2009

Security Requirements Engineering-The Reluctant Oxymoron

Michael N. Johnstone
Edith Cowan University

DOI: 10.4225/75/57b4011e30de8
Originally published in the Proceedings of the 7th Australian Information Security Management Conference, Perth, Western Australia, 1st to 3rd December 2009.

Security Requirements Engineering-The Reluctant Oxymoron

Michael N. Johnstone
School of Computer and Security Science
Edith Cowan University
Perth, Western Australia
Email: m.johnstone@ecu.edu.au

Abstract

Security is a focus in many systems that are developed today, yet this aspect of systems development is often relegated when the shipping date for a software product looms. This leads to problems post-implementation in terms of patches required to fix security defects or vulnerabilities. A simplistic answer is that if the code was correct in the first instance, then vulnerabilities would not exist. The reality of a complex software artefact is however, driven by other concerns. Rather than probing programs for coding errors that lead to vulnerabilities, it is perhaps more beneficial to look at the root causes of how and why vulnerabilities come to exist in software. This paper explores the reasons why this might be so, uses two simple case studies to illustrate the effects of failing to specify requirements correctly and suggests that software development methods that build in security concerns at the beginning of a project might be the way forward.

Keywords

Security, Vulnerability, Software Engineering, Requirements Engineering, Conceptual Modelling.

INTRODUCTION

A vulnerability, according to RFC 2828 (2000, p190) is "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.". An exploit is, therefore, an attack that takes advantage of a vulnerability thus realising a threat. Before examining the issues in modern software development that cause concern in the security community, it is instructive to be reminded of the scope of the exploit problem. The security landscape is looking grim. Symantec (2008) indicate that since 2005 there has been a significant increase in the number of security threats reported (see figure 1). Admittedly, it is acknowledged that this figure shows threats reported, not total threats or actual successfully exploited systems, but it can be assumed to be a reasonable indicator of the growth of the problem. Cross-validation with other sources (viz. CERT, 2009; OECD, 2008) suggests that the Symantec statistics are accurate. Further, it is accepted that such statistics may be somewhat skewed to favour more "socially acceptable" threats as the public dissemination of a problem is not always considered to be in the best interests of a private firm. Nonetheless, the shape of the curve suggested by the figures is worrying and the sheer volume of new threats calls into question the ability of software developers to ameliorate or arrest the problem. Finally, Malware as exploit code includes viruses, worms, trojans and bots and it is likely that variants of these exploits make up a significant portion of the reported statistics. NIST (2009), however, provide data specific to software flaws (CVEs) and whilst the numbers are not of the same magnitude (cf. 246 and 5, 632 CVEs in 1998 and 2008 respectively), the trend and curve shape is similar to that reported by Symantec.

A recent OECD (2008) report notes that available data on malware exploits is derived from a variety of sources; national computer response teams (e.g. AusCERT), software vendors (e.g. Microsoft), and security vendors (e.g. Symantec) and therefore cannot easily be compared. The OECD note, however, that "…it is more or less possible to highlight certain tendencies that seem to be shared: i) an [sic] significant and noticeable rise in security incidents related to malware ; and, ii) trojan malware becoming more and more prevalent when looking across types of malware.".

Shostack and Stewart (2008) assert that most software is insecure and provide several reasons why this is so, the most interesting being that "because security is difficult for prospective customers to evaluate, it is rarely prioritized above other factors in their purchasing process" (Shostack and Stewart, 2008, p89). This is further compounded by security requirements being omitted from requirements specifications altogether as noted by Wysopal et al. (2007), a topic that will be discussed later in this paper. Anderson (2001, p7) was far more direct when he said "Much has been written on the failure of information security mechanisms to protect end users from privacy violations and fraud. This misses the point. The real driving forces behind security system design usually have nothing to do with such altruistic goals. They are much more likely to be the desire to grab a monopoly, to charge different prices to different users for essentially the same service, and to dump risk. Often this is perfectly rational.".

Having highlighted the scope of the problem faced by users of software, this paper now examines the reasons why exploits exist by looking at what software engineers do to gather requirements for software systems and why (using two case studies) those approaches fall short in the area of security requirements. Briefly, the key issues are first, the

overwhelming complexity of software systems; and second, the (relative) immaturity of software engineering as a discipline.

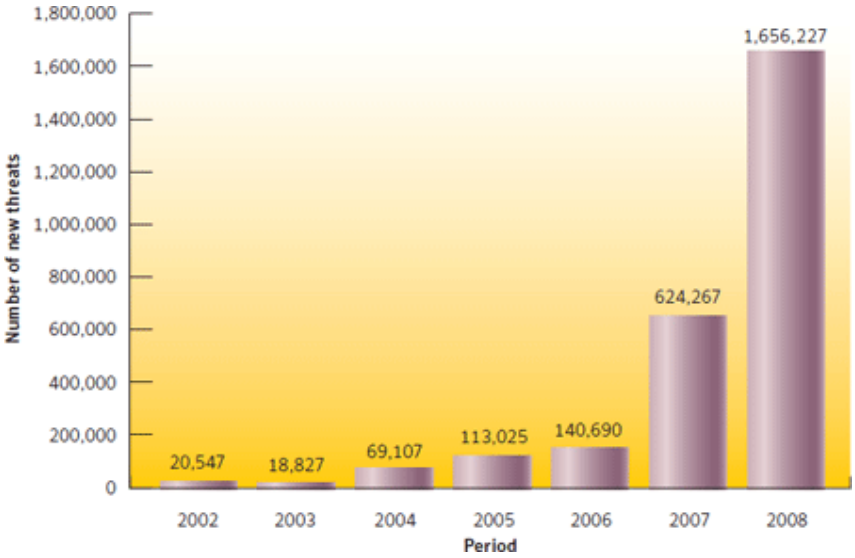

Figure 1 - New malicious code signatures (Symantec, 2008)

REQUIREMENTS ENGINEERING ISSUES

It has been suggested that not enough attention is placed on security requirements, so much so that if they are considered at all, it is in a fairly ad hoc manner (Viega, 2005; Mead and Stehney, 2005). Software Engineers are, however, aware of the high cost of failure in building systems-especially in the early stages of gathering requirements. Many reports have been published that cite poor requirements determination as one of the top three causes of IT project failure, the CHAOS Report being probably the most well-known (Standish, 2003). Therefore, there it is not surprising that both in academia and in practice, significant resources are focussed on requirements elicitation, specification and validation. Figure 2 shows a common model of requirements determination.

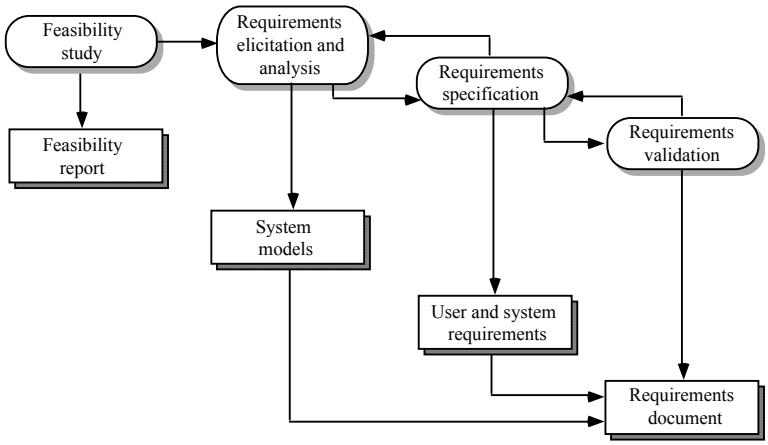

Figure 2 - The Requirements Engineering Process (adapted from Sommerville, 2001, p55)

Usually, requirements are split into two general classes: functional requirements, which represent what the target system is meant to do or how it is intended to behave from the point of view of an external actor; and non-functional requirements, which detail aspects of a system that, while they do not affect whether a function can be delivered or not, may have a significant effect on performance or quality. Traditionally, security requirements fall into the latter class (as evidenced by IEEE Standard 830 for software requirements specifications), but, as will be seen later, can (and should be) considered as functional requirements.

The general problem faced by software engineers is twofold. First, reality (and therefore any software system operating in the real world) is complex. This leads to models that are necessarily simpler than the reality that they represent in

order to provide some means of discussing complex problems but the models are incomplete and are only models-as Korzybski (1936) succinctly said "the map is not the territory". This leads to the second point-it is crucial to capture the essence of the problem and to dismiss the dross (a point well-made by Brooks, 1987). The difficulty lies in knowing which is which, especially in a discipline that is approximately only 40 years old-something of a contrast with more established forms of engineering such as civil or mechanical engineering.

The complexity of software systems is a concern as quality (and security) requirements may be set aside as a delivery date approaches. Since the development of the "modern" computer (circa 1940), program code complexity has been increasing at roughly an order of magnitude every decade. The systems to control NASA's Space Shuttle contain in the region of 40M lines of code, which is certainly a large software system. Similarly, Windows Vista contains 50M lines of code (Cusumano, 2006). Surprisingly, the largest software system currently in use is not an avionics or business system. Charette (2009) reports that high-end motor vehicles contain 70-100 microprocessors running almost 100M lines of code.

Given that such software systems are complex, the solution does not appear to lie in following outmoded, prescriptive development methods. A decade ago, Lyytinen et al. (1998) conducted a field study of leading edge (web application) development firms and found that the timeline for development drastically shortened compared to prior work to meet consumer demands and significantly that the new knowledge (required to successfully build such systems) is unprecedented in both depth and breadth. These pressures resulted in the firms surveyed now settling for adequate instead of optimal solutions even after spending more on staff education and training. The same firms also chose to eliminate existing (rigid) software development methods because timelines made them unusable and obsolete. It is unlikely that anything has changed in the last ten years. The OECD (2008) recently highlighted the pressures put on software developers and stated: "Software developers typically face difficult development trade-offs between security, openness of software as a platform, user friendliness, and development costs. Investments in security may delay time to market and hence have additional opportunity cost in the form of lost first-mover advantages.". In summary, modern software systems are exceedingly complex and, in order to keep up with demand, software engineers settle for any solution, therefore it is hardly surprising that security requirements are treated as negotiable items.

The second issue has to do with the relative immaturity of software engineering as a discipline. This can be seen as the discipline moves from one extreme to another (indeed, Jayaratna, 1994, suggests that there are over 1000 methodologies) in an effort to find the best way to build a software system. The failure of software systems to perform to stated requirements has been well-documented since the "software crisis" of the 1960s. This led to an over-reaction in terms of rigour that spawned sequential process models and concomitant development methods such as SSADM (Downs et al., 1988) that were prescriptive and over-reliant on copious documentation. Even then, there was no guarantee that the system delivered would be that required by the client. Wastell (1996) makes a good argument as to why and how methodology can be used as a "social defence" instead of using it to solve a customer's problem and build a system. In response, agile methods such as Extreme Programming or "XP" (Beck, 1999) suggest building whole system functions from end to end, using the YAGNI (you aren't going to need it) precept. This idea has merit in that the system works (at least partially) in a short space of time, but the precept ignores anything that is not required to fulfil the current set of delivered functions. This means that a system may have to be re-architected at a later stage at significant cost (which negates the whole point of using a method such as XP).

Delivering functional requirements is a dominant theme in the methods described in the preceding paragraph-this is the problem facing security (and, to some extent, other non-functional) requirements. What is particularly interesting is that most experienced software engineers would probably acknowledge that it is the non-functional requirements that cause the most problems unless those requirements are also clearly articulated at the beginning of a project. This paper will illustrate that key point with some examples and then suggest potential solutions.

CASE ONE-THE MARS LANDER

This well-known case is not specifically about security requirements, but it is included here to show how simple (incorrect) assumptions about requirements can have a deleterious effect on the performance of a system.

The official cause of the failure of the Mars Climate Orbiter was "the failure to use metric units in the coding of a ground software file, "Small Forces," used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces). A file called Angular Momentum Desaturation (AMD) contained the output data from the SM_FORCES software. The data in the AMD file was required to be in metric units per existing software interface documentation, and the trajectory modelers [sic] assumed the data was provided in metric units per the requirements." (NASA, 1999, p.6).

Effectively, two teams were working with different units (metric and imperial) and both assumed that the other team was either using the same units or was aware of the difference. The NASA document states that metric units were a (non-

functional) requirement of the system. In contrast, Oberg (1999) claims that "NASA did not originally specify the actual units used, … If the values provided by the spacecraft engineers at Lockheed Martin Astronautics Co., Denver, Colo., had been in pound force [instead of Newtons], they would have been too large by a factor of 4.45.". Oberg goes on to say that while the difference was noted in the behaviour of the spacecraft, the concerns were not acted upon because the escalation process was not followed, ultimately resulting in a USD$125M loss.

The lesson to be learned here (apart from the obvious project management one about team communication) is that non-functional requirements matter. Whilst it is important that the software in this case be able to control the appropriate hardware to fire the appropriate thruster (a functional requirement), it is clearly paramount that the software uses the correct physical units (a non-functional requirement).

CASE TWO-AUTHENTICATION FOR AN ATM SYSTEM

A fairly common problem taught to undergraduate software engineers is that of modelling an automatic teller machine (ATM). Usually, students are given some simple requirements such as:

1. The ATM must be able to dispense banknotes if a customer makes a withdrawal request, has sufficient funds in his/her account, has been authenticated, and the ATM has the requisite banknotes available;
2. A customer must be authenticated to use the functions of the ATM;
3. Before a withdrawal can be processed, a customer's account funds are checked to confirm that there are sufficient funds available to complete the transaction;
4. If the amount requested to be withdrawn is a multiple of the banknotes available in the ATM, the ATM checks that the requisite banknotes are available prior to dispensing the banknotes, otherwise the ATM must advise the customer that the current request cannot be completed.

This leads to models such as figure 3, where there is some separation of functions. The problem lies in the hidden ambiguity of requirement#2. Software engineers are taught that requirements are about what the system is meant to do, not about how to do it. The trouble with this requirement is that the choice of authentication mechanism can drastically alter the cost and effectiveness of the system.

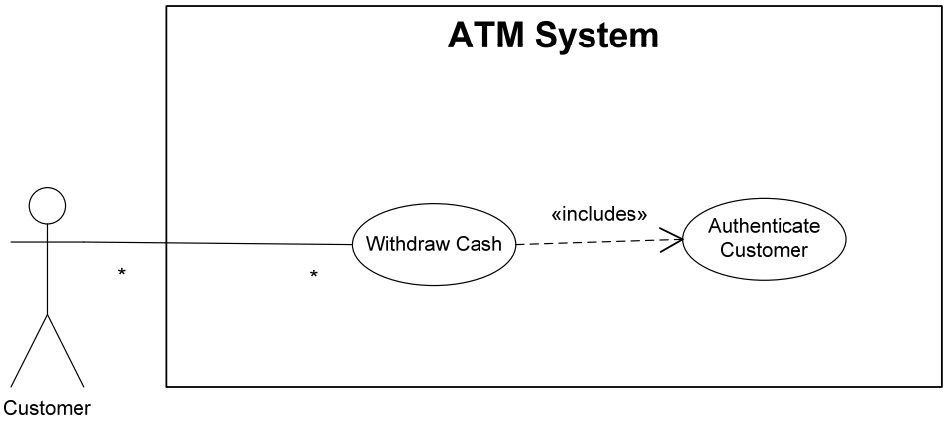

Figure 3 - A Simple Use Case Model of an ATM System

Figure 4 provides several ways that requirement#2 could be implemented (not that all of them would be-the variations are for illustrative purposes). Notice that all of them will work (thus fulfilling the requirement), but they are quite different methods of authentication. Standard information security practice says that authentication should be based on something you know (e.g. a PIN), something you carry (e.g. a card), or something you are (e.g. biometrics). The solutions of figure 4 meet those principles of information security, but the cost of implementing each of them varies considerably. Kotonya and Sommerville (1998) recognise requirement#2 as an abstract non-functional requirement because it does not specify functionality which must be provided for the system to work. Figure 4 provides details of that abstract requirement. Kotonya and Sommerville note that this is a common situation where an abstract non-functional requirement is decomposed into more detailed functional requirements. The idea of abstraction or layering of functions in systems is not new and was first expressed by Parnas in the 1970s. What is novel is that the requirements change class (non-functional vs. functional), depending on the level of abstraction. This provides the basis of a logical argument for inferring that security (non-functional) requirements are in fact functional requirements at some level and therefore should be given the same priority as other system functions. In contrast, a statement such as "the system will use the

RSA Labs public key cryptosystem with 2048 bit keys" is clearly a non-functional requirement as it constrains the system to use a specific algorithm from a specific vendor.

Interestingly, the reverse argument also holds true. Consider requirement#2 to be an abstract functional requirement (because it describes a system function irrespective of any target implementation) and the realisations of figure 4 to be non-functional requirements (because they specify particular technologies), then this situation is a functional requirement being decomposed into more detailed non-functional requirements, also a common situation (but the reverse of that described by Kotonya and Sommerville, 1998).

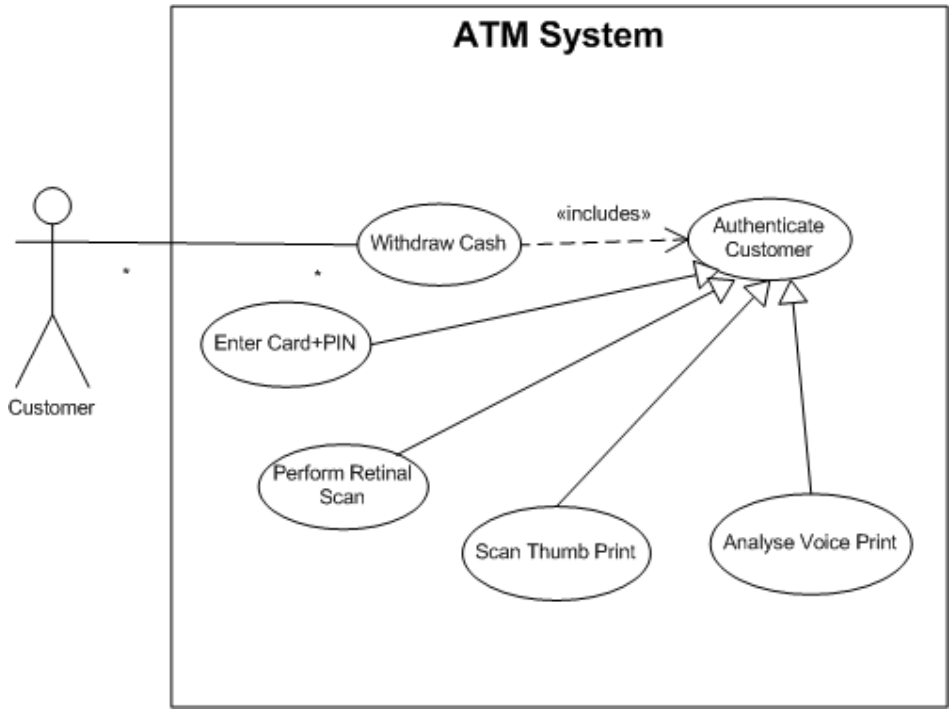

Figure 4- A More Refined Use Case Model of an ATM System.

POTENTIAL SOLUTIONS

Several potential solutions are immediately evident. One approach is to promote awareness and provide better education for either software engineers or computer security professionals regarding security requirements. Another is to encourage the use of security-oriented development methods. A final approach is to inculcate secure coding practices into undergraduates. Each of these approaches will be considered for their merits and pitfalls.

An obvious first solution is to educate software engineers (both students and professionals) as to the need to treat security requirements with the same level of gravitas as (other) functional requirements. This will have little effect on either party at present. Students are unlikely to have the maturity to see the necessity for security requirements. This is not surprising as students are still learning how to elicit, specify and validate requirements, so increasing their cognitive load will probably not have the desired effect unless something else in the curriculum is removed.

Software engineering professionals will defer taking action unless the security requirements are visible to the customer. Software engineers spend time on the parts of a system that are exposed to the customer (as pointed out by Shostack and Stewart, 2008). If there is a small risk that a vulnerability might be exploited, then that might be considered a tolerable risk, especially as such a risk is deferred until after delivery in that it is unlikely to be part of the formal acceptance criteria for the system. As in safety-critical systems, the key drivers for software professionals to take security requirements seriously will be a) contractual enforcement of adherence to standards (e.g. ISO/IEC 17799); or b) the risk of litigation if it can be proven that a known, pre-existing vulnerability was the direct cause of, or contributed to, significant economic loss or loss of life. In the case of the former, this is already common practice for certain classes of software systems (cf. Def Stan 00-55 in the UK), so introducing a security-related standard should not raise any significant issues. In the case of the latter, this is a reasonable position to take as it would be a sign that the software

engineering discipline is maturing and also given that professionals in other engineering disciplines are already treated to the same level of scrutiny in their work.

Davis and Dark (2003) suggest that the information assurance community (and presumably by extension the computer and information security communities) can learn from the method used by software engineering educators in terms of taking a holistic approach and moving from broad principles to focussed technical subjects. This is evident in the framing of the Software Engineering Body of Knowledge (SWEBOK), and in fact, Davis and Dark suggest constructing a common body of knowledge for information assurance. The result would be, according to Davis and Dark, that "Students repeatedly internalize knowledge and skills leading to, for example, reusable and safe software. Over time, students adopt best practice models as second nature.".

The drive to provide more secure systems has led to the development of a range of secure development methods, perhaps the best-known being SQUARE (Mead and Stehney 2005), CLASP (OWASP, 2006) and SDL (Howard and Lipner, 2006). These methods put security requirements at the forefront of all stages of the development lifecycle (which is, after all, what they were designed to do), but suffer from the disadvantage that, being relatively new methods, they have not yet been extensively field-tested. A further issue with methods in a new area is they are also subject to being evaluated and compared (perhaps overly so) to one another using various criteria (see, for example, De Win et al., 2009). This has echoes of the CRIS series of workshops in the 1980s and 1990s that sought to compare more conventional systems development methods. Thus computer security professionals will find that the software engineering community understands these issues.

The final approach, that of educating undergraduates to use secure coding practices is a long-term goal. Some universities are already doing this by delivering subjects that teach the theory and practice of secure coding. Such subjects usually have titles similar to "Programming Secure Software Systems", and thus are easily identifiable in the curriculum. The clear benefit here is that, over time, secure coding practices will become the norm rather than the exception, leading to a significant reduction in vulnerabilities. For this approach to be effective, students must understand exploits very well in order to see where vulnerabilities may lie. This approach is problematic in that for some this is tantamount to educating the next generation of hackers. Frincke (2003) and Rubin and Cheung (2006) provide some interesting discussion as to the merits of this approach.

A combination of these approaches will probably be the way forward, tempered by the (fairly blunt) economic arguments put forward by Anderson (2001).

CONCLUSION

This study used evidence of malware exploit vulnerabilities from several sources as a basis for recognising the scope of the vulnerability problem in existing software systems. Further, the complex nature of modern software systems was unveiled and reasons why software engineers do not prioritise security requirements appropriately put forward.

Specifically, this study used two case studies to show how misinterpretations of non-functional requirements can have significant consequences for the end product. It was argued that security requirements are, in fact, functional requirements at some level and should be given the same priority as other system functions. One case provided a simple example of the decomposition of a supposedly non-functional security requirement into several functional requirements, thus lending credence to the argument. More significantly, the reverse argument also holds true, that abstract functional requirements can be decomposed into concrete non-functional requirements. Potential solutions to reduce vulnerabilities were discussed, viz., education, security-oriented development methods and secure coding practices, with the suggestion that it is likely that a combination of approaches will eventuate.

Whilst the use of a simple case study facilitated the realisation that security requirements can be treated as functional requirements, it is acknowledged that a limitation of this work is that it is unwise to generalise from a single case study therefore further work could be to field-test this idea to see how security requirements are surfaced and detailed in real-world systems.

REFERENCES

Anderson, R. (2001). Why Information Security is Hard: An Economic Perspective. Cambridge University Technical Report.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change.* Upper Saddle River, NJ: Addison Wesley.

Brooks, F.P. (1987). "No Silver Bullet: Essence and Accidents of Software Engineering". *Computer*, (April), 10-19.

CERT. (2009). CERT Statistics (Historical). Retrieved August 20, 2009, from http://www.cert.org/stats/

Charette, R.N. (2009) This Car Runs on Code, IEEE Spectrum, February 2009 Retrieved September 11, 2009, from http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code

Cusumano, M.A. (2006) 'What road ahead for Microsoft and Windows?' *Communications of the ACM*, 49(7), pp. 21-23.

Davis, J. and Dark. M. (2003). "Teaching Students to Design Secure Systems". *IEEE Security & Privacy*, March/April, pp. 56-58.

De Win, B., Scandariato, R., Buyens, K., Grégoire,J. and Joosen, W. (2009) On the secure software development process: CLASP, SDL and Touchpoints compared. *Information and Software Technology* 51, pp. 1152–1171.

Downs, E., Clare, P. and Coe, I. (1988). *Structured Systems Analysis and Design Method: Application and Context.* Hemel Hempstead, Herts.: Prentice-Hall (UK).

Frinke, D. (2003). "Who Watches the Security Educators?". *IEEE Security & Privacy*, May/June, pp. 56-58.

Howard, M. and Lipner, S. (2006) *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Redmond, WA: Microsoft Press.

Jayaratna, N. (1994). *Understanding and Evaluating Methodologies.* London: McGraw-Hill.

Korzybski, A. (1936). *The Extensional Method*. in *ALFRED KORZYBSKI: Collected Writings 1920-1950*, pp. 239-244, International Non-Aristotelian Library, Institute of General Semantics. Retrieved June 20, 2004, from http://www.korzybski.org/extensional.html

Kotonya, G. and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques.* Chichester, West Sussex: John Wiley & Sons.

Lyytinen, K., Rose, G. and Welke, R. (1998) The Brave New World of Development in the InterNetwork Computing Architecture (InterNCA): Or How Distributed Computing Platforms Will Change Systems Development. *Information Systems Journal*, 8(3), pp. 241-253.

Mead, N.R. and Stehney, T. (2005). Security Quality Requirements Engineering (SQUARE) Methodology. *Proc. Software Engineering for Secure Systems: Building Trustworthy Applications (SESS'05)*, May 15-16, 2005, St Louis, MO,USA.

NASA (1999) Mars Climate Orbiter Mishap Investigation Board: Phase I Report. Washington DC: NASA.

NIST. (2009) National Vulnerability Database (NVD) CVE Statistics. Retrieved December 16, 2009, from http://web.nvd.nist.gov/view/vuln/statistics.seam?cid=2.

Oberg, J. (1999). "Why the Mars probe went off course ". *IEEE Spectrum*, 36(12), December, pp. 34-39.

OECD. (2008) Malicious Software (Malware): A Security Threat to the Internet Economy. Ministerial Background Report DSTI/ICCP/REG(2007)5/FINAL. Seoul, Korea: OECD.

OWASP. (2006) OWASP CLASP (Comprehensive, lightweight application security process) Project. Retrieved August 20, 2009, from http://www.owasp.org.

RFC 2828 (2000) *Internet Security Glossary*. Internet Engineering Task Force. Retrieved September 22, 2009, from http://www.ietf.org/rfc/rfc2828.txt

Rubin, B.S. and Cheung. D. (2006). "Computer Security Education and Research: Handle with Care". *IEEE Security & Privacy*, November/December, pp. 56-59.

Shostack, A. and Stewart, A. (2008) *The New School of Information Security*. Upper Saddle River, NJ: Addison Wesley.

Sommerville, I. (2001) *Software Engineering*. 6th ed. Essex, England: Pearson Education.

Standish Group (2003). Latest Standish Group CHAOS report shows project success rates have improved by 50%. Retrieved June 09, 2003, from http://www.standishgroup.com/press/article.php?id=2

Symantec. (2008). Internet security threat report. Volume XIV. Analysis of threat activity January-December 2008. Retrieved June 22, 2009, from http://www.symantec.com/business/theme.jsp?themeid=threatreport

Viega, J. (2005). Building Security Requirements with CLASP. *Proc. Software Engineering for Secure Systems: Building Trustworthy Applications (SESS'05)*, May 15-16, 2005, St Louis, MO,USA.

Wastell, D.G. (1996). 'The Fetish of Technique: Methodology as a Social Defence.' *Information Systems Journal* , 6(1), pp. 25-40.

Wysopal, C., Nelson, L., Dai Zovi, D. and Dustin, E. (2007). *The Art of Software Security Testing*. Upper Saddle River, NJ: Addison Wesley.

COPYRIGHT