CyBOK

SECURE SOFTWARE LIFECYCLE KNOWLEDGE AREA

Issue 1.0

AUTHOR: Laurie Williams – North Carolina State University

EDITOR: Andrew Martin – Oxford University

REVIEWERS:

Rod Chapman – Altran UK

Fabio Massacci – University of Trento

Gary McGraw – Synopsys

Nancy Mead – Carnegie Mellon University

James Noble – Victoria University Wellington

Riccardo Scandariato – University of Gothenburg

The CyBOK project would like to understand how the CyBOK is being used and its uptake. The project would like organisations using, or intending to use, CyBOK for the purposes of education, training, course development, professional development etc. to contact it at contact@cybok.org to let the project know how they are using CyBOK.

Issue 1.0 is a stable public release of the Secure Software Lifecycle Knowledge Area. However, it should be noted that a fully collated CyBOK document which includes all the Knowledge Areas is anticipated to be released in October 2019. This will likely include updated page layout and formatting of the individual Knowledge Areas.

Secure Software Lifecycle

Laurie Williams

August 2019

INTRODUCTION

The purpose of this Secure Software Lifecycle knowledge area is to provide an overview of software development processes for implementing secure software from the design of the software to the operational use of the software. This implementation may involve new coding as well as the incorporation of third party libraries and components. The goal of this overview is for use in academic courses in the area of software security; and to guide industry professionals who would like to use a secure software lifecycle.

The Software Security Knowledge Area in the Cyber Security Body of Knowledge provides a structured overview of secure software development and coding and the known categories of software implementation vulnerabilities and of techniques that can be used to prevent or detect such vulnerabilities or to mitigate their exploitation. By contrast, this Secure Software Lifecycle Knowledge Area focuses on the components of a comprehensive software development process to prevent and detect security defects and to respond in the event of an exploit.

This Knowledge Area will begin with a history of secure software lifecycle models. Section 2 provides examples of three prescriptive secure software lifecycle processes; the Microsoft Secure Development Lifecycle, Touchpoints, and SAFECode. Section 3 discusses how these processes can be adapted in six specific domains: agile/DevOps, mobile, cloud computing, internet of things, road vehicles, and ecommerce/payment card. Section 4 provides information on three frameworks for assessing an organisation's secure software lifecycle process.

CONTENT

1 Motivation

[1, 2, 3, 4, 5, 6, 7, 8, 9]

Historically, and at times currently, organisations have focused their security strategies at the network system level, such as with firewalls, and have taken a *reactive* approach to software security, using an approach commonly referred to as 'penetrate and patch'. [4] With this approach, security is assessed when the product is complete via penetration testing by attempting known attacks; or vulnerabilities are discovered post release when organisations are victims of an attack on deployed software. In either case, organisations then react by finding and fixing the vulnerability via a security patch. The following shortcomings are likely to be more prevalent with a predominantly reactive approach to cyber security:

- **Breaches are costly.** Based upon a study of 477 companies in 15 countries, in 2018 the Poneman Institute [3] reported that a breach cost, on average, 7.9 million US dollars in the United States and 5.3 million US dollars in the Middle East. Breaches were the least expensive

in India and Brazil, but these countries still spent an average of 1.8 million and 1.2 million US dollars per breach, respectively. Loss of reputation caused by a breach is difficult to quantify.

- **Attackers can find and exploit vulnerabilities without being noticed.** Based upon a study of 477 companies in 15 countries, in 2018 the Poneman Institute [3] reported that the mean time to identify that a breach had occurred was 197 days, and the mean time to find and fix a vulnerability once the breach was detected was an additional 69 days.
- **Patches can introduce new vulnerabilities or other problems.** Vulnerability patches are considered urgent and can be rushed out, potentially introducing new problems to a system. For example, Microsoft's early patches for the Meltdown[1] chip flaw introduced an even more serious vulnerability in Windows 7[2]. The new vulnerability allowed attackers to read kernel memory much faster and to write their own memory, and could allow an attacker to access every user-level computing process running on a machine.
- **Patches often go unapplied by customers.** Users and system administrators may be reluctant to apply security patches. For example, the highly-publicised Heartbleed[3] vulnerability in OpenSSL allows attackers to easily and quietly exploit vulnerable systems, stealing passwords, cookies, private crypto-keys, and much more. The vulnerability was reported in April 2014; but in January 2017 a scan revealed 200,000 Internet-accessible devices remained unpatched [6]. Once a vulnerability is publicly reported, attackers formulate a new mechanism to exploit the vulnerability with the knowledge that many organisations will not adopt the fix.

In 1998, McGraw [4] advocated moving beyond the penetrate and patch approach based upon his work on a DARPA-funded research effort investigating the application of software engineering to the assessment of software vulnerabilities. He contended that *proactive* rigorous software analysis should play an increasingly important role in assessing and preventing vulnerabilities in applications based upon the well-known fact that security violations occur because of errors in software design and coding. In 2002, Viega and McGraw published the first book on developing secure programs, *Building Secure Software* [5], with a focus on preventing the injection of vulnerabilities and reducing security risk through an integration of security into a software development process.

In the early 2000s, attackers became more aggressive, and Microsoft was a focus of this aggression with exposure of security weaknesses in their products, particularly the Internet Information Server (IIS). Gartner, a leading research and advisory company who seldom advises its clients to steer clear of specific software, advised companies to stop using IIS. In response to customer concerns and mounting bad press, the then Microsoft CEO, Bill Gates, sent the *Trustworthy Computing* memo [1] to all employees on January 15, 2002. The memo was also widely circulated on the Internet. An excerpt of the memo defines Trustworthy Computing:

> 'Trustworthy Computing is the highest priority for all the work we are doing. We must lead the industry to a whole new level of Trustworthiness in computing ... Trustworthy Computing is computing that is as available, reliable and secure as electricity, water services and telephony'.

The Trustworthy Computing memo caused a shift in the company. Two weeks later, Microsoft announced the delay of the release of Windows .NET Server [7] to ensure a proper security review (referred to as the Windows Security Push), as mandated by Microsoft's Trustworthy Computing initiative outlined in this memo. In 2003, Microsoft employees Howard and Le Blanc [8] publicly published a second edition of a book on writing secure code to prevent vulnerabilities, to detect design

[1] https://meltdownattack.com/ Meltdown lets hackers get around a barrier between applications and computer memory to steal sensitive data.

[2] https://www.cyberscoop.com/microsoft-meltdown-patches-windows-7-memory-management/

[3] http://heartbleed.com/

flaws and implementation bugs, and to improve test code and documentation. The first edition had been required reading for all members of the Windows team during the Push.

During the ensuing years, Microsoft changed their development process to build secure products through a comprehensive overhaul of their development process from early planning through product end-of-life. Their products contained demonstrably fewer vulnerabilities [8]. After internal use of the process, Microsoft codified and contributed their 13-stage internal development process, the Microsoft Security Development Lifecycle (SDL) to the community through its book entitled *The Security Development Lifecycle* [2] in 2006. True to Gates' original intent, the Microsoft SDL provided the foundation for the information technology industry by providing the first documented comprehensive and prescriptive lifecycle. Also in 2006, McGraw published the first book on software security best practices [9].

As discussed in the rest of this knowledge area, organisations have built upon the foundation set forth by Microsoft and by Viega and McGraw [5, 4].

2 Prescriptive Secure Software Lifecycle Processes

[2, 5, 8, 10, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36]

Secure software lifecycle processes are *proactive* approaches to building security into a product, treating the 'disease' of poorly designed, insecure software at the source, rather than 'applying a band aid' to stop the symptoms through a *reactive* penetrate and patch approach. These processes work software security deeply into the full product development process and incorporate people and technology to tackle and prevent software security issues. This section will provide information on three prominent secure software lifecycle processes and then reflect on the commonalities between them in Table 1.

2.1 Secure Software Lifecycle Processes

Three exemplar prescriptive secure software lifecycle processes are summarised in this section. The processes are *prescriptive* in that they explicitly recommend software practices. The three processes were chosen because the practices of these processes are integrated and cover a broad spectrum of the lifecycle phases, from software requirements to release/deployment and software maintenance. Two of these processes were identified in a systematic mapping study [24] on security approaches in software development lifecycles. As such, the practices span the prevention of security defects, the detection of security defects, and the mitigation of security defects once a product is in the field. The three were also chosen due to their maturity in terms of the number of years they have existed and in terms of their widespread acceptance in the industry. As will be discussed in Section 2.2, no 'best' secure software lifecycle process exists. Practitioners should consider incorporating practices from each of these processes into their own secure software process.

2.1.1 Microsoft Security Development Lifecycle (SDL)

As discussed in Section 1, Microsoft used an internal development process, the Security Development Lifecyle (SDL), to improve the security of their products. Howard and Lipner [2] disseminated a snapshot of this process in 2006. Since that time, Microsoft has continued to evolve their SDL and to provide up-to-date resources to the community [10], including an increased focus on compliance requirements that are being imposed on the industry.

Currently [10], the Microsoft SDL contains 12 practices which are enumerated below. For each of the practices, techniques for implementing the practice may be mentioned though the SDL does not prescribe the specific technique.

1. **Provide Training**. A range of professionals, such as developers, service engineers, program managers, product managers, and project managers, participate in the development of secure products while still addressing business needs and delivering user value. Software developers and architects must understand technical approaches for preventing and detecting vulnerabilities. The entire development organisation should be cognisant of the attacker's perspective, goals, and techniques; and of the business implications of not building secure products.

   Often, the formal education of these professionals does not include cyber security. Additionally, attack vectors, security tools, secure programming languages, and experiences are constantly evolving, so knowledge and course material must be refreshed. Ongoing cyber security training is essential for software organisations.

2. **Define Security Requirements**. Security requirements should be defined during the initial design and planning phases. Factors that influence these requirements include the specific functional requirements of the system, the legal and industry compliance requirements, internal and external standards, previous security incidents, and known threats.

   Techniques have been developed for systematically developing security requirements. For example, Security Quality Requirements Engineering (SQUARE) [25] is a nine-step process that helps organisations build security into the early stages of the production lifecycle. Abuse cases, as will be discussed in Section 2.1.2 bullet 5, are another means of specifying security requirements. van Lamsweerde extended the Keep All Objectives Satisfied (KAOS) framework for goal-based requirements specification language to include anti-models [26]. An anti-model is constructed by addressing malicious obstacles (called anti-goals) set up by attackers to threaten a system's security goals. An obstacle negates existing goals of the system. Secure i* [27] extends the i*-modeling framework with modeling and analysis of security trade-offs and aligns security requirements with other requirements.

   Security requirements must be continually updated to reflect changes in required functionality, standards, and the threat landscape.

3. **Define Metrics and Compliance Reporting**. Lord Kelvin is quoted as stating, 'If you can not measure it, you can not improve it'. The management team should understand and be held accountable for minimum acceptable levels of security using security metrics [12]. A subset of these metrics may be set as key performance indicators (KPIs) for management reporting. Defect tracking should clearly label security defects and security work items as such to allow for accurate prioritisation, risk management, tracking, and reporting of security work. Additionally, products increasingly must comply with regulatory standards, such as the Payment Card Industry Data Security Standard (PCI DSS)[4], or the EU General Data Protection Regulation (GDPR)[5], which may impose additional process steps and metrics for compliance, reporting, and audits.

4. **Perform Threat Modelling**. Through the use of threat modelling [13, 21], teams consider, document and discuss the security implications of designs in the context of their planned operational environment and in a structured fashion. Teams should consider the motivations of their adversaries and the strengths and weaknesses of systems to defend against the associated threat scenarios. An approach is to consider the (1) the malicious and benevolent interactors with the system; (2) the design of the system and its components (i.e. processes and data stores), (3) the trust boundaries of the system; and (4) the data flow of the system within and across trust boundaries to/from its interactors. Threats can be enumerated using a systematic approach of considering each system component relative to the STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) [8] threats:

[4] https://www.pcisecuritystandards.org/
[5] https://eugdpr.org/

(a) **Spoofing identity**. Spoofing threats allow an attacker to pose as another user or allow a rogue server to pose as a valid server.

(b) **Tampering with data**. Data tampering threats involves malicious modification of data.

(c) **Repudiation**. Repudiation threats are associated with users who deny performing an action without other parties having any way to prove otherwise.

(d) **Information disclosure**. Information disclosure threats involve the exposure of information to individuals who are not supposed to have access to it.

(e) **Denial of service**. A denial of service (DoS) attack denies service to valid users by making the system unavailable or unusable.

(f) **Elevation of privilege**. An unprivileged user gains privileged access and thereby has sufficient access to compromise or destroy the system.

Threat modelling aids the team in enumerating threats, so that the system design can be fortified and security features can be selected. In addition to STRIDE, other models exist to formulate threat models, such as 12 methods[6], including attack trees [28] which are conceptual diagrams of threats on systems and possible attacks to reach those threats. A closely-related practice to threat modelling is Architectural Risk Analysis, as will be discussed in Section 2.1.2 bullet 2.

Games have been created to aid teams in collaboratively (and competitively) conducting threat modeling:

(a) Elevation of Privilege[7]

(b) Security Cards[8]

(c) Protection Poker [29]

5. **Establish Design Requirements**. Design requirements guide the implementation of 'secure features' (i.e., features that are well engineered with respect to security). Additionally, the architecture and design must be resistant to known threats in the intended operational environment.

   The design of secure features involves abiding by the timeless security principles set forth by Saltzer and Schroeder [15] in 1975 and restated by Viega and McGraw [5] in 2002. The eight Saltzer and Schroeder principles are:

   - **Economy of mechanism**. Keep the design of the system as simple and small as possible.
   - **Fail-safe defaults**. Base access decisions on permissions rather than exclusion; the default condition is lack of access and the protection scheme identifies conditions under which access is permitted. Design a security mechanism so that a failure will follow the same execution path as disallowing the operation.
   - **Complete mediation**. Every access to every object must be checked for authorisation.
   - **Open design**. The design should not depend upon the ignorance of attackers but rather on the possession of keys or passwords.
   - **Separation of privilege**. A protection mechanism that requires two keys to unlock is more robust than one that requires a single key when two or more decisions must be made before access should be granted.
   - **Least privilege**. Every program and every user should operate using the least set of privileges necessary to complete the job.

[6] https://insights.sei.cmu.edu/sei_blog/2018/12/threatmodeling2availablemethods.html
[7] https://www.usenix.org/conference/3gse14/summit-program/presentation/shostack
[8] https://securitycards.cs.washington.edu/

- **Least common mechanism**. Minimise the amount of mechanisms common to more than one user and depended on by all users.
- **Psychological acceptability**. The human interface should be designed for ease of use so that users routinely and automatically apply the mechanisms correctly and securely.

Two other important secure design principles include the following:

- **Defense in depth**. Provide multiple layers of security controls to provide redundancy in the event a security breach.
- **Design for updating**. The software security must be designed for change, such as for security patches and security property changes.

Design requirements also involve the selection of security features, such as cryptography, authentication and logging to reduce the risks identified through threat modelling. Teams also take actions to reduce the attack surface of their system design. The attack surface, a concept introduced by Howard [14] in 2003, can be thought of as the sum of the points where attackers can try to enter data to or extract data from a system [23, 22].

In 2014, the IEEE Center for Secure Design [16] enumerated the top ten security design flaws and provided guidelines on techniques for avoiding them. These guidelines are as follows:

(a) Earn or give, but never assume, trust.

(b) Use an authentication mechanism that cannot be bypassed or tampered with.

(c) Authorise after you authenticate.

(d) Strictly separate data and control instructions, and never process control instructions received from untrusted sources.

(e) Define an approach that ensures all data are explicitly validated.

(f) Use cryptography correctly.

(g) Identify sensitive data and how they should be handled.

(h) Always consider the users.

(i) Understand how integrating external components changes your attack surface.

(j) Be flexible when considering future changes to objects and actors.

6. **Define and Use Cryptography Standards**. The use of cryptography is an important design feature for a system to ensure security- and privacy-sensitive data is protected from unintended disclosure or alteration when it is transmitted or stored. However, an incorrect choice in the use of cryptography can render the intended protection weak or ineffective. Experts should be consulted in the use of clear encryption standards that provide specifics on every element of the encryption implementation and on the use of only properly vetted encryption libraries. Systems should be designed to allow the encryption libraries to be easily replaced, if needed, in the event the library is broken by an attacker, such as was done to the Data Encryption Standard (DES) through 'Deep Crack'[9], a brute force search of every possible key as designed by Paul Kocher, president of Cryptography Research.

7. **Manage the Security Risk of Using Third-Party Components**. The vast majority of software projects are built using proprietary and open-source third-party components. The Black Duck On-Demand audit services group [17] conducted open-source audits on over 1,100 commercial applications and found open-source components in 95% of the applications with an average 257 components per application. Each of these components can have vulnerabilities upon adoption or in the future. An organisation should have an accurate inventory of third-party

[9] https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html

components [31], continuously use a tool to scan for vulnerabilities in its components, and have a plan to respond when new vulnerabilities are discovered. Freely available and proprietary tools can be used to identify project component dependencies and to check if there are any known, publicly disclosed, vulnerabilities in these components.

8. **Use Approved Tools**. An organisation should publish a list of approved tools and their associated security checks and settings such as compiler/linker options and warnings. Engineers should use the latest version of these tools, such as compiler versions, and take advantage of new security analysis functionality and protections. Often, the resultant software must be backward compatible with previous versions.

9. **Perform Static Analysis Security Testing (SAST)**. SAST tools can be used for an automated security code review to find instances of insecure coding patterns and to help ensure that secure coding policies are being followed. SAST can be integrated into the commit and deployment pipeline as a check-in gate to identify vulnerabilities each time the software is built or packaged. For increased efficiency, SAST tools can integrate into the developer environment and be run by the developer during coding. Some SAST tools spot certain implementation bugs, such as the existence of unsafe or other banned functions and automatically replace with (or suggest) safer alternatives as the developer is actively coding. See also Section 3.1 (Static Detection) in the Software Security knowledge area in the Cyber Security Body of Knowledge.

10. **Perform Dynamic Analysis Security Testing (DAST)**. DAST performs run-time verification of compiled or packaged software to check functionality that is only apparent when all components are integrated and running. DAST often involves the use of a suite of pre-built attacks and malformed strings that can detect memory corruption, user privilege issues, injection attacks, and other critical security problems. DAST tools may employ *fuzzing*, an automated technique of inputting known invalid and unexpected test cases at an application, often in large volume. Similar to SAST, DAST can be run by the developer and/or integrated into the build and deployment pipeline as a check-in gate. DAST can be considered to be automated penetration testing. See also Section 3.2 (Dynamic Detection) in the Software Security knowledge area in the Cyber Security Body of Knowledge.

11. **Perform Penetration Testing**. Manual penetration testing is black box testing of a running system to simulate the actions of an attacker. Penetration testing is often performed by skilled security professionals, who can be internal to an organisation or consultants, opportunistically simulating the actions of a hacker. The objective of a penetration test is to uncover any form of vulnerability - from small implementation bugs to major design flaws resulting from coding errors, system configuration faults, design flaws or other operational deployment weaknesses. Tests should attempt both unauthorised misuse of and access to target assets and violations of the assumptions. A widely-referenced resource for structuring penetration tests is the OWASP Top 10 Most Critical Web Application Security Risks[10]. As such, penetration testing can find the broadest variety of vulnerabilities, although usually less efficiently compared with SAST and DAST [18]. Penetration testers can be referred to as white hat hackers or ethical hackers. In the penetration and patch model, penetration testing was the only line of security analysis prior to deploying a system.

12. **Establish a Standard Incident Response Process**. Despite a secure software lifecycle, organisations must be prepared for inevitable attacks. Organisations should proactively prepare an Incident Response Plan (IRP). The plan should include who to contact in case of a security emergency, establish the protocol for efficient vulnerability mitigation, for customer response and communication, and for the rapid deployment of a fix. The IRP should include plans for code inherited from other groups within the organisation and for third-party code. The IRP

[10] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

should be tested before it is needed. Lessons learned through responses to actual attack should be factored back into the SDL.

2.1.2 Touchpoints

International software security consultant, Gary McGraw, provided seven Software Security Touchpoints [9] by codifying extensive industrial experience with building secure products. McGraw uses the term *touchpoint* to refer to software security best practices which can be incorporated into a secure software lifecycle. McGraw differentiates vulnerabilities that are implementation bugs and those that are design flaws [16]. *Implementation bugs* are localized errors, such as buffer overflow and input validation errors, in a single piece of code, making spotting and comprehension easier. *Design flaws* are systemic problems at the design level of the code, such as error-handling and recovery systems that fail in an insecure fashion or object-sharing systems that mistakenly include transitive trust issues [9]. Kuhn et al. [31] analysed the 2008 - 2016 vulnerability data from the US National Vulnerability Database (NVD)[11] and found that 67% of the vulnerabilities were implementation bugs. The seven touchpoints help to prevent and detect both bugs and flaws.

These seven touchpoints are described below and are provided in order of effectiveness based upon McGraw's experience with the utility of each practice over many years, hence prescriptive:

1. **Code Review (Tools)**.

   Code review is used to detect implementation bugs. Manual code review may be used, but requires that the auditors are knowledgeable about security vulnerabilities before they can rigorously examine the code. 'Code review with a tool' (a.k.a. the use of static analysis tools or SAST) has been shown to be effective and can be used by engineers that do not have expert security knowledge. For further discussion on static analysis, see Section 2.1.1 bullet 9.

2. **Architectural Risk Analysis**.

   Architectural risk analysis, which can also be referred to as threat modelling (see Section 2.1.1 bullet 4), is used to prevent and detect design flaws. Designers and architects provide a high-level view of the target system and documentation for assumptions, and identify possible attacks. Through architectural risk analysis, security analysts uncover and rank architectural and design flaws so mitigation can begin. For example, risk analysis may identify a possible attack type, such as the ability for data to be intercepted and read. This identification would prompt the designers to look at all their code's traffics flows to see if interception was a worry, and whether adequate protection (i.e. encryption) was in place. That review that the analysis prompted is what uncovers design flaws, such as sensitive data is transported in the clear.

   No system can be perfectly secure, so risk analysis must be used to prioritise security efforts and to link system-level concerns to probability and impact measures that matter to the business building the software. Risk exposure is computed by multiplying the probability of occurrence of an adverse event by the cost associated with that event [32].

   McGraw proposes three basic steps for architectural risk analysis:

   - *Attack resistance analysis*. Attack resistance analysis uses a checklist/systematic approach of considering each system component relative to *known* threats, as is done in Microsoft threat modelling discussed in Section 2.1.1 bullet 4. Information about known attacks and attack patterns are used during the analysis, identifying risks in the architecture and understanding the viability of known attacks. Threat modelling with the incorporation of STRIDE-based attacks, as discussed in Section 2.1.1 bullet 4, is an example process for performing attack resistance analysis.

[11] http://nvd.nist.gov

- *Ambiguity analysis*. Ambiguity analysis is used to capture the creative activity required to discover *new* risks. Ambiguity analysis requires two or more experienced analysts who carry out separate analysis activities in parallel on the same system. Through unifying the understanding of multiple analysis, disagreements between the analysts can uncover ambiguity, inconsistency and new flaws.
- *Weakness analysis*. Weakness analysis is focused on understanding risk related to security issues in other third-party components (see Section 2.1.1 bullet 7). The idea is to understand the assumptions being made about third-party software and what will happen when those assumptions fail.

Risk identification, ranking, and mitigation is a continuous process through the software lifecycle, beginning with the requirement phase.

3. **Penetration Testing**.

   Penetration testing can be guided by the outcome of architectural risk analysis (See Section 2.1.2 bullet 2). For further discussion on penetration testing, see Section 2.1.1, bullet 11.

4. **Risk-based Security Testing**.

   Security testing must encompass two strategies: (1) testing of security functionality with standard functional testing techniques; and (2) risk-based testing based upon attack patterns and architectural risk analysis results (see Section 2.1.2 bullet 2), and abuse cases (see Section 2.1.2 bullet 5). For web applications, testing of security functionality can be guided by the OWASP Application Security Verification Standard (ASVS) Project[12] open standard for testing application technical security controls. ASVS also provides developers with a list of requirements for secure development.

   Guiding tests with knowledge of the software architecture and construction, common attacks, and the attacker's mindset is extremely important. Using the results of architectural risk analysis, the tester can properly focus on areas of code where an attack is likely to succeed.

   The difference between risk-based testing and penetration testing is the level of the approach and the timing of the testing. Penetration testing is done when the software is complete and installed in an operational environment. Penetration tests are outside-in, black box tests. Risk-based security testing can begin before the software is complete and even pre-integration, including the use of white box unit tests and stubs. The two are similar in that they both should be guided by risk analysis, abuse cases and functional security requirements.

5. **Abuse Cases**.

   This touchpoint codifies 'thinking like an attacker'. Use cases describe the desired system's behaviour by benevolent actors. Abuse cases [19] describe the system's behaviour when under attack by a malicious actor. To develop abuse cases, an analyst enumerates the types of malicious actors who would be motivated to attack the system. For each bad actor, the analyst creates one or more abuse case(s) for the functionality the bad actor desires from the system. The analyst then considers the interaction between the use cases and the abuse cases to fortify the system. Consider an automobile example. An actor is the driver of the car, and this actor has a use case 'drive the car'. A malicious actor is a car thief whose abuse case is 'steal the car'. This abuse case *threatens* the use case. To prevent the theft, a new use case 'lock the car' can be added to *mitigate* the abuse case and fortify the system.

   Human error is responsible for a large number of breaches. System analysts should also consider actions by benevolent users, such as being the victim of a phishing attack, that result in

[12] https://www.owasp.org/index.php/Category:OWASP_Application/_Security_Verification_Standard_Projecttab=Home

a security breach. These actions can be considered misuse cases [20] and should be analysed similarly to abuse cases, considering what use case the misuse case threatens and the fortification to the system to mitigate the misuse case.

The attacks and mitigations identified by the abuse and misuse case analysis can be used as input into the security requirements (Section 2.1.1 bullet 2.); penetration testing (Section 2.1.1 bullet 11); and risk-based security testing (Section 2.1.2 bullet 4).

6. **Security Requirements**.

   For further discussion on security requirements, see Section 2.1.1 bullet 2.

7. **Security Operations**.

   Network security can integrate with software security to enhance the security posture. Inevitably, attacks will happen, regardless of the applications of the other touchpoints. Understanding attacker behaviour and the software that enabled a successful attack is an essential defensive technique. Knowledge gained by understanding attacks can be fed back into the six other touchpoints.

The seven touchpoints are intended to be cycled through multiple times as the software product evolves. The touchpoints are also process agnostic, meaning that the practices can be included in any software development process.

2.1.3 SAFECode

The Software Assurance Forum for Excellence in Code (SAFECode)[13] is a non-profit, global, industry-led organisation dedicated to increasing trust in information and communications technology products and services through the advancement of effective software assurance methods. The SAFECode mission is to promote best practices for developing and delivering more secure and reliable software, hardware and services. The SAFECode organisation publishes the 'Fundamental practices for secure software development: Essential elements of a secure development lifecycle program' [33] guideline to foster the industry-wide adoption of fundamental secure development practices. The fundamental practices deal with assurance – the ability of the software to withstand attacks that attempt to exploit design or implementation errors. The eight fundamental practices outlined in their guideline are described below:

1. **Application Security Control Definition**. SAFECode uses the term Application Security Controls (ASC) to refer to security requirements (see Section 2.1.1 bullet 2). Similarly, NIST 800-53 [34] uses the phrase *security control* to refer to security functionality and security assurance requirements.

   The inputs to ASC include the following: secure design principles (see Section 2.1.3 bullet 3); secure coding practices; legal and industry requirements with which the application needs to comply (such as HIPAA, PCI, GDPR, or SCADA); internal policies and standards; incidents and other feedback; threats and risk. The development of ASC begins before the design phase and continues throughout the lifecycle to provide clear and actionable controls and to be responsive to changing business requirements and the ever-evolving threat environment.

2. **Design**. Software must incorporate security features to comply with internal security practices and external laws or regulations. Additionally, the software must resist known threats based upon the operational environment. (see Section 2.1.1 bullet 5.) Threat modelling (see Section 2.1.1 bullet 4), architectural reviews, and design reviews can be used to identify and address design flaws before their implementation into source code.

[13] https://safecode.org/

The system design should incorporate an encryption strategy (see Section 2.1.1 bullet 6) to protect sensitive data from unintended disclosure or alteration while the data are at rest or in transit.

The system design should use a standardised approach to identity and access management to perform authentication and authorisation. The standardisation provides consistency between components and clear guidance on how to verify the presence of the proper controls. Authenticating the identity of a principal (be it a human user, other service or logical component) and verifying the authorisation to perform an action are foundational controls of the system. Several access control schemes have been developed to support authorisation: mandatory, discretionary, role-based or attribute-based. Each of these has benefits and drawbacks and should be chosen based upon project characteristics.

Log files provide the evidence needed in forensic analysis when a breach occurs to mitigate repudiation threats. In a well-designed application, system and security log files provide the ability to understand an application's behaviour and how it is used at any moment, and to distinguish benevolent user behaviour from malicious user behaviour. Because logging affects the available system resources, the logging system should be designed to capture the critical information while not capturing excess data. Policies and controls need to be established around storing, tamper prevention and monitoring log files. OWASP provides valuable resources on designing and implementing logging[14][15].

3. **Secure Coding Practices**. Unintended code-level vulnerabilities are introduced by programmer mistakes. These types of mistakes can be prevented and detected through the use of coding standards; selecting the most appropriate (and safe) languages, frameworks and libraries, including the use of their associated security features (see Section 2.1.1 bullet 8); using automated analysis tools (see Section 2.1.1 bullets 9 and 10); and manually reviewing the code.

   Organisations provide standards and guidelines for secure coding, for example:

   (a) OWASP Secure Coding Practices, Quick Reference Guide [16]

   (b) Oracle Secure Coding Guidelines for Java SE [17]

   (c) Software Engineering Institute (SEI) CERT Secure Coding Standards [18]

   Special care must also be given to handling unanticipated errors in a controlled and graceful way through generic error handlers or exception handlers that log the events. If the generic handlers are invoked, the application should be considered to be in an unsafe state such that further execution is no longer considered trusted.

4. **Manage Security Risk Inherent in the Use of Third-Party Components**. See Section 2.1.1 bullet 7.

5. **Testing and Validation**. See Section 2.1.1 bullets 9-11 and Section 2.1.2 bullets 1, 3 and 4.

6. **Manage Security Findings**. The first five practices produce artifacts that contain or generate findings related to the security of the product (or lack thereof). The findings in these artifacts should be tracked and actions should be taken to remediate vulnerabilities, such as is laid out in the Common Criteria (see Section 4.3) flaw remediation procedure [35]. Alternatively, the team may consciously accept the security risk when the risk is determined to be acceptable.

[14] https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
[15] https://www.owasp.org/images/e/e0/OWASP_Logging_Guide.pdf
[16] https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf
[17] https://www.oracle.com/technetwork/java/seccodeguide-139067.html
[18] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

Acceptance of risk must be tracked, including a severity rating; a remediation plan, an expiration or a re-review deadline; and the area for re-review/validation.

Clear definitions of severity are important to ensure that all participants have and communicate with a consistent understanding of a security issue and its potential impact. A possible starting point is mapping to the severity levels, attributes, and thresholds used by the Common Vulnerability Scoring System (CVSS)[19] such as 10–8.5 is critical, 8.4–7.0 is high, etc. The severity levels are used to prioritise mitigations based upon their complexity of exploitation and impact on the properties of a system.

7. **Vulnerability Response and Disclosure**. Even with following a secure software lifecycle, no product can be 'perfectly secure' because of the constantly changing threat landscapes. Vulnerabilities will be exploited and the software will eventually be compromised. An organisation must develop a vulnerability response and disclosure process to help drive the resolution of externally discovered vulnerabilities and to keep all stakeholders informed of progress. ISO provides industry-proven standards[20] for vulnerability disclosure and handling. To prevent vulnerabilities from re-occurring in new or updated products, the team should perform a root cause analysis and feed the lessons learned into the secure software lifecycle practices. For further discussion, see Sections 2.1.1 bullet 12 and 2.1.2 bullet 7.

8. **Planning the Implementation and Deployment of Secure Development**. A healthy and mature secure development lifecycle includes the above seven practices but also an integration of these practices into the business process and the entire organisation, including program management, stakeholder management, deployment planning, metrics and indicators, and a plan for continuous improvement. The culture, expertise and skill level of the organisation needs to be considered when planning to deploy a secure software lifecycle. Based upon past history, the organisation may respond better to a corporate mandate, to a bottom-up groundswell approach or to a series of pilot programs. Training will be needed (see Section 2.1.1 bullet 1). The specification of the organisation's secure software lifecycle including the roles and responsibilities should be documented. Plans for compliance and process health should be made (see Section 4).

2.2 Comparing the Secure Software Lifecycle Models

In 2009, De Win et al. [36] compared CLASP, Microsoft's originally-documented SDL [2], and Touchpoints (see Section 2.1.2)for the purpose of providing guidance on their commonalities and the specificity of the approach, and making suggestions for improvement. The authors mapped the 153 possible activities of each lifecycle model into six software development phases: education and awareness; project inception; analysis and requirements; architectural and detailed design; implementation and testing; and release, deployment and support. The activities took the practices in Sections 2.1.1–2.1.3 into much finer granularity. The authors indicated whether each model includes each of the 153 activities and provides guidance on the strengths and weaknesses of each model. The authors found no clear comprehensive 'winner' among the models, so practitioners could consider using guidelines for the desired fine-grained practices from all the models.

Table 1 places the the practices of Sections 2.1.1–2.1.3 into the six software development phases used by De Win et al. [36]. Similar to prior work [36], the models demonstrate strengths and weaknesses in terms of guidance for the six software development phases. No model can be considered perfect for all contexts. Security experts can customize a model for their organizations considering the spread of practices for the six software development phases.

[19] https://www.first.org/cvss/

[20] https://www.iso.org/standard/45170.html and https://www.iso.org/standard/53231.html

Table 1: Comparing the Software Security Lifecycle Models

	Microsoft SDL	Touchpoints	SAFECode
Education and awareness	• Provide training		• Planning the implementation and deployment of secure development
Project inception	• Define metrics and compliance reporting • Define and use cryptography standards • Use approved tools		• Planning the implementation and deployment of secure development
Analysis and requirements	• Define security requirements • Perform threat modelling	• Abuse cases • Security requirements	• Application security control definition
Architectural and detailed design	• Establish design requirements	• Architectural risk analysis	• Design
Implementation and testing	• Perform static analysis security testing (SAST) • Perform dynamic analysis security testing (DAST) • Perform penetration testing • Define and use cryptography standards • Manage the risk of using third-party components	• Code review (tools) • Penetration testing • Risk-based security testing	• Secure coding practices • Manage security risk inherent in the use of third-party components • Testing and validation
Release, deployment, and support	• Establish a standard incident response process	• Security operations	• Vulnerability response and disclosure

3 Adaptations of the Secure Software Lifecycle

[33, 37, 38, 39, 40, 41, 42, 43, 44, 45]

The secure software lifecycle models discussed in Section 2.1 can be integrated with any software development model and are domain agnostic. In this section, information on six adaptations to secure software lifecycle is provided.

3.1 Agile Software Development and DevOps

Agile and continuous software development methodologies are highly iterative, with new functionality being provided to customers frequently - potentially as quickly as multiple times per day or as 'slowly' as every two to four weeks.

Agile software development methodologies can be functional requirement-centric, with the functionality being expressed as *user stories*. SAFECode [37] provides practical software security guidance to agile practitioners. This guidance includes a set of 36 recommended security-focused stories that can be handled similarly to the functionality-focused user stories. These stories are based upon common security issues such as those listed in the OWASP Top 10[21] Most Critical Web Application Security Risks. The stories are mapped to Common Weakness Enumerations (CWE™)[22] identifiers, as applicable. The security-focused stories are worded in a format similar to functionality stories (i.e., As a [stakeholder], I want to [new functionality] so that I can [goal]). For example, a security-focused story using this format is provided: *As Quality Assurance, I want to verify that all users have access to the specific resources they require which they are authorised to use*, that is mapped to CWE-862 and CWE-863. The security-focused stories are further broken down into manageable and concrete tasks that are owned by team roles, including architects, developers, testers and security experts, and are mapped to SAFECode Fundamental Practices [33]. Finally, 17 operational security tasks were specified by SAFECode. These tasks are not directly tied to stories but are handled as continuous maintenance work (such as, *Continuously verify coverage of static code analysis tools*) or as an item requiring special attention (such as, *Configure bug tracking to track security vulnerabilities*).

With a DevOps approach to developing software, development and operations are tightly integrated to enable fast and continuous delivery of value to end users. Microsoft has published a DevOps secure software lifecycle model [38] that includes activities for operations engineers to provide fast and early feedback to the team to build security into DevOps processes. The Secure DevOps model contains eight practices, including eight of the 12 practices in the Microsoft Security Development Lifecycle discussed in Section 2.1.1:

1. **Provide Training**. The training, as outlined in Section 2.1.1 bullet 1, must include the operations engineers. The training should encompass attack vectors made available through the deployment pipeline.
2. **Define Requirements**. See Section 2.1.1 bullet 2.
3. **Define Metrics and Compliance Reporting**. See Section 2.1.1 bullet 3.
4. **Use Software Composition Analysis (SCA) and Governance**. When selecting both commercial and open-source third-party components, the team should understand the impact that a vulnerability in the component could have on the overall security of the system and consider performing a more thorough evaluation before using them. Software Composition Analysis

[21] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[22] https://cwe.mitre.org/; CWE is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

(SCA) tools, such as WhiteSource[23] can assist with licensing exposure, provide an accurate inventory of components, and report any vulnerabilities with referenced components. See also Section 2.1.1 bullet 7.

5. **Perform Threat Modelling**. See Section 2.1.1 bullet 4. Threat modelling may be perceived as slowing down the rapid DevOps pace. However, products that are deployed rapidly under a DevOps deployment process should have a defined overall architecture within which the DevOps process makes changes and adds features. That architecture should be threat modeled, and when the team needs to change the architecture the threat model should also be updated. New features that do not have an architectural impact represent a null change to the threat model.

6. **Use Tools and Automation**. See Section 2.1.1 bullets 8, 9 and 10. The team should carefully select tools that can be integrated into the engineer's integrated development environment (IDE) and workflow such that they cause minimal disruption. The goal of using these tools is to detect defects and vulnerabilities and not to overload engineers with too many tools or alien processes outside of their everyday engineering experience. The tools used as part of a secure DevOps workflow should adhere to the following principles:

   (a) Tools must be integrated into the continuous integration/continuous delivery (CI/CD) pipeline.

   (b) Tools must not require security expertise beyond what is imparted by the training.

   (c) Tools must avoid a high false-positive rate of reporting issues.

7. **Keep Credentials Safe**. Scanning for credentials and other sensitive content in source files is necessary during pre-commit to reduce the risk of propagating the sensitive information through the CI/CD process, such as through Infrastructure as Code or other deployment scripts. Tools, such as CredScan[24], can identify credential leaks, such as those in source code and configuration files. Some commonly found types of credentials include default passwords, hard-coded passwords, SQL connection strings and Certificates with private keys.

8. **Use Continuous Learning and Monitoring**. Rapidly-deployed systems often monitor the health of applications, infrastructure and networks through instrumentation to ensure the systems are behaving 'normally'. This monitoring can also help uncover security and performance issues which are departures from normal behaviour. Monitoring is also an essential part of supporting a defense-in-depth strategy and can reduce an organisation's mean-time-to-identify (MTTI) and mean-time-to-contain (MTTC) an attack.

3.2 Mobile

Security concerns for mobile apps differ from traditional desktop software in some important ways, including local data storage, inter-app communication, proper usage of cryptographic APIs and secure network communication. The OWASP Mobile Security Project [39] is a resource for developers and security teams to build and maintain secure mobile applications; see also the *Web and Mobile* Knowledge Area.

Four resources are provided to aid in the secure software lifecycle of mobile applications:

1. **OWASP Mobile Application Security Verification Standard (MASVS) Security Requirements and Verification**. The MASVS defines a mobile app security model and lists generic security requirements for mobile apps. The MASVS can be used by architects, developers, testers, security professionals, and consumers to define and understand the qualities of a secure mobile app.

[23] https://www.whitesourcesoftware.com/

[24] https://secdevtools.azurewebsites.net/helpcredscan.html

2. **Mobile Security Testing Guide (MSTG)**. The guide[25] is a comprehensive manual for mobile application security testing and reverse engineering for iOS and Android mobile security testers. The guide provides the following content:

   (a) *A general mobile application testing guide* that contains a mobile app security testing methodology and general vulnerability analysis techniques as they apply to mobile app security. The guide also contains additional technical test cases that are operating system independent, such as authentication and session management, network communications, and cryptography.

   (b) *Operating system-dependent testing guides* for mobile security testing on the Android and iOS platforms, including security basics; security test cases; reverse engineering techniques and prevention; and tampering techniques and prevention.

   (c) *Detailed test cases* that map to the requirements in the MASVS.

3. **Mobile App Security Checklist**. The checklist[26] is used for security assessments and contains links to the MSTG test case for each requirement.

4. **Mobile Threat Model**. The threat model [40] provides a checklist of items that should be documented, reviewed and discussed when developing a mobile application. Five areas are considered in the threat model:

   (a) **Mobile Application Architecture**. The mobile application architecture describes device-specific features used by the application, wireless transmission protocols, data transmission medium, interaction with hardware components and other applications. The attack surface can be assessed through a mapping to the architecture.

   (b) **Mobile Data**. This section of the threat model defines the data the application stores, transmits and receives. The data flow diagrams should be reviewed to determine exactly how data are handled and managed by the application.

   (c) **Threat Agent Identification**. The threat agents are enumerated, including humans and automated programs.

   (d) **Methods of Attack**. The most common attacks utilised by threat agents are defined so that controls can be developed to mitigate attacks.

   (e) **Controls**. The controls to mitigate attacks are defined.

3.3 Cloud Computing

The emergence of cloud computing bring unique security risks and challenges. In conjunction with the Cloud Security Alliance (CSA)[27], SAFECode has provided a 'Practices for Secure Development of Cloud Applications' [41] guideline as a supplement to the 'Fundamental Practices for Secure Software Development' guideline [33] discussed in Section 2.1.3. - see also the *Distributed Systems* knowledge area. The Cloud guideline provides additional secure development recommendations to address six threats unique to cloud computing and to identify specific security design and implementation practices in the context of these threats. These threats and associated practices are provided:

1. **Threat: Multitenancy**. Multitenancy allows multiple consumers or tenants to maintain a presence in a cloud service provider's environment, but in a manner where the computations, processes, and data (both at rest and in transit) of one tenant are isolated from and inaccessible to another tenant. Practices:

[25] https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide
[26] https://github.com/OWASP/owasp-mstg/tree/master/Checklists
[27] https://cloudsecurityalliance.org/

(a) Model the application's interfaces in threat models. Ensure that the multitenancy threats, such as information disclosure and privilege elevation are modeled for each of these interfaces, and ensure that these threats are mitigated in the application code and/or configuration settings.

(b) Use a 'separate schema' database design and tables for each tenant when building multitenant applications rather than relying on a 'TenantID' column in each table.

(c) When developing applications that leverage a cloud service provider's Platform as a Service (PaaS) services, ensure common services are designed and deployed in a way that ensures that the tenant segregation is maintained.

2. **Tokenisation of Sensitive Data**. An organisation may not wish to generate and store intellectual property in a cloud environment not under its control. Tokenisation is a method of removing sensitive data from systems where they do not need to exist or disassociating the data from the context or the identity that makes them sensitive. The sensitive data are replaced with a token for those data. The token is later used to rejoin the sensitive data with other data in the cloud system. The sensitive data are encrypted and secured within an organisation's central system which can be protected with multiple layers of protection and appropriate redundancy for disaster recovery and business continuity. Practices:

   (a) When designing a cloud application, determine if the application needs to process sensitive data and if so, identify any organisational, government, or industry regulations that pertain to that type of sensitive data and assess their impact on the application design.

   (b) Consider implementing tokenisation to reduce or eliminate the amount of sensitive data that need to be processed and or stored in cloud environments.

   (c) Consider data masking, an approach that can be used in pre-production test and debug systems in which a representative data set is used, but does not need to have access to actual sensitive data. This approach allows the test and debug systems to be exempt from sensitive data protection requirements.

3. **Trusted Compute Pools**. Trusted Compute Pools are either physical or logical groupings of compute resources/systems in a data centre that share a security posture. These systems provide measured verification of the boot and runtime infrastructure for measured launch and trust verification. The measurements are stored in a trusted location on the system (referred to as a trusted platform module or TPM) and verification occurs when an agent, service or application requests the trust quote from the TPM. Practices:

   (a) Ensure the platform for developing cloud applications provides trust measurement capabilities and the APIs and services necessary for your applications to both request and verify the measurements of the infrastructure they are running on.

   (b) Verify the trust measurements as either part of the initialisation of your application or as a separate function prior to launching the application.

   (c) Audit the trust of the environments your applications run on using attestation services or native attestation features from your infrastructure provider.

4. **Data Encryption and Key Management**. Encryption is the most pervasive means of protecting sensitive data both at rest and in transit. When encryption is used, both providers and tenants must ensure that the associated cryptographic key materials are properly generated, managed and stored. Practices:

   (a) When developing an application for the cloud, determine if cryptographic and key management capabilities need to be directly implemented in the application or if the application can leverage cryptographic and key management capabilities provided by the PaaS environment.

(b) Make sure that appropriate key management capabilities are integrated into the application to ensure continued access to data encryption keys, particularly as the data move across cloud boundaries, such as enterprise to cloud or public to private cloud.

5. **Authentication and Identity Management**. As an authentication consumer, the application may need to authenticate itself to the PaaS to access interfaces and services provided by the PaaS. As an authentication provider, the application may need to authenticate the users of the application itself. Practices:

   (a) Cloud application developers should implement the authentication methods and credentials required for accessing PaaS interfaces and services.

   (b) Cloud application developers need to implement appropriate authentication methods for their environments (private, hybrid or public).

   (c) When developing cloud applications to be used by enterprise users, developers should consider supporting single sign on (SSO) solutions.

6. **Shared-Domain Issues**. Several cloud providers offer domains that developers can use to store user content, or for staging and testing their cloud applications. As such, these domains, which may be used by multiple vendors, are considered 'shared domains' when running client-side script (such as JavaScript) and from reading data. Practices:

   (a) Ensure that your cloud applications are using custom domains whenever the cloud provider's architecture allows you to do so.

   (b) Review your source code for any references to shared domains.

The European Union Agency for Network and Information Security (ENISA) [43] conducted an in-depth and independent analysis of the information security benefits and key security risks of cloud computing. The analysis reports that the massive concentrations of resources and data in the cloud present a more attractive target to attackers, but cloud-based defences can be more robust, scalable and cost-effective.

3.4 Internet of Things (IoT)

The Internet of Things (IoT) is utilised in almost every aspect of our daily life, including the extension into industrial sectors and applications (i.e. industrial IoT or IIoT). IoT and IIoT constitute an area of rapid growth that presents unique security challenges. [From this point forth we include IIoT when we use IoT.] Some of these are considered in the *Cyber-Physical Systems* knowledge area, but we consider specifically software lifecycle issues here. Devices must be securely provisioned, connectivity between these devices and the cloud must be secure, and data in storage and in transit must be protected. However, the devices are small, cheap, resource-constrained. Building security into each device may not be considered to be cost effective by its manufacturer, depending upon the value of the device and the importance of the data it collects. An IoT-based solution often has a large number of geographically-distributed devices. As a result of these technical challenges, trust concerns exist with the IoT, most of which currently have no resolution and are in need of research. However, the US National Institute of Standards and Technology (NIST) [42] recommends four practices for the development of secure IoT-based systems.

1. **Use of Radio Frequency Identifier (RFID) tags**. Sensors and their data may be tampered with, deleted, dropped, or transmitted insecurely. Counterfeit 'things' exist in the marketplace. Unique identifiers can mitigate this problem by attaching Radio Frequency Identifier (RFID) tags to devices. Readers activate a tag, causing the device to broadcast radio waves within a bandwidth reserved for RFID usage by governments internationally. The radio waves transmit identifiers or codes that reference unique information associated with the device.

2. **Not using or allowing the use of default passwords or credentials**. IoT devices are often not developed to require users and administrators to change default passwords during system set up. Additionally, devices often lack intuitive user interfaces for changing credentials. Recommended practices are to require passwords to be changed or to design in intuitive interfaces. Alternatively, manufacturers can randomise passwords per device rather than having a small number of default passwords.

3. **Use of the Manufacturer Usage Description (MUD) specification**. The Manufacturer Usage Description (MUD)[28] specification allows manufacturers to specify authorised and expected user traffic patterns to reduce the threat surface of an IoT device by restricting communications to/from the device to sources and destinations intended by the manufacturer.

4. **Development of a Secure Upgrade Process**. In non-IoT systems, updates are usually delivered via a secure process in which the computer can authenticate the source pushing the patches and feature and configuration updates. IoT manufacturers have, generally, not established such a secure upgrade process, which enables attackers to conduct a man-in-the-middle push of their own malicious updates to the devices. The IoT Firmware Update Architecture [29] provides guidance on implementing a secure firmware update architecture including hard rules defining how device manufacturers should operate.

Additionally, the UK Department for Digital, Culture, Media, and Sport have provided the *Code of Practice for consumer IoT security*[30]. Included in the code of practice are 13 guidelines for improving the security of consumer IoT products and associated services. Two of the guidelines overlap with NIST bullets 2 and 4 above. The full list of guidelines include the following: (1) No default passwords; (2) Implement a vulnerability disclosure policy; (3) Keep software updated; (4) Securely store credentials and security-sensitive data; (5) Communicate securely (i.e. use encryption for sensitive data); (6) Minimise exposed attack surfaces; (7) Ensure software integrity (e.g. use of a secure boot); (8) Ensure that personal data is protected (i.e. in accordance with GDPR); (9) Make systems resilient to outages; (10) Monitor system telemetry data; (11) Make it easy for consumers to delete personal data; (12) Make installation and maintenance of devices easy; and (13) Validate input data. Finally, Microsoft has provided an Internet of Things security architecture[31]

3.5 Road Vehicles

A hacker that compromises a connected road vehicle‘s braking or steering system could cause a passenger or driver to lose their lives. Attacks such as these have been demonstrated, beginning with the takeover of a Ford Escape and a Toyota Prius by white-hat hackers Charlie Miller and Chris Valasek in 2013[32] . Connected commercial vehicles are part of the critical infrastructure in complex global supply chains. In 2018, the number of reported attacks on connected vehicles shot up six times more than the number just three years earlier [44], due to both the increase in connected vehicles and their increased attractiveness as a target of attackers [45]. Broader issues with Cyber-Physical Systems are addressed in another knowledge area.

The US National Highway Traffic Safety Administration (HTSA) defines road vehicle cyber security as the protection of automotive electronic systems, communication networks, control algorithms, software, users and underlying data from malicious attacks, damage, unauthorised access or manipulation[33]. The HTSA provides four guidelines for the automotive industry for consideration in their secure software development lifecycle:

[28] https://tools.ietf.org/id/draft-ietf-opsawg-mud-22.html

[29] https://tools.ietf.org/id/draft-moran-suit-architecture-02.html

[30] https://www.gov.uk/government/publications/code-of-practice-for-consumer-iot-security/code-of-practice-for-consumer-iot-security

[31] https://docs.microsoft.com/en-us/azure/iot-fundamentals/iot-security-architecture

[32] https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[33] https://www.nhtsa.gov/crash-avoidance/automotive-cybersecurityautomotive-cybersecurity-overview

1. The team should follow a secure product development process based on a systems-engineering approach with the goal of designing systems free of unreasonable safety risks including those from potential cyber security threats and vulnerabilities.
2. The automotive industry should have a documented process for responding to incidents, vulnerabilities and exploits. This process should cover impact assessment, containment, recovery and remediation actions, the associated testing, and should include the creation of roles and responsibilities for doing so. The industry should also establish metrics to periodically assess the effectiveness of their response process.
3. The automotive industry should document the details related to their cyber security process, including the results of risk assessment, penetration testing and organisations decisions related to cyber security. Essential documents, such as cyber security requirements, should follow a robust version control protocol.
4. These security requirements should be incorporated into the product's security requirements, as laid out in Section 2.1.1 bullet 2, Section 2.1.2 bullet 6, and Section 2.1.3 bullet 1.:
   (a) Limit developer/debugging access to production devices, such as through an open debugging port or through a serial console.
   (b) Keys (e.g., cryptographic) and passwords which can provide an unauthorised, elevated level of access to vehicle computing platforms should be protected from disclosure. Keys should not provide access to multiple vehicles.
   (c) Diagnostic features should be limited to a specific mode of vehicle operation which accomplishes the intended purpose of the associated feature. For example, a diagnostic operation which may disable a vehicle's individual brakes could be restricted to operating only at low speeds or not disabling all the brakes at the same time.
   (d) Encryption should be considered as a useful tool in preventing the unauthorised recovery and analysis of firmware.
   (e) Limit the ability to modify firmware and/or employ signing techniques to make it more challenging for malware to be installed on vehicles.
   (f) The use of network servers on vehicle ECUs should be limited to essential functionality, and services over these ports should be protected to prevent use by unauthorised parties.
   (g) Logical and physical isolation techniques should be used to separate processors, vehicle networks, and external connections as appropriate to limit and control pathways from external threat vectors to cyber-physical features of vehicles.
   (h) Sending safety signals as messages on common data buses should be avoided, but when used should employ a message authentication scheme to limit the possibility of message spoofing.
   (i) An immutable log of events sufficient to enable forensic analysis should be maintained and periodically scrutinised by qualified maintenance personnel to detect trends of cyber-attack.
   (j) Encryption methods should be employed in any IP-based operational communication between external servers and the vehicle, and should not accept invalid certificates.
   (k) Plan for and design-in features that could allow for changes in network routing rules to be quickly propagated and applied to one, a subset or all vehicles

The International Standards Organization (ISO)[34] and the Society for Automotive Engineering (SAE) International [35] are jointly developing an international Standard, *ISO 21434 Road vehicles - cyber*

[34] https://www.iso.org/standard/70918.html
[35] www.sae.org

security engineering[36]. The standard will specify minimum requirements on security engineering processes and activities, and will define criteria for assessment. Explicitly, the goal is to provide a structured process to ensure cyber security is designed in upfront and integrated throughout the lifecycle process for both hardware and software.

The adoption of a secure software lifecycle in the automotive industry may be driven by legislation, such as through the US SPY Car Act[37] or China and Germany's Intelligent and Connected Vehicles (ICVs) initiative[38].

3.6 ECommerce/Payment Card Industry

The ability to steal large quantities of money makes the Payment Card Industry (PCI) an especially attractive target for attackers. In response, the PCI created the Security Standards Council, a global forum for the ongoing development, enhancement, storage, dissemination, and implementation of security standards for account data protection. The Security Standards Council established the Data Security Standard (PCI DSS), which must be upheld by any organisations that handle payment cards, including debit and credit cards. PCI DSS contains 12 requirements[39] that are a set of security controls that businesses are required to implement to protect credit card data. These specific requirements are incorporated into the product's security requirements, as laid out in Section 2.1.1 bullet 2, Section 2.1.2 bullet 6, and Section 2.1.3 bullet 1. The 12 requirements are as follows:

1. Install and maintain a firewall configuration to protect cardholder data.
2. Do not use vendor-supplied defaults for system passwords and other security parameters.
3. Protect stored cardholder data.
4. Encrypt transmission of cardholder data across open, public networks.
5. Use and regularly update antivirus software.
6. Develop and maintain secure systems and applications, including detecting and mitigating vulnerabilities and applying mitigating controls.
7. Restrict access to cardholder data by business need-to-know.
8. Assign a unique ID to each person with computer access.
9. Restrict physical access to cardholder data.
10. Track and monitor all access to network resources and cardholder data.
11. Regularly test security systems and processes.
12. Maintain a policy that addresses information security.

4 Assessing the Secure Software Lifecycle

[46, 47]

Organisations may wish to or be required to assess the maturity of their secure development lifecycle. Four assessment approaches are described in this section.

[36] https://www.iso.org/standard/70918.html
[37] https://www.congress.gov/bill/115th-congress/senate-bill/680
[38] http://icv.sustainabletransport.org/
[39] https://searchsecurity.techtarget.com/definition/PCI-DSS-12-requirements

4.1 SAMM

The Software Assurance Maturity Model (SAMM)[40] is an open framework to help organisations formulate and implement a strategy for software security that is tailored to the specific risks facing the organisation. Resources are provided for the SAMM to enable an organisation to do the following:

1. Define and measure security-related activities within an organisation.
2. Evaluate their existing software security practices.
3. Build a balanced software security program in well-defined iterations.
4. Demonstrate improvements in a security assurance program.

Because each organisation utilises its own secure software process (i.e., its own unique combination of the practices laid out in Sections 2 and 3), the SAMM provides a framework to describe software security initiatives in a common way. The SAMM designers enumerated activities executed by organisations in support of their software security efforts. Some example activities include: build and maintain abuse case models per project; specify security requirements based upon known risks; and identify the software attack surface. These activities are categorised into one of 12 security practices. The 12 security practices are further grouped into one of four business functions. The business functions and security practices are as follows:

1. Business Function: Governance
   (a) Strategy and metrics
   (b) Policy and compliance
   (c) Education and guidance
2. Business Function: Construction
   (a) Threat assessment
   (b) Security requirements
   (c) Secure architecture
3. Business Function: Verification
   (a) Design review
   (b) Code review
   (c) Security testing
4. Business Function: Deployment
   (a) Vulnerability management
   (b) Environment hardening
   (c) Operational enablement

The SAMM assessments are conducted through self-assessments or by a consultant chosen by the organisation. Spreadsheets are provide by SAMM for scoring the assessment, providing information for the organisation on their current maturity level:

[40] https://www.opensamm.org/ and https://www.owasp.org/images/6/6f/SAMM_Core_V1-5_FINAL.pdf

- 0: Implicit starting point representing the activities in the Practice being unfulfilled.
- 1: Initial understanding and ad hoc provision of the Security Practice.
- 2: Increase efficiency and/or effectiveness of the Security Practice.
- 3: Comprehensive mastery of the Security Practice at scale.

Assessments may be conducted periodically to measure improvements in an organisation's security assurance program.

4.2 BSIMM

Gary McGraw, Sammy Migues, and Brian Chess desired to create a descriptive model of the state-of-the-practice in secure software development lifecycle. As a result, they forked an early version of SAMM (see Section 4.1) to create the original structure of the Building Security In Maturity Model (BSIMM) [46, 47] in 2009. Since that time, the BSIMM has been used to structure a multi-year empirical study of the current state of software security initiatives in industry.

Because each organisation utilises its own secure software process (i.e., its own unique combination of the practices laid out in Sections 2 and 3), the BSIMM provides a framework to describe software security initiatives in a common way. Based upon their observations, the BSIMM designers enumerated 113 activities executed by organisations in support of their software security efforts. Some example activities include: build and publish security features; use automated tools along with a manual review; and integrate black-box security tools into the quality assurance process. Each activity is associated with a maturity level and is categorised into one of 12 practices. The 12 practices are further grouped into one of four domains. The domains and practices are as follows:

1. Domain: Governance
   (a) Strategy and metrics
   (b) Compliance and policy
   (c) Training
2. Domain: Intelligence
   (a) Attack models
   (b) Security features and design
   (c) Standards and requirements
3. Domain: Secure software development lifecycle touchpoints
   (a) Architecture analysis
   (b) Code review
   (c) Security testing
4. Domain: Deployment
   (a) Penetration testing
   (b) Software environment
   (c) Configuration management and vulnerability management

BSIMM assessments are conducted through in-person interviews by software security professionals at Cigital (now Synopsys) with security leaders in a firm. Via the interviews, the firm obtains a scorecard on which of the 113 software security activities the firm uses. After the firm completes the interviews, they are provided information comparing themselves with the other organisations that have been assessed. BSIMM assessments have been conducted since 2008. Annually, the overall results of the assessments from all firms are published, resulting in the BSIMM1 through BSIMM9 reports. Since the BSIMM study began in 2008, 167 firms have participated in BSIMM assessment, sometimes multiple times, comprising 389 distinct measurements. To ensure the continued relevance of the data reported, the BSIMM9 report excluded measurements older than 42 months and reported on 320 distinct measurements collected from 120 firms.

4.3 The Common Criteria

The purpose of this Common Criteria (CC)[41] is to provide a vehicle for international recognition of a secure information technology (IT) *product* (where the SAMM and BSIMM were assessments of a development process). The objective of the CC is for IT products that have earned a CC certificate from an authorised Certification/Validation Body (CB) to be procured or used with no need for further evaluation. The Common Criteria seek to provide grounds for confidence in the reliability of the judgments on which the original certificate was based by requiring that a CB issuing Common Criteria certificates should meet high and consistent standards. A developer of a new product range may provide guidelines for the secure development and configuration of that product. This guideline can be submitted as a Protection Profile (the pattern for similar products that follow on). Any other developer can add to or change this guideline. Products that earn certification in this product range use the protection profile as the delta against which they build.

Based upon the assessment of the CB, a product receives an Evaluation Assurance Level (EAL). A product or system must meet specific assurance requirements to achieve a particular EAL. Requirements involve design documentation, analysis and functional or penetration testing. The highest level provides the highest guarantee that the system's principal security features are reliably applied. The EAL indicates to what extent the product or system was tested:

- **EAL 1: Functionally tested**. Applies when security threats are not viewed as serious. The evaluation provides evidence that the system functions in a manner consistent with its documentation and that it provides useful protection against identified threats.
- **EAL 2: Structurally tested**. Applies when stakeholders require low-to-moderate independently-assured security but the complete development record is not readily available, such as with securing a legacy system.
- **EAL 3: Methodically tested and checked**. Applies when stakeholders require a moderate level of independently-assured security and a thorough investigation of the system and its development, without substantial re-engineering.
- **EAL 4: Methodically designed, tested and reviewed**. Applies when stakeholders require moderate-to-high independently-assured security in commodity products and are prepared to incur additional security-specific engineering costs.
- **EAL 5: Semi-formally designed and tested**. Applies when stakeholders require high, independently-assured security in a planned development and require a rigorous development approach that does not incur unreasonable costs from specialist security engineering techniques.
- **EAL 6: Semi-formally verified design and tested**. Applies when developing systems in high-risk situations where the value of the protected assets justifies additional costs.

[41] https://www.commoncriteriaportal.org/ccra/index.cfm

- **EAL 7: Formally verified design and tested**. Applies when developing systems in extremely high-risk situations and when the high value of the assets justifies the higher costs.

The CC provides a set of security functional and security assurance requirements. These requirements, as appropriate, are incorporated into the product's security requirements, as laid out in Section 2.1.1 bullet 2, Section 2.1.2 bullet 6, and Section 2.1.3 bullet 1.

5 Adopting a Secure Software Lifecycle

[46, 47, 48]

This knowledge area has provided a myriad of possible practices an organisation can include in its secure software lifecycle. Some of these practices, such as those discussed in Section 2, potentially apply to any product. Other practices are domain specific, such as those discussed in Section 3.

Organisations adopting new practices often like to learn from and adopt practices that are used by organisations similar to themselves [48]. When choosing which security practices to include in a secure software lifecycle, organisations can consider looking at the latest BSIMM [46, 47] results which provide updated information on the adoption of practices in the industry.

DISCUSSION

[49]

This chapter has provided an overview of of three prominent and prescriptive secure software lifecycle processes and six adaptations of these processes that can be applied in a specified domain. However, the cybersecurity landscape in terms of threats, vulnerabilities, tools, and practices is ever evolving. For example, a practice has has not be been mentioned in any of these nine processes is the use of a bug bounty program for the identification and resolution of vulnerabilities. With a bug bounty program, organisations compensate individuals and/or researchers for finding and reporting vulnerabilities. These individuals are external to the organisation producing the software and may work independently or through a bug bounty organisation, such as HackerOne[42].

While the majority of this knowledge area focuses on technical practices, the successful adoption of these practices involves organisational and cultural changes in an organisation. The organisation, starting from executive leadership, must support the extra training, resources, and steps needed to use a secure development lifecycle. Additionally, every developer must uphold his or her responsibility to take part in such a process.

A team and an organisation need to choose the appropriate software security practices to develop a customised secure software lifecycle based upon team and technology characteristics and upon the security risk of the product.

While this chapter has provided practices for developing secure products, information insecurity is often due to economic disincentives [49] which drives software organizations to choose the rapid deployment and release of functionality over the production of secure products. As a result, increasingly governments and industry groups are imposing cyber security standards on organisations as a matter of legal compliance or as a condition for being considered as a vendor. Compliance requirements may lead to faster adoption of a secure development lifecycle. However, this compliance-driven adoption may divert efforts away from the real security issues by driving an over-focus on compliance requirements rather than on the pragmatic prevention and detection of the most risky security concerns.

	HowardSDL [2]	ViegaBSS [5]	HowardWSC [8]	SAFECodeFundamental [33]
1 Motivation	c1	c1	c1	
2 Prescriptive Secure Software Lifecycle Processes				
2.1 Secure Software Lifecycle Processes	c2	c2	c2	c2
2.2 Comparing the Secure Software Lifecycle Models				
3 Adaptations of the Secure Software Lifecycle				
3.1 Agile Software Development and DevOps				c3
3.2 Mobile				
3.3 Cloud Computing				
3.4 Internet of Things (IoT)				
3.5 Road Vehicles				
3.6 ECommerce/Payment Card Industry				
4 Assessing the Secure Software Lifecycle				
5 Adopting a Secure Software Lifecycle				

CROSS-REFERENCE OF TOPICS VS REFERENCE MATERIAL

FURTHER READING

Building Secure Software: How to Avoid Security Problems the Right Way [5]

This book introduces the term software security as an engineering discipline for building security into a product. This book provides essential lessons and expert techniques for security professionals who understand the role of software in security problems and for software developers who want to build secure code. The book also discusses risk assessment, developing security tests, and plugging security holes before software is shipped.

Writing Secure Code, Second Edition. [8]

The first edition of this book was internally published in Microsoft and was required reading for all members of the Windows team during the Windows Security Push. The second edition was made publicly available in the 2003 book and provides secure coding techniques to prevent vulnerabilities, to detect design flaws and implementation bugs, and to improve test code and documentation.

Software Security: Building Security In [9]

This book discusses seven software securing best practices, called touchpoints. It also provides information on software security fundamentals and contexts for a software security program in an enterprise.

[42] https://www.hackerone.com

The Security Development Lifecycle (Original Book) [2]

This seminal book provides the foundation for the other processes laid out in this knowledge area, and was customised over the years by other organisations, such as Cisco [43]. The book lays out 13 stages for integrating practices into a software development lifecycle such that the product is more secure. This book is out of print, but is avaialble as a free download[44].

The Security Development Lifecycle (Current Microsoft Resources) [10]

The Microsoft SDL are practices that are used internally to build secure products and services, and address security compliance requirements by introducing security practices throughout every phase of the development process. This webpage is a continuously-updated version of the seminal book [2] based on Microsoft's growing experience with new scenarios such as the cloud, the Internet of Things (IoT) and artificial intelligence (AI).

Software Security Engineering: A Guide for Project Managers [25]

This book is a management guide for selecting from among sound software development practices that have been shown to increase the security and dependability of a software product, both during development and subsequently during its operation. Additionally, this book discusses governance and the need for a dynamic risk management approach for identifying priorities throughout the product lifecycle.

Cyber Security Engineering: A Practical Approach for Systems and Software Assurance [50]

This book provides a tutorial on the best practices for building software systems that exhibit superior operational security, and for considering security throughout your full system development and acquisition lifecycles. This book provides seven core principles of software assurance, and shows how to apply them coherently and systematically. This book addresses important topics, including the use of standards, engineering security requirements for acquiring COTS software, applying DevOps, analysing malware to anticipate future vulnerabilities, and planning ongoing improvements.

SAFECode's Fundamental Practices for Secure Software Development: Essential Elements of a Secure Development Lifecycle Program, Third Edition [33]

Eight practices for secure development are provided based upon the experiences of member companies of the SAFECode organisation.

OWASP's Secure Software Development Lifecycle Project (S-SDLC) [11]

Based upon a committee of industry participants, the Secure Software Development Lifecycle Project (S-SDLC) defines a standard Secure Software Development Life Cycle and provides resources to help developers know what should be considered or best practices at each phase of a development lifecycle (e.g., Design Phase/Coding Phase/Maintain Phase/etc.) The committee of industry participants are members of the Open Web Application Security Project (OWASP)[45], an international not-for-profit organisation focused on improving the security of web application software. The earliest secure software lifecycle contributions from OWASP were referred to as the Comprehensive, Lightweight Application Security Process (CLASP).

[43] https://www.cisco.com/c/en/us/about/trust-center/technology-built-in-security.html stickynav=2
[44] https://blogs.msdn.microsoft.com/microsoft_press/2016/04/19/free-ebook-the-security-development-lifecycle/
[45] https://www.owasp.org/

Security controls

Government and standards organizations have provided security controls to be integrated in a secure software or systems lifecyle:

1. The Trustworthy Software Foundation [46] provides the the Trustworthy Software Framework (TSFr) [47] a collection of good practice, existing guidance and relevant standards across the five main facets of trustworthiness: Safety; Reliability; Availability; Resilience; and Security. The purpose of the TSFr is to provide a minimum set of controls such that, when applied, all software (irrespective of implementation constraints) can be specified, realised and used in a trustworthy manner.

2. The US National Institute of Standards and Technology (NIST) has authored the Systems Security Engineering Cyber Resiliency Considerations for the Engineering [51] framework (NIST SP 800-160). This Framework provides resources on cybersecurity knowledge, skills, and abilities (KSAs), and tasks for a number of work roles for achieving the identified cyber resiliency outcomes based on a systems engineering perspective on system life cycle processes.

3. The Software Engineering Institute (SEI) has collaborated with professional organisations, industry partners and institutions of higher learning to develop freely-available curricula and educational materials. Included in these materials are resources for a software assurance program[48] to train professionals to build security and correct functionality into software and systems.

4. The United Kingdom (UK) National Cyber Security Centre[49] provide resources for secure software development:

   (a) Application development[50]: recommendations for the secure development, procurement, and deployment of generic and platform-specific applications.

   (b) Secure development and deployment guidance[51]: more recommendations for the secure development, procurement, and deployment of generic and platform-specific applications.

   (c) The leaky pipe of secure coding[52]: a discussion of how security can be woven more seamlessly into the development process, particularly by developers who are not security experts.

Training materials

Training materials are freely-available on the Internet. Some sites include the following:

1. The Trustworthy Software Foundation provides a resource library [53] of awareness materials and guidance targeted for those who teach trustworthy software principles, those who seek to learn about Trustworthy Software and those who want to ensure that the software they use is trustworthy. The resources available include a mixture of documents, videos, animations and case studies.

[46] https://tsfdn.org
[47] https://tsfdn.org/ts-framework/
[48] https://www.sei.cmu.edu/education-outreach/curricula/software-assurance/index.cfm
[49] https://www.ncsc.gov.uk/
[50] https://www.ncsc.gov.uk/collection/application-development
[51] https://www.ncsc.gov.uk/collection/developers-collection
[52] o https://www.ncsc.gov.uk/blog-post/leaky-pipe-secure-coding
[53] https://tsfdn.org/resource-library/

2. The US National Institute of Standards and Technology (NIST) has created the NICE Cyber security Workforce Framework [52]. This Framework provides resources on cyber security Knowledge, Skills, and Abilities (KSAs), and tasks for a number of work roles.

3. The Software Engineering Institute (SEI) has collaborated with professional organisations, industry partners and institutions of higher learning to develop freely-available curricula and educational materials. Included in these materials are resources for a software assurance program[54] to train professionals to build security and correct functionality into software and systems.

4. SAFECode offers free software security training courses delivered via on-demand webcasts[55].

REFERENCES

[1] "Bill gates: Trustworthy computing," https://www.wired.com/2002/01/bill-gates-trustworthy-computing/, January 17, 2002.

[2] M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA, USA: Microsoft Press, 2006.

[3] Poneman Institute, "2018 cost of a data breach study: Global overview." [Online]. Available: https://securityintelligence.com/series/ponemon-institute-cost-of-a-data-breach-2018/

[4] G. McGraw, "Testing for security during development: why we should scrap penetrate-and-patch," *IEEE Aerospace and Electronic Systems Magazine*, vol. 13, no. 4, pp. 13–15, April 1998.

[5] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way (Addison-Wesley Professional Computing Series)*, 1st ed. Addison-Wesley Professional, 2002.

[6] T. Greene, "That heartbleed problem may be more pervasive than you think." [Online]. Available: https://www.networkworld.com/article/3162232/security/that-hearbleed-problem-may-be-more-pervasive-than-you-think.html

[7] eWeek editors, "Microsoft trustworthy computing timeline." [Online]. Available: https://www.eweek.com/security/microsoft-trustworthy-computing-timeline

[8] M. Howard and D. E. Leblanc, *Writing Secure Code*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2003.

[9] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[10] Microsoft, "The security development lifecycle," https://www.microsoft.com/en-us/securityengineering/sdl/, 2019.

[11] "Owasp secure software development lifecycle project," https://www.owasp.org/index.php/OWASP_Secure_Software_Development_Lifecycle_Project, 2018.

[12] P. Morrison, D. Moye, R. Pandita, and L. Williams, "Mapping the field of software life cycle security metrics," *Information and Software Technology*, vol. 102, pp. 146 – 159, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095058491830096X

[13] A. Shostack, *Threat Modeling: Designing for Security*, 1st ed. Wiley Publishing, 2014.

[14] M. Howard, "Fending off future attacks by reducing attack surface," *MSDN Magazine*, February 4, 2003. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms972812.aspx

[15] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," 1975. [Online]. Available: https://www.acsac.org/secshelf/papers/protection_information.pdf

[16] IEEE Center for Secure Design, "Avoiding the top 10 software security design flaws." [Online]. Available: https://cybersecurity.ieee.org/blog/2015/11/13/avoiding-the-top-10-security-flaws

[17] Synopsys Center for Open Source Reearch and Innovation, "2018 open source se-

[54] https://www.sei.cmu.edu/education-outreach/curricula/software-assurance/index.cfm

[55] https://safecode.org/training/

curity and risk analysis," 2018. [Online]. Available: https://www.blackducksoftware.com/open-source-security-risk-analysis-2018

[18] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Information and Software Technology*, vol. 55, no. 7, pp. 1279 – 1288, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584912002339

[19] P. Hope, G. McGraw, and A. I. Anton, "Misuse and abuse cases: getting past the positive," *IEEE Security and Privacy*, vol. 2, no. 3, pp. 90–92, May 2004.

[20] G. Sindre and A. L. Opdahl, "Eliciting security requirements by misuse cases," in *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000*, Nov 2000, pp. 120–131.

[21] K. Tuma, G. Calikli, and R. Scandariato, "Threat analysis of software systems: A systematic literature review," *Journal of Systems and Software*, vol. 144, 06 2018.

[22] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 371–386, May 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.60

[23] C. Theisen, N. Munaiah, M. Al-Zyoud, J. C. Carver, A. Meneely, and L. Williams, "Attack surface definitions: A systematic literature review," *Information and Software Technology*, vol. 104, pp. 94 – 103, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584918301514

[24] N. M. Mohammed, M. Niazi, M. Alshayeb, and S. Mahmood, "Exploring software security approaches in software development lifecycle: A systematic mapping study," *Comput. Stand. Interfaces*, vol. 50, no. C, pp. 107–115, Feb. 2017. [Online]. Available: https://doi.org/10.1016/j.csi.2016.10.001

[25] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, and N. R. Mead, *Software Security Engineering: A Guide for Project Managers (The SEI Series in Software Engineering)*, 1st ed. Addison-Wesley Professional, 2008.

[26] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 148–157. [Online]. Available: http://dl.acm.org.prox.lib.ncsu.edu/citation.cfm?id=998675.999421

[27] G. Elahi and E. Yu, "A goal oriented approach for modeling and analyzing security trade-offs," in *Proceedings of the 26th International Conference on Conceptual Modeling*, ser. ER'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 375–390. [Online]. Available: http://dl.acm.org.prox.lib.ncsu.edu/citation.cfm?id=1784489.1784524

[28] V. Saini, Q. Duan, and V. Paruchuri, "Threat modeling using attack trees," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 124–131, Apr. 2008. [Online]. Available: http://dl.acm.org.prox.lib.ncsu.edu/citation.cfm?id=1352079.1352100

[29] L. Williams, A. Meneely, and G. Shipley, "Protection poker: The new software security "game";," *IEEE Security Privacy*, vol. 8, no. 3, pp. 14–20, May 2010.

[30] G. McGraw, "The new killer app for security: Software inventory," *Computer*, vol. 51, no. 2, pp. 60–62, February 2018.

[31] R. Kuhn, M. Raunak, and R. Kacker, "What proportion of vulnerabilities can be attributed to ordinary coding errors?: Poster," in *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, ser. HoTSoS '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:1. [Online]. Available: http://doi.acm.org/10.1145/3190619.3191686

[32] US National Institute of Standards and Technology (NIST), "Guide for conducting risk assessments, special publication 800-30," 2002. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final

[33] SAFECode, "Fundamental practices for secure software development: Essential elements of a secure development lifecycle program, third edition." [Online]. Available: https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_

for_Secure_Software_Development_March_2018.pdf
[34] US National Institute of Standards and Technology (NIST), "Security and privacy controls for federal information systems and organizations; nist special publication 800-53, revision 4," April 2013. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-53r4
[35] F. O. for Information Security, "Guidelines for developer documentation according to common criteria version 3.1," 2007. [Online]. Available: https://www.commoncriteriaportal.org/files/ccfiles/CommonCriteriaDevelopersGuide_1_0.pdf
[36] B. D. Win, R. Scandariato, K. Buyens, J. Grégoire, and W. Joosen, "On the secure software development process: Clasp, sdl and touchpoints compared," *Information and Software Technology*, vol. 51, no. 7, pp. 1152 – 1171, 2009, detailed data analysis of practices available online. [Online]. Available: lirias.kuleuven.be/1655460
[37] SAFECode, "Practical security stories and security tasks for agile development environments." [Online]. Available: http://safecode.org/wp-content/uploads/2018/01/SAFECode_Agile_Dev_Security0712.pdf
[38] Microsoft, "Secure devops," https://www.microsoft.com/en-us/securityengineering/devsecops, 2019.
[39] "Owasp mobile security project," https://www.owasp.org/index.php/OWASP_Mobile_Security_Project, 2017.
[40] OWASP, "Owasp mobile security project - mobile threat model," 2013. [Online]. Available: https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Mobile_Threat_Model
[41] SAFECode, "Practices for secure development of cloud applications." [Online]. Available: https://safecode.org/publication/SAFECode_CSA_Cloud_Final1213.pdf
[42] J. Voas, R. Kuhn, P. Laplante, and S. Applebaum, "Inernet of things (iot) trust concerns, draft," 2018. [Online]. Available: https://csrc.nist.gov/publications/detail/white-paper/2018/10/17/iot-trust-concerns/draft
[43] ENISA, "Benefits, risks and recommendations for information security," 2009. [Online]. Available: https://www.enisa.europa.eu/publications/cloud-computing-risk-assessment/at_download/fullReport
[44] Yossi Vardi, "Where automotive cybersecurity is headed in 2019." [Online]. Available: https://thenextweb.com/contributors/2019/02/10/where-automotive-cybersecurity-is-headed-in-2019/
[45] G. McGraw, "From mainframes to connected cars: How software drives the automotive industry," *Security Ledger*, vol. August 15, 2018.
[46] G. McGraw, S. Migues, and J. West, "Building security in maturity model," https://www.bsimm.com/, 2009.
[47] L. Williams, G. McGraw, and S. Migues, "Engineering security vulnerability prevention, detection, and response," *IEEE Software*, vol. 35, no. 5, pp. 76–80, Sep. 2018.
[48] G. A. Moore, *Crossing the Chasm: Marketing and Selling Disruptive Products to Mainstream Customers*. Harper Collins, 2002.
[49] R. Anderson, "Why information security is hard - an economic perspective," in *Seventeenth Annual Computer Security Applications Conference*, Dec 2001, pp. 358–365.
[50] N. R. Mead and C. Woody, *Cyber Security Engineering: A Practical Approach for Systems and Software Assurance*, 1st ed. Addison-Wesley Professional, 2016.
[51] R. Ross, R. Graubart, D. Bodeau, and R. McQuaid, "Nist systems security engineering 800-160," 2018. [Online]. Available: https://csrc.nist.gov/CSRC/media/Publications/sp/800-160/vol-2/draft/documents/sp800-160-vol2-draft.pdf
[52] W. Newhouse, S. Keith, B. Scribner, and G. Witte, "Nice cybersecurity workforce framework, special publication 800-181," 2017. [Online]. Available: https://www.nist.gov/itl/applied-cybersecurity/nice/resources/nice-cybersecurity-workforce-framework