Static Analysis is Not Enough:

The Role of Architecture and Design in Software Assurance

Walter Houser, NIST

"Software design errors are like space explosions. They are seldom heard and hard to spot. You do not see them coming but when they hit, they hit with more energy than they did when initiated. Once found, you spend the rest of the project dodging their debris. Spend your time and attention in the design phase. Knock out design flaws while they are small; do not wait until they are death stars. [1]"

- Gary Petersen, Shim Enterprise, Inc., CrossTalk, July 2004[1]

Abstract. Static analysis testing of software source code is necessary but not sufficient. Of the nearly 1000 CWEs, 40 percent can be introduced in the architecture and design phase of the development life cycle.[2] By their very nature, architectural and design flaws are difficult to find via static analysis. Furthermore, fixes to architectural and design errors can be complex, can inject additional defects, and can alert adversaries to the existence of these weaknesses. Moreover design flaws can obscure coding bugs that static analysis might otherwise detect, as demonstrated by the Heartbleed vulnerability.[3] This paper describes the techniques that architects and designers can employ to minimize the implementation of architectural and design flaws.

Introduction

Identification and mitigation of flaws early in the software development life cycle (SDLC) may avoid a ten to hundred-fold cost in post deployment detection and remediation.[4, 5] Yet cybersecurity reviews are often done just prior to an application's going live, typically because of a requirement for compliance.[6, 7] Architectural and design flaws found late in the SDLC can be costly to repair so they are often catalogued and not acted on until the next release (assuming the application survives that long). Moreover, by their very nature, architecture and design flaws are resistant to code patches. Fixes to these errors can further compound the problem by injecting additional defects. Moreover, patches can alert adversaries to the existence of these flaws.[8] Given that there are more than 61000 documented common vulnerabilities and exposures (CVE),[9] web application firewall rules can provide only partial mitigation against the exploitation of applications. This paper describes the techniques architects and designers can employ to minimize flaws in applications.

Definitions

The following definitions clarify concepts in this article:

Bug: A mistake introduced during the development, implementation, and sustainment phases of the SDLC.[12]

Flaw: A mistake introduced during the conceptual, architectural, or design phases of the SDLC.[13] Flawed code can be bug-free. "Microsoft reports that more than 50% of the problems the company has uncovered during its ongoing security push are architectural in nature. Cigital data show a 60/40 split in favor of flaws…"[14]

Software architecture: "…the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them… In this regard, architecture is the primary determiner of modularity and thus the nature and degree to which multiple design decisions can be decoupled from each other. Thus, when there are areas of likely or potential change, whether it be in system functionality, performance, infrastructure, or other areas, architecture decisions can be made to encapsulate them and so increase the extent to which the overall engineering activity is insulated from the uncertainties associated with these localized changes."[15]

Software development life cycle (SDLC): The scope of activities associated with a system, encompassing the system's initiation, development and acquisition, implementation, operation and maintenance, and ultimately its disposal that instigates another system initiation.[16]

Vulnerability: An occurrence of a weakness (or multiple weaknesses) within software, in which the weakness can be used by a party to cause the software to modify or access unintended data, interrupt proper execution, or perform incorrect actions that were not specifically granted to the party who uses the weakness.[17]

Weakness: A type of mistake in software that, in proper conditions, could contribute to the introduction of vulnerabilities within that software. This term applies to mistakes regardless of whether they occur in implementation, design, or other phases of the SDLC.[18] For the purposes of this paper, weaknesses are categorized as being either flaws or bugs.

Common Weaknesses in Architecture and Design

In contrast to the dictionary of specific software vulnerabilities itemized in the CVE, the more general Common Weakness Enumeration (CWE)[10] is a dictionary of classes of flaws and bugs. Managed by MITRE Corporation under the sponsorship of the Department of Homeland Security (DHS), the CWE provides a publicly available, unified, measurable set of software weaknesses. As part of building the CVE, MITRE developed in 2005 a preliminary classification and categorization of vulnerabilities, attacks, faults, and other concepts to generalize the CVE into common software weaknesses. However, while sufficient for CVE, those groupings were too rough to be used to identify and categorize the functionality offered by the code security assessment tool makers. The CWE List was created to better address those additional needs.[11] CVE are akin to an inventory of traffic accidents while the CWE are analogous to the conditions (e.g. highway architecture, bridge and road design, traffic signage, driver practices, enforcement procedures) that lead to accidents. Given the large number of CVE and the process for reporting them, CVE references to architectural and design causes are intermittent and largely unverified. When generating findings

from code scans, static code analysis tools can draw upon the CWE for weakness descriptions and mitigation recommendations; identifying the relevant CVE would be difficult given their specificity and their disconnection from the code that generated them.

Thus the CWE enables more effective discussion, description, selection, and use of software security tools and services for finding weaknesses in source code and operational systems. Just as importantly, the CWE promotes a better understanding and management of software weaknesses related to architecture and design.

Although the CWE is most commonly associated with static code analysis reporting, MITRE classifies 381 of the total of 943 CWEs "as likely introduced in the Architecture and Design phase of the development life cycle."[19] Most of these CWEs are not apparent in the coding phase but can be identified during the design phase via the techniques discussed below. In other words, over 40 percent of the CWE can be addressed early in the SDLC in the architecture and design phase. But if not detected then, these weaknesses can slip past subsequent automated and manual reviews, with substantial cost and delays.

Three Perspectives of Architecture

To better understand these flaws and how to address them, we posit three perspectives for architecture: enterprise architecture, security architecture, and software architecture. The enterprise architecture perspective looks to the welfare and the effectiveness of the entire enterprise. The enterprise architect is concerned with the designing, planning and governing of strategic missions and programs. The enterprise architect seeks to optimize and harmonize services, processes, or components across the enterprise to achieve interoperability and portability. To enable the business objectives of the enterprise, the enterprise architect works with the respective software architects to leverage solutions across multiple systems and product lines. For example, the enterprise architect will consolidate individual systems into a service-oriented architecture and eliminate one-off personnel data sets, asset inventories, and expense tracking and reimbursement. The enterprise architect will promote adoption of a single technical architecture (such as Microsoft's .NET or Java Platform, Enterprise Edition) to avoid the headaches of interconnecting incompatible technologies. Portability and interoperability are the key architectural objectives.

Examples of flaws that can stem from a misunderstanding, or a misapplication of, enterprise architecture include:

- Erroneous business rules can lead, for example, to CWE-20: Improper Input Validation,
- Misunderstood user authorities and responsibilities can lead, for example, to CWE-272: Least Privilege Violation and CWE-200: Information Exposure,
- Errors in integration and module development can be linked to a deficient or missing SDLC strategy, [2] and
- Incompatible data definitions and inconsistent data management can lead to CWE-202: Exposure of Sensitive Data Through Data Queries.

The perspective of security architecture concerns itself with marshalling the controls, tools, and skills needed to protect the enterprise from external and internal threats. The security architect applies knowledge of the business goals and roles to identify and mitigate threat to the enterprise.

"… the first iteration of the analysis should take place when only the operational concept and a notional architecture is defined. Though the fidelity of the analysis may be fairly rough, this early stage is the perfect time to be considering what attacks your system could be facing, and whether there are design, architecture, physical composition choices or changes in operational concepts that could dramatically help to mitigate, manage, or control those attacks with minimal cost and schedule impact."[20]

The recent iOS Security White Paper[21] from Apple is a good example of security architecture. The White Paper covers the iPhone system hardware and software security, encryption and data protection, application security, network security, internet services, and device controls. Apple identifies the threats at each of these levels and describes how the architecture addresses them. "iOS protects not only the device and its data at rest, but the entire ecosystem, including everything users do locally, on networks, and with key Internet services."

Some flaws that should be addressed by security architecture include:

- CWE-260: Password in Configuration File
- CWE-261: Weak Cryptography for Passwords
- CWE-310 Cryptographic Issues
- CWE-330 Use of Insufficiently Random Values
- See CWE-254: Security Features for other CWE in this category.

The Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK®) asserts that software architectural design "(sometimes called high-level design) develops top-level structure and organization of the software and identifies the various components."[22] It is a top down process with various subcomponents and relationships between components. A software architect works within a business unit to identify stakeholders and their concerns, developing the architectural requirements for the systems that the business unit develops and deploys. The software architect coordinates enterprise coding standards and recommended practices with software development managers and software designers to reduce training costs and learning curves and promote portability of developers and applications across the enterprise.

The SWEBOK[23] distinguishes architectural design from detailed design which "describes the desired behavior of these components." However, this distinction will vary by organization. Some enterprises will not have software architects, instead dividing those duties between enterprise architects and system and software designers. In some cases the enterprise architecture will incorporate the higher level constructs of the security architecture, relegating the details to the developers and security engineers. The software architects could develop the security architecture, but this is not common given the special-

ized knowledge and skills of the security architect.

Some software architectural design flaws include:

- CWE-203: Information Exposure Through Discrepancy
- CWE-710: Coding Standards Violation
- CWE-289: Authentication Bypass by Alternate Name
- CWE-349: Acceptance of Extraneous Untrusted Data With Trusted Data
- CWE-280: Improper Handling of Insufficient Permissions or Privileges
- CWE-227: Improper Fulfillment of API Contract ('API Abuse')
- CWE-733: Compiler Optimization Removal or Modification of Security-critical Code
- CWE-14: Compiler Removal of Code to Clear Buffers

Maturity Model Context for Software Architecture and Design Controls

What can organizations do to mitigate or remediate architectural and design errors? Why not just fix the flaws identified in the CWE/SANS Top 25 Most Dangerous Software Errors? Before itemizing the anti-flaw arsenal, an SDLC-based framework can provide context for these controls and identify the gaps that will result from a purely tool and techniques point of view. MITRE[24] recommends that one should "treat the Top 25 as an early step in a larger effort towards achieving software security. Longer term strategic possibilities are covered in efforts such as Building Security In Maturity Model (BSIMM),[25] SAFECode,[26] OSAMM,[27] Microsoft SDL,[28] and OWASP ASVS."[29]

As an assessment of maturity models is beyond the scope of this article,[30] we will focus on BSIMM owing to its broad appeal to security practitioners seeking pragmatic and actionable content. BSIMM version V reports on the software security initiatives of sixty-seven firms drawn from twelve verticals (with some overlap): financial services (26), independent software vendors (25), cloud (16), technology firms (14), telecommunications (5), retail (4), security (4), healthcare (3), media (3), insurance (2), energy (1), and internet service provider (1).[31] Two of the four domains of the BSIMM Version 5[32] – Governance and Intelligence – list practices that are exclusively related to architecture and designs. A third domain – Secure Software Development Lifecycle (SSDL) Touchpoints – has the first of its three practices as Architecture Analysis. In all, seven of the 12 practices or categories of BSIMM activities precede the development phase of the SDLC.

BSIMM's governance domain encompasses planning, assigning roles and responsibilities, identifying software security goals, determining budgets, and identifying metrics and gates. This domain also identifies controls for compliance, setting organizational software security policy, and auditing. Also part of governance domain, training is critical because software developers and architects often start with little security knowledge. BSIMM's intelligence domain addresses organization-wide resources such as attack models to capture information used to think like an attacker (threat modeling, abuse case development and refinement, data classification, and technology-specific attack patterns). This domain also covers the creation of security patterns for major security controls, building middleware frameworks for those controls, and creating and publishing other proactive security guidance. Lastly the intelligence domain involves eliciting explicit security requirements from the organization, determining which COTS to recommend, building standards for major security controls, creating security standards for technologies in use, and creating a standards review board. BSIMM's SSDL touchpoints domain includes architecture analysis: 1) capturing the software architecture in concise diagrams, 2) applying lists of risks and threats, 3) adopting a review process, and 4) constructing an assessment and remediation plan for the organization.[33]

Security Controls for Software Architecture and Design

Given the BSIMM (or one of the other maturity models) to provide context, enterprises can integrate the following methods into their system development lifecycle process.

Threat Modeling

Threat modeling (aka threat analysis) is "a design-time conceptual exercise where a system's dataflow is analyzed to find security vulnerabilities and identify ways they may be exploited."[34] The process of identifying threats can reveal flaws in data handling, data sensitivity, authorization requirements, workflow, business logic, and hazards and other manifestations of failure[35] that can be addressed with changes in documentation rather than revisions to deployed code or re-engineered solutions. Moreover, static analysis tools lack knowledge of the operating environment and can only infer the potential threats facing the system being analyzed."[36]

Categorize data by information types[37] to establish data sensitivities, data sharing rules, and access privileges. Next, users can be grouped by privilege and their duties can be separated to prevent abuse and theft. The manner in which duties are separated may be an architectural determination derived from organizational missions; the software architect or software designer will need to ensure the appropriate stakeholders are identified and consulted before coding begins.

Data flow analysis can identify the appropriate levels of encryption and appropriate data handling, enforcement of encrypted data transfers between web browsers and servers, and encrypted backend data storage and transfer. Threat analysis can identify restrictions on the type and quantity of content a user can upload and download, as well as the file headers needed to enforce those actions.

Attack Trees

Attack trees, or threat trees,[38] provide a formal model for depicting the attacker perspective on the security of systems. The potential attacks against a system are arranged in a tree structure, with the ultimate objective as the root node. The leaves represent the various paths to achieving that objective. In Figure 1, the path to "obtaining the authentication credentials" uses network monitoring to recognize credential data. The arc labeled 'AND' indicates that the connected two leaf nodes both must

be completed before going to the parent node. Every arc that isn't labeled as an 'AND' is considered an 'OR' condition. Adding monetary values to the leaf nodes can indicate the financial cost for an attacker to accomplish the attack. With assigned monetary values, the designer can create countermeasures that increase the expense of attack routes.[39]

Misuse Cases

Misuse cases[40] are like use cases that present potential abuses of the system. Like use cases, misuse cases require understanding of the functionality to be provided by the application. A use case generally describes behavior that the system owner wants the system to implement. In contrast, misuse cases create conditions or situations that are undesirable in the view of the stakeholders. Misuse cases help organizations understand their software as the attackers would. Just as use-case models have proven quite helpful for eliciting functional requirements, misuse cases can effectively reveal security requirements. Use misuse cases to turn non-functional requirements into functional requirements.

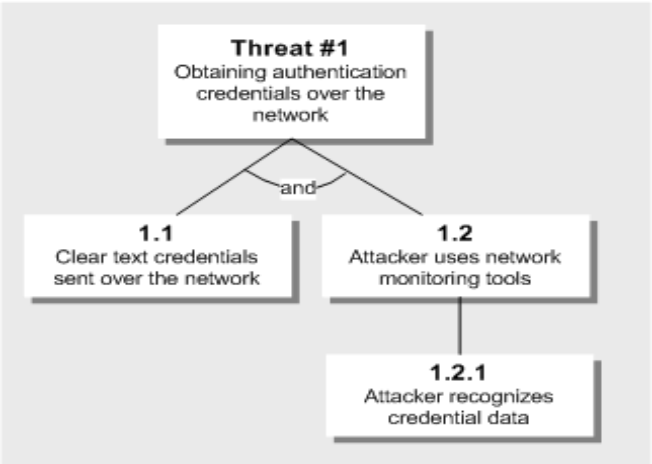

Figure 1: Attack Tree Example

Identity and Access Management

Owing to the complexity of the task, most applications use enterprise services for identity and access management (IDAM) of internal users[41]. However, an increasing number of applications are using outside social networks to authenticate external users. Although on-boarding for social networks tends to be far less rigorous than enterprise on-boarding processes, membership in a social network raises the threshold for malicious and spam accounts. However outsourcing IDAM for external users in this manner trades away user privacy to avoid the technical and management challenges in creating and maintaining accounts for external users. Furthermore such outsourced services are hard (or impossible) to scan for coding errors and review for design flaws, particularly if provided for free or in barter for customer usage data.

The software or security architect, or possibly the application designer, will need to identify and negotiate roles and responsibilities for IDAM controls. The architects and designers will need to agree on password complexity, resulting action for failed login attempts, session timeouts, user session refresh, forced re-authentication, re-authentication for privileged or sensitive data access, etc. The enterprise needs consistency for password reset and login input sanitization to prevent injection attacks. The software or security architect should ensure the IDAM interface enforces strict authentication and least privilege for administrative user access and prevention of session hijacking attacks against privileged users. The IDAM must meet authorization and accountability requirements, as well as provide logging and reviewing of administrative transactions for insider threats.

Least Privilege

The concept of least privilege is: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job."[42] System designers can reduce the damage caused if a system is compromised by considering least privilege during threat modeling. A compromised application running with full privileges can perform more damage than a compromised application executing with reduced privileges. Most operating systems make little if any distinction in access privilege between a web browser and a word processer, despite the greater risks associated with the former's exposure to the internet. Sandbox applications to employ operating system security features to restrict the access available to sandboxed processes.

Formal methods

Formal methods are the incorporation of mathematically based techniques for the specification, development, and verification of software.[43] Owing to mathematical syntax and semantics, formal specification is precise when compared to non-formal and even semi-formal specifications that may be ambiguous or internally inconsistent. "Much anecdotal evidence suggests that formal verification can increase productivity, improve quality, and reduce development time by finding errors early and avoiding rework and testing delays."[44] Formality is used to achieve clarity of expression, to force early expression of precise behavior, and to allow more powerful verification and validation techniques to be applied."[45]

Formal methods add to the time needed to specify an application, yet they can improve the code quality and reduce testing and maintenance costs.[46] Formal methods should be employed in safety critical systems such as trains and nuclear reactors. Formal methods can be used in the design phase to build and refine the software's formal design specification, as well as employed in verification to prove that each step satisfies the requirements imposed by previous steps. Likewise, they can be used to improve software security, although they are not well understood and not commonly practiced in that context owing to the training and discipline required.

To compensate for a lack of experienced personnel and organizational support, formal methods can focus on algorithms,

components, properties, and other key functions in software, rather than applying them to the entire system. As for training limitations, it may be difficult to find developers with the needed expertise in formal logic, the range of appropriate formal methods for an application, or appropriate automated software development tools for implementing formal methods. Therefore, developers should employ formal methods for software likely to be reused or for critical functionality.

Secure Design Patterns

Secure design patterns[47] are descriptions or templates describing general solutions that can be applied in many different situations to eliminate or mitigate the consequences of flaws. By providing a higher level of abstraction than secure coding guidelines, secure design patterns can be applied across programming languages. Secure design patterns differ from security patterns in that the former do not describe specific security mechanisms such as access control, authentication, authorization, and logging. Whereas security patterns are focused on security-related functionality, secure design patterns can (and should) be employed broadly in a system.

Secure Session Management

The hypertext transport protocol is stateless[48]; the protocol does not maintain user state from one page to the next. Session management allows web applications to authenticate users at the beginning of the session by issuing a Session ID. This ID ensures that all actions during the session are performed by the same user (or web browser) that originally supplied their authentication information. Attackers will seek to manipulate the Session ID to steal the session from an authenticated user. Developers can defend against such attacks by requiring re-authentication when the session has timed out or when the user attempts to use sensitive functionality. The system designers must identify all functions where preservation of session state is necessary.

Conclusions

Over 40 percent of the CWEs can and need to be addressed in the architecture and design phases of the SDLC, especially as they are not usually apparent in the coding phase. They can be prevented, discovered, and mitigated via attack trees, threat models, misuse cases, and secure design patterns. Designers can use these techniques to identify sensitive information for encryption at rest and in transit. Session identifiers must be protected. Formal methods can improve software security with mathematically proven techniques. Lastly, managers should assure themselves that development teams are taking appropriate measures to prevent software weaknesses.[49] Figure 2 suggests how these solutions inter-relate and a possible sequence in which they may be applied.

Acknowledgements

Ram Sriram, Paul Black, Yan Wu, Vadim Okun, Bertrand Stivalet, and Clarence "Butch" Rappe provided useful feedback on earlier versions of this paper.

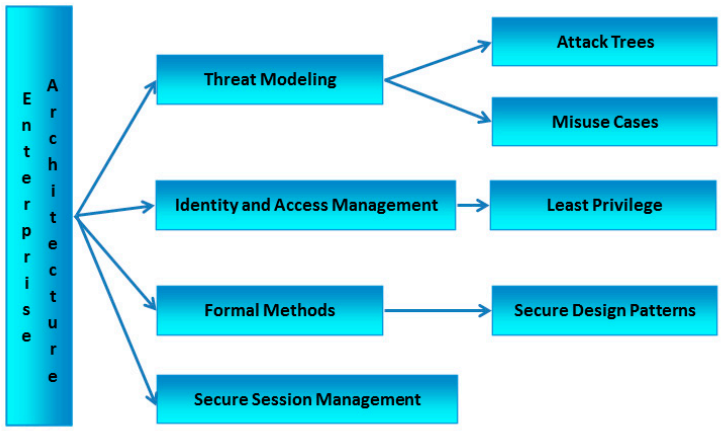

Figure 2: Security Controls for Software Architecture and Design

ABOUT THE AUTHOR

Walter Houser works in the Software and Systems Division of the National Institute of Standards and Technology (NIST) on the Software Assurance Metrics and Tool Evaluation (SAMATE) Project. There he investigates design flaws, code weaknesses, and reviews static analysis tool findings. Previously, he led information assurance projects with numerous Federal agencies and served as an IT policy officer, webmaster, applications development manager, and enterprise architect. Over 40 years ago he began his career as a COBOL and FORTRAN programmer; his work with SAMATE returns him to his professional roots to bring a coder's perspective to the challenge

Phone: 703-284-6180
E-mail: walter.houser@nist.gov

REFERENCES

1. Certain commercial entities, equipment, or materials may be identified in this article to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.
2. "The SSG [software security group] takes a proactive role in software design by building or providing pointers to secure-by-design middleware frameworks or common libraries. In addition to teaching by example, this middleware aids architecture analysis and code review because the building blocks make it easier to spot errors. " Also "Secure coding standards help developers avoid the most obvious bugs and provide ground rules for code review. Secure coding standards are necessarily specific to a programming language and can address the use of popular frameworks and libraries." Although the identification and maintenance of frameworks, libraries and coding standards is arguably a security architecture function, the decision to create an SSG and to establish and enforce these methods is an enterprise commitment. "We have observed that successful software security initiatives are typically run by a senior executive who reports to the highest levels in an organization. These executives lead an internal group that we call the Software Security Group (SSG), Software Security Group (SSG)…" McGraw, Gary. Sammy Migues, Jacob West. Building Security In Maturity Model, Version 5, October 2013, at <http://bsimm.com/>
`

NOTES

1. Shim, Gary, "Movie Physics and the Software Industry," CrossTalk The Journal of Defense Software Engineering, July 2004 <http://www.crosstalkonline.org/storage/issue-archives/2004/200407/200407-Petersen.pdf>
2. MITRE. CWE-701: Weaknesses Introduced During Design <https://cwe.mitre.org/data/lists/701.html>
3. A. Chou, On Detecting Heartbleed with Static Analysis, April 2014, at <http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html>
4. Kan, Stephen. Metrics and Models in Software Quality Engineering, 1995, P.171. "Defect removal at the earlier development phases is generally less expensive… Fagan (1976) contended that rework done at the I0, I1, and I2 inspection levels can be 10 to 100 times less expensive than if done in the last half of the process (formal testing after code integration)."
5. Jones, Capers and Olivier Bonsignour, The Economics of Software Quality, 2012, P.97. "The oft repeated aphorism that 'it costs 100 times as much to fix a bug after release as during design' is based on a flawed analysis that ignores fixed costs. Due to fixed costs, the following rule applies: 'Cost per defect is always cheapest where the number of defects found is highest.' … the cost-per-defect metric focuses too much attention on defect repairs and ignores the greater economic advantages of high quality in shortening schedules and lowering costs." Furthermore on page 111 … "Cost per defect penalizes quality and is always cheapest where the greatest numbers of bugs are found." And on page 112 "writing test cases and running them act like fixed costs."
6. Federal Information Security Management Act of 2002 Title III of the E-Government Act of 2002 (Pub.L. 107–347, 116 Stat. 2899).
7. Payment Card Industry Data Security Standard <https://www.pcisecuritystandards.org/security_standards/index.php>
8. Rains, Tim. Director of Trustworthy Computing, Microsoft <http://blogs.technet.com/b/security/archive/2013/08/15/the-risk-of-running-windows-xp-after-support-ends.aspx>
9. <https://cve.mitre.org/cve/cve.html>
10. MITRE. CWE–Common Weakness Enumeration. 10 Apr 2014 <http://cwe.mitre.org/>.
11. Ibid. "What is the relationship between CWE and CVE?" <http://cwe.mitre.org/about/faq.html#A.8>
12. MITRE. CWE-701: Weaknesses Introduced During Design <https://cwe.mitre.org/data/lists/701.html>
13. McGraw, Gary. Software Security. 2006. P. 14. Please note that MITRE does not make this distinction in describing the CWE
14. Ibid, p. 16.
15. Ibid, p. 18.
16. Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK®) A Project of the IEEE Computer Society, 2014, Page 7-5.
17. "National Information Assurance (IA) Glossary: CNSS Instruction No. 4009." Committee on National Security Systems, 26 April 2010, page 72. <http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf>.
18. MITRE. <https://cwe.mitre.org/documents/glossary/index.html#Vulnerability> MITRE.
19. MITRE. <https://cwe.mitre.org/documents/glossary/index.html#Weakness>
20. MITRE, Engineering for Attacks <https://cwe.mitre.org/community/swa/attacks.html>
21. <http://images.apple.com/ipad/business/docs/iOS_Security_Feb14.pdf>
22. SWEBOK® Version 3.0 IEEE Computer Society, 2014, P. 2-1
23. SWEBOK® Version 3.0 IEEE Computer Society, 2014, P. 2-1
24. <http://cwe.mitre.org/top25/index.html>
25. McGraw, Migues, and West, ibid.
26. Software Assurance Forum for Excellence in Code (SAFECode). Fundamental Practices for Secure Software Development, page 2. <http://www.safecode.org/publications/SAFECode_Dev_Practices0211.pdf> Software Assurance Forum for Excellence in Code http://www.safecode.org/publications.php
27. The Open Software Assurance Maturity Model <http://www.opensamm.org/>
28. Microsoft Security Development Lifecycle <http://www.microsoft.com/security/sdl/default.aspx>
29. Open Web Application Security Project (OWASP) Application Security Verification Standard Project (ASVS) <https://www.owasp.org/index.php/ASVS>
30. SwA Capability Benchmarking, <https://buildsecurityin.us-cert.gov/swa/forums-and-working-groups/processes-and-practices/swa-capability-benchmarking>
31. The list of 47 firms who have agreed to be identified is listed at <http://bsimm.com/community/>.
32. McGraw, Migues, and West, ibid.
33. McGraw, Migues, and West, ibid, page 19.
34. McGraw, Migues, and West, op.cit.
35. Critical Code: Software Productivity for Defense, 2010, Computer Science and Telecommunications Board, National Academy of Sciences. Page 92.
36. SAFECode, Op. cit.
37. NIST Special Publication 800-60 Rev. 1 Guide for Mapping Types of Information and Information Systems to Security Categories Aug 2008.
38. Swiderski, Frank, and Window Snyder, Threat Modeling, Microsoft Press, 2004.
39. Related to attack trees, "kill chains" can be reviewed by designers when developing attack trees. Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains Eric M. Hutchins, Michael J. Clopperty, Rohan M. Amin, Ph.D. <http://papers.rohanamin.com/wp-content/uploads/papers.rohanamin.com/2011/08/iciw2011.pdf>. For an example of a kill chain analysis, see A "Kill Chain" Analysis of the 2013 Target Data Breach, Majority Staff Report, Committee on Commerce, Science, and Transportation, U.S. Senate. March 26, 2014 <http://www.commerce.senate.gov/public/?a=Files.Serve&File_id=24d3c229-4f2f-405d-b8db-a3a67f183883>
40. OWASP Comprehensive, Lightweight application Security Process (CLASP), <http://www.lulu.com/shop/owasp/owasp-clasp-v12/paperback/product-1869926.html>
41. National Strategy for Trusted Identities in Cyberspace (NSTIC), <www.nist.gov/nstic/>
42. Saltzer, Jerome and Michael Schroeder "The Protection of Information in Computer Systems.", 1975 <http://web.mit.edu/Saltzer/www/publications/protection/>.
43. Ghezzi, Carlo. Mehdi Jazayeri and Dino Mandrioli Fundamentals of Software Engineering (2nd Edition), Sep 29, 2002. 334-337 and pp 550-559. "Validation of the specification and verification planning required slightly more resources than expected,… The design activity turned out to be substantially simpler than in similar traditional projects, because of the higher quality of the specification."
44. P. Black, personal correspondence.
45. Ibid
46. The National Security Agency (NSA) asked Praxis High Integrity Systems to undertake a research project to develop part of an existing secure system (the Tokeneer System) using Praxis' Correctness by Construction development process. "Notice that the productivity during coding for the TIS core is higher than for the support software despite the core coding effort including static analysis. This is because there was very little rework of the TIS core software since the early lifecycle activities produced an unambiguous definition of the required software functionality… It should be noticed that almost half of the project costs were incurred prior to coding. This reflects the emphasis placed on correct construction of requirements, specification and design." <http://www.adacore.com/sparkpro/tokeneer/>
47. Dougherty, Chad. Kirk Sayer, Robert Seacord, David Svoboda, and Kazuya Togashi, "Secure Design Patterns, Software Engineering Institute, October 2009. <www.cert.org/archive/pdf/09tr010.pdf>. See also Architecture and Design Considerations for Secure Software, Version 2.0, 18 May 2012. <https://buildsecurityin.us-cert.gov/sites/default/files/ArchitectureAndDesign_PocketGuide_v2%200_05182012_PostOnline.pdf>.
48. MITRE. CWE-384: Session Fixation Document1<http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.htmlhttp://cwe.mitre.org/data/definitions/384.html> and CWE-613: Insufficient Session Expiration <https://cwe.mitre.org/data/definitions/613.html>
49. Department of Homeland Security, "Architecture and Design Considerations for Secure Software," Version 2.0, 18 May 2012. <https://buildsecurityin.us-cert.gov/sites/default/files/ArchitectureAndDesign_PocketGuide_v2%200_05182012_PostOnline.pdf>