of duties). So, point 2 notes that reviews conducted by (or overseen by) parties independent of the development team are necessary at key milestones. For example, the corporate security team could conduct pre-release penetration testing independently of the development team and track the remediation of identified issues.

Point 3 is perhaps self-evident, but nevertheless important: people have a hard time doing the right thing if they aren't told what's the right thing to do. Development team security training (including program managers, testers, and managers) is thus an important component of any SDL implementation. Training programs should provide job-relevant curricula, track comprehension (ideally linked to application on-the-job), and be supported by re-usable guidance, code libraries/routines, and checklists that developers can easily access on the job to enforce good behavior. Application security training could be the topic of an entirely separate discussion, so we'll say little else about it going forward other than to reiterate its importance to the success of the overall SDL effort.

Point 4 acknowledges that information security practices continue to evolve towards more mature, quantitative risk management approaches. These same principles are ideal to apply to software security assurance as well. For example, Microsoft's DREAD¹² risk rating system strives to quantify the severity of software vulnerabilities, and thus define the priority of remediation efforts. (DREAD is somewhat proprietary to Microsoft, but is illustrative of the concept of quantitative assessment; other risk quantification systems include CVSS2,¹³ FAIR,¹⁴ and FMEA¹⁵.) Beyond just the scoring system itself, it is important to establish thresholds for prioritization, or to put it colloquially, a "bug bar." The bug bar essentially defines for an organization the thresholds at which work will be done to remediate a flaw. It can be immensely helpful to define thoughtful thresholds like this in collaboration with all stakeholders in advance of performing assessments, to avoid disagreements over how to remediate flaws (resulting in delayed release, unacceptably risky flaws in released code, or both).

Point 5 needs little explanation. Automation yields greater efficiency, and SDL is no exception. Some key areas with high potential for improvement through automation include:

- Security code review (although the accuracy and relevance of output from common tools remains suspect)
- Fuzz testing (to be defined later)
- SDL process automation, e.g. self-help web portals for workflow management

We've included a brief overview of application security tools at the end of this article.

Pitfalls

We've covered some key SDL principles that can help improve the chance of good outcomes. Are there any practices that should be avoided?

Anecdotally, one of the main reasons for failure of an SDL initiative is lack of focus on benefits to the organization, and by extension, its customers. Many organizations approach SDL assuming that the implied virtues of more secure software will simply make it acceptable to all stakeholders. In addition, there is historical disagreement around whether focusing on return on investment (ROI) for security is worthwhile or achievable (we believe economic justification is imperative, and will return to this concept later). To counteract this tendency, we recommend developing a good "scouting report" on all stakeholders (especially customers!), how they perceive SDL, their key objectives, and expected performance indicators. Often, this basic research can point to simple and easily implemented initial steps that result in early wins, good momentum, and a strong head start towards a sustainable SDL program.

Culture shock can also torpedo SDL implementations. The culture of software development generally resists structure and discipline, and specific group dynamics can present even further challenges. Often, security is perceived from the start as an outsider, and the various behavioral changes proposed within the SDL initiative are thus viewed with suspicion at the outset. Be prepared to adapt your specific SDL

implementation to the general and specific culture of development within your organization to bypass culture shock and outsider perception out of the gate.

Those chartered with initiating SDL are often tempted to set unrealistic expectations for SDL outcomes in order to address the outsider perception issue. Obviously, this is not recommended. Software development cultures are often focused tightly on schedule and resource allocation, and individuals who mismanage those two fundamentals often sacrifice substantial reputational capital that is very difficult to re-acquire for subsequent release cycles.

Lack of alignment with other security initiatives can also introduce “audit fatigue” amongst developers, who typically bristle at being interrupted multiple times for what they perceive is the same issue. One of the typical examples here is web development shops that have to comply with PCI DSS.¹⁶ They are faced with complying with both SDL and PCI-related security initiatives separately if those programs are not well-coordinated.

Finally, and perhaps most importantly, organizational governance is often neglected in the design of SDL programs. The common wisdom is to “get executive buy-in” for initiatives of this nature, and this is of course important. However, development cultures are more often driven by “bottom-up” perceptions, so it’s important to consider lobbying of all stakeholders early and often.

SDL Implementation Examples

What does an SDL implementation look like in practice? As we’ve proposed, it should be well-aligned with the existing development rhythm and culture. Figure 2 shows a mock development lifecycle for a large enterprise.

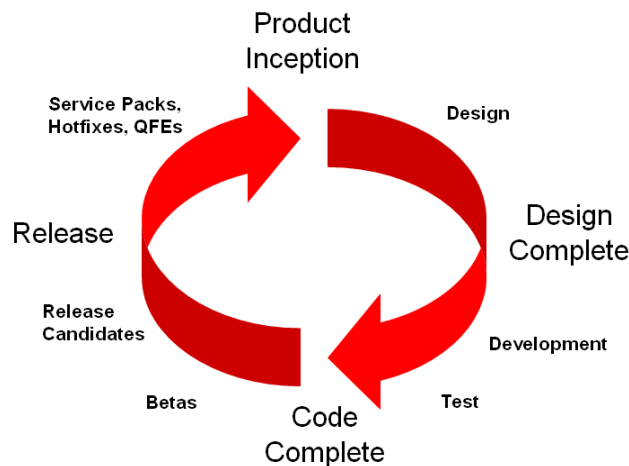


Figure 2: A mock large enterprise development lifecycle

Beginning with the lifecycle in Figure 2 as a baseline, in Figure 3 we overlay some common SDL milestones. This is one possible implementation for a large-scale organization with substantial resources.



Figure 3: A sample SDL implementation overlaid on top of the previously introduced mock development cycle.

It’s important to note how Figure 3 aligns with the SDL principles we articulated earlier:

Principle	Implementation in Figure 3
Distribute the work of assessment and remediation, especially to the development team	A Security Liaison is assigned at project inception, who will be accountable for managing security workflow throughout. ²
Independent review at key milestones	The red lines indicate milestones where independent review can occur. Note that these are closely aligned to existing development process gates.
Provide relevant training and re-usable guidance (checklists)	Training is an SDL gate that occurs early in the cycle. ³ Also, the “Build Standards” gate at the “Test” milestone illustrates an opportunity to provide re-usable checklists.

² Note that the security liaison manages workflow, not security *outcomes*, such as code security quality and other metrics. Ultimately, the development team leadership/executives are accountable for outcomes.

³ In practice, the number of developers who receive required training fluctuates between and within cycles, but the idea here is to enforce training early in a given cycle to ensure people have the training they require to do their jobs.

Principle	Implementation in Figure 3
Strive for quantitative risk management, and set thresholds	The iterative nature of the overlay cycle provides multiple opportunities to check metrics (such as DREAD score mitigation) during the current and in future releases.
Leverage automation	A number of gates could require automated checks, such as the “Security Testing” and code review milestones.

Although Figures 2 and 3 are illustrative of how SDL principles might be implemented on a large scale, we’ve stressed the importance of starting small with SDL and iteratively growing the program to a scale that is sustainable for a given organization. Figure 4 provides an example of a smaller-scale SDL implementation based on what we assert are the minimum components for success.

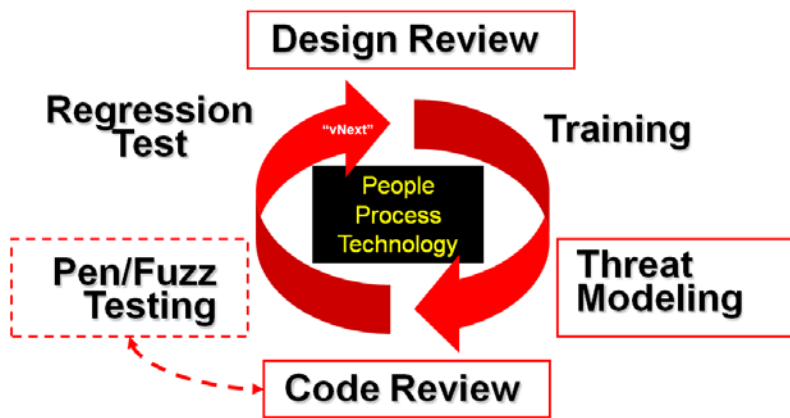


Figure 4: A light-weight SDL implementation example.

Note that many of the enterprise-scale SDL checkpoints (shown in Figure 3) have been eliminated in the example shown in Figure 4. The boxes highlighted in red in Figure 4 comprise an even more minimal SDL implementation made up of Design Review, Threat Modeling, and Code Review (with optional Penetration/Fuzz Testing). The terms shown in Figure 4 are defined in the Glossary at the end of this article.

Development Infrastructure

It’s worthwhile to pause for a moment to highlight the importance of basic fundamental software development hygiene. Of course, many opinions exist (especially within development communities) about the exact meaning of “basic software development fundamentals,” and we’re not interested in starting such a debate here. Our primary point is that much of the theory and practice of SDL depend upon certain fundamentals being in place, and most SDL initiatives will not be successful without at least

some structure to which it can be anchored. Some of the common key enablers of successful SDL implementations are listed in Table 1.

Fundamental Dev Practice	Assists SDL By
Consistent test, build environments	Separating test and production data, ensuring expected run-time parameters
Concurrent versions system (CVS)	Enforcing known-state versioning and providing reversion capability
Defect management system	Provides a central repository for managing and measuring defect reduction
Reporting	Provides consistent communication of data to appropriate decision-makers

Table 1: Development practice fundamentals that enable SDL.

What About Alternative Programming Models?

The SDL models we've described so far all align with classic "waterfall" development processes. But this raises a popular question: Can SDL be applied successfully to iterative/unstructured Agile programming methods like Scrum and Extreme Programming? Absolutely,¹⁷ but there is a point at which a lack of structure or too much "adaptive-ness" can hamper SDL. SDL is based on the premise that a minimal level of structure exists to which systematic evaluations of security quality can be anchored. If there is no structure to which anything can be anchored, then SDL will likely be challenging to implement. This again highlights the challenge of cultural integration for those without substantial development experience (e.g. security professionals attempting to implement SDL): it can be hard to differentiate natural resistance to change from active attempts to cover up an overall poorly managed existing development effort.

SDL ROI

To this point, we've discussed the "what" and some of the "how" of SDL. Before concluding, we'll take a step back and briefly survey the "why." Our basic premise is that, at least in business scenarios, the key driver of SDL should be *economics*, with emphasis on the "should be". Finding and interpreting data to support this contention is challenging.

Generally, there is a lack of systemic empirical evidence supporting measurable economic outcomes following implementation of SDL. Most studies to date have focused on hard operation costs yielding intangible benefits. For example:

- Savings-to-cost ratios ranging from break-even to 8 or 9 times¹⁸
- Average loss of 0.76% market value when a vulnerability is disclosed¹⁹
- Return on investment equated to 21%²⁰

Other studies have shown the value of good design, user education, and automated response technology over finding and fixing bugs.^{21,22}

Microsoft has published data showing a sharp reduction in security bulletins published from Windows 2000 Server to Windows Server 2003.²³ Combining this data with separate claims by Microsoft that a security bulletin costs the company approximately \$100,000 (in 2002 dollars)²⁴, one can impute that Microsoft saved approximately \$3.7M due to their "Secure Windows Initiative" push that was one of the primary progenitors of their brand of SDL. Microsoft publishes more recent data on vulnerability count (but not economic metrics) across various product lines on its SDL portal page.²⁵ Of course, this analysis does not quantify tangible investment in SDL beforehand (let alone even as a percentage of overall spend); it just illustrates benefits in terms of hypothetically realized savings from un-issued bulletins.

Admittedly, this brief review of economic data in support of SDL has not done justice to the topic. Our sense based on anecdotal experience is that, like most things, the ideal risk/reward balance is not one-size-fits-all, but is rather best gleaned from experimentation and keen focus on tying SDL metrics to economic outcomes early and often. To end with one final piece of guidance on quantitative data supporting SDL, we paraphrase the Pareto Principle: invest more in finding the “vital few” issues that cause the vast bulk of security vulnerabilities. We’ve provided one last table to help illustrate this point:

	Budget	Quantity Found	Quality Impact
React <ul style="list-style-type: none">• Find & fix bugs• Bolt-on	20%	80%	20%
Prevent <ul style="list-style-type: none">• Improve dev practices• Baked-in	80%	20%	80%

Table 2: Planning for SDL should focus on finding the "vital few" issues that cause the preponderance of security vulnerabilities.

About The Author



Joel Scambray, CISSP, is co-founder and CEO of [Consciere](http://www.consciere.com) (www.consciere.com), provider of strategic security advisory services. He has assisted organizations ranging from small startups to Microsoft Corp. address information security challenges and opportunities for over a dozen years, in diverse roles including consultant, corporate leader, entrepreneur, and co-author of the *Hacking Exposed* book series.

The author wishes to acknowledge important contributions to this article from Andre Gironda of tssci security; and Birgit Lahti, Kevin Nassery, and Kevin Rich of Consciere.

Glossary

Below we've defined some of the terms related to SDL discussed in this article. We've ordered the definitions chronologically according to the lifecycle pictured in Figure 4.

Design Review: Development staff interviews and documentation review to clarify business & security objectives, architecture, design assumptions

Threat Modeling: Structured interaction with development staff to document a prioritized list of security threats according to industry standard practices (e.g. Microsoft, Trike)

Penetration Testing: "Adversarial use" to scrub near-final application for known security vulnerabilities using common tools and techniques

Fuzz Testing: Automated input testing of instrumented components/protocols to identify potentially exploitable boundary conditions

Code Review: Analysis of relevant application source/binary code for known security vulnerabilities

Infrastructure Review: For software services, security assessment of supporting IT infrastructure and analysis of key integration points

Tools & Guidance

Below, we've listed some tools and guidance commonly employed within SDL. This is a brief list, and is not intended to be comprehensive.

Category	Tool/Reference
Threat Modeling	<p>Microsoft Threat Modeling (http://msdn2.microsoft.com/en-us/security/Aa570411.aspx)</p> <p>Trike Threat Modeling (http://www.octotrike.org/)</p>
Code Review	<p>OWASP Code Review Project (http://www.owasp.org/index.php/Category:OWASP_Code_Review_Project)</p> <p>Coverity (www.coverity.com)</p> <p>Fortify (www.fortify.com)</p> <p>Ounce Labs (www.ouncelabs.com)</p> <p>Veracode (http://www.veracode.com/)</p>
Fuzz Testing	<p>Peach Fuzzer Platform (http://peachfuzzer.com/)</p>

Category	Tool/Reference
	Mu Dynamics (http://www.mudynamics.com/)
Web App Security Scanners	http://sectools.org/web-scanners.html
.NET Input/Output Validation/Encoding	Microsoft Anti-XSS 3.0 Beta (http://www.microsoft.com/downloads/details.aspx?FamilyId=051ee83c-5ccf-48ed-8463-02f56a6bfc09&displaylang=en) Microsoft Code Analysis Tool (CAT.NET); http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en
General Guidance and Tools	The Open Web Application Security Project (OWASP), http://www.owasp.org

References

¹ OWASP Security Spending Benchmarks Project Report, March 2009.

http://www.owasp.org/images/b/b2/OWASP_SSB_Project_Report_March_2009.pdf

² 2009 Data Breach Investigations Report, a study conducted by the Verizon Business RISK team.

<http://securityblog.verizonbusiness.com>

³ National Vulnerability Database statistics. <http://web.nvd.nist.gov/view/vuln/statistics>

⁴ NIST SP 800-64 Rev. 2, Oct 2008, Security Considerations in the System Development Life Cycle.

<http://csrc.nist.gov/publications/PubsSPs.html>

⁵ ISO 27000 Series. http://en.wikipedia.org/wiki/ISO/IEC_27000-series

⁶ OCTAVE (Operationally Critical Threat, Asset, and Vulnerability Evaluation). <http://www.cert.org/octave/>

⁷ The Microsoft Security Development Lifecycle (SDL). <http://msdn.microsoft.com/en-us/security/cc448177.aspx>

⁸ CLASP (Comprehensive, Lightweight Application Security Process).

http://www.owasp.org/index.php/Category:OWASP_CLASP_Project

⁹ The Building Security In Maturity Model. <http://www.bsi-mm.com/>

¹⁰ Open Software Assurance Maturity Model (OpenSAMM). <http://www.opensamm.org/>

-
- ¹¹ “OpenSamm shows a way.” Real Software blog, Jim Rice, April 17, 2009. <http://swreflections.blogspot.com/2009/04/opensamm-shows-way.html>
- ¹² Microsoft’s DREAD risk calculation system. <http://msdn.microsoft.com/en-us/library/aa302419.aspx>
- ¹³ Common Vulnerability Scoring System v. 2.0. <http://www.first.org/cvss/cvss-guide.html>
- ¹⁴ Factor Analysis of Information Risk (FAIR). <http://fairwiki.riskmanagementinsight.com/>
- ¹⁵ Failure mode and effects analysis (FMEA). http://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis
- ¹⁶ Payment Card Industry Data Security Standard (PCI DSS). <https://www.pcisecuritystandards.org/>
- ¹⁷ “Streamline Security Practices For Agile Development.” Bryan Sullivan. <http://msdn.microsoft.com/en-us/magazine/dd153756.aspx>
- ¹⁸ “Calculating Security Return on Investment,” Don O’Neill. <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/business/677.html>
- ¹⁹ Telang, Rahul & Wattal, Sunil. “Effect of Vulnerability Disclosures on Market Value of Software Vendors – An Event Study Analysis.” Workshop on Information Systems and Economics, Washington, DC, 2004. <http://opim-sun.wharton.upenn.edu/wise2004/sat622.pdf>
- ²⁰ Soo Hoo, Kevin; Sadbury, A. W.; & Jaquith, A. R. “Return on Security Investments.” *Secure Business Quarterly* 1, 2 (2001).
- ²¹ Rescorla, E. “Is Finding Security Holes a Good Idea?” The Third Workshop on Economics and Information Security. Minneapolis, MN, 2004. <http://www.dtc.umn.edu/weis2004/rescorla.pdf>
- ²² http://en.wikipedia.org/wiki/Windows_Error_Reporting
- ²³ The Changing Threat Environment, Aucsmith, WinHEC 2005. http://download.microsoft.com/download/9/8/f/98f3fe47-dfc3-4e74-92a3-088782200fe7/TWWI05021_WinHEC05.ppt. Slide 36 contains a graph showing “Critical” and “Important” security bulletins released for Windows 200 Server and Windows Server 2003.
- ²⁴ *Writing Secure Code*. Howard & LeBlanc, 2002.
- ²⁵ The Microsoft Security Development Lifecycle (SDL): Measurable Security Improvements. <http://msdn.microsoft.com/en-us/security/cc424866.aspx>