ESSENTIAL ACTIVITIES FOR SECURE SOFTWARE DEVELOPMENT

Mamdouh Alenezi[1] and Sadiq Almuairfi[2]

[1]College of Computer and Information Science,
Prince Sultan University, Riyadh, Saudi Arabia
[2]Information Technology Centre, Prince Sultan University, Riyadh, Saudi Arabia

ABSTRACT

Diverse types of software are used in almost all sectors of businesses in the modern world. They provide mechanisms that enable buyers and sellers to interact virtually, reduce manual work in businesses and institutions as well as make work a lot easier. Increased demand for software has led to the increased investment that has subsequently attracted numerous security attacks. Millions of resources are held in various software worldwide, cyber-attack criminals have made a career in breaching software security for selfish gains, thus necessitating the development and establishment of secure software. Through a literature review, the work introduces concepts and terms used in secure software development, presents the best practices and provides a review of the models that could be used. Confidentiality, integrity, availability, and non-repudiation are secure software terms that mean it should be secret, safe, and accessible and keeps a record of every activity undertaken. The proposed work advocates for several best practices among them the creation of a secure perimeter that limits access to key segments or parts of the system in addition to reducing attacking surface or rather reducing the opportunities available for cyber-attack. In regard to the engineering of software, the paper recommends that system requirements must be established before the software is created. Additional engineering ought to be done after the system has been evaluated just before the official launch. Moreover, the paper recommends the adoption of strategies that are used by renowned software models such as Microsoft Software Development Life-cycle among others. Those models have put secure software strategies throughout the life-cycle of software development. They recognize the need to put secure engineering systems during the design and utilization of the software because new methods of breaching software security come up every new day. The paper concludes by noting that continued collaborative efforts to guarantee more secure software is still a demanding need. Adherence to basic secure software development and utilization is essential in addition to developing additional engineering that maintains the integrity, confidentially and accessibility of the software.

KEYWORDS

Software Engineering, Software Quality, Software Security Development.

1. INTRODUCTION

Our society is buzzing with activities conducted either with the help of a software or within the software. All manner of businesses is in one way or another using software in a manner that has left human beings addicted to software use [6]. The concern is the safety of transactions, personal information and valuable possessions held within those software systems. Hence the need for human beings to keep evaluating the safety of software systems. However, software systems security can be quite a broad topic to figure out. But the news of cyber-attacks lost money and service interruptions in government and non-government institutions are common. The rampant nature of successful software attacks has demoralized many of those who hope for more secure systems.

Despite its complex nature, the security of software systems can be enhanced by adhering to basic principles that minimize the risks [6, 14]. Furthermore, stakeholders have to acknowledge that software security is not a destination, but a continuous journey that requires attentiveness and rapid actions that deter if not prevent attacks. It also implies that risks cannot be eliminated completely, but can be reduced to minimal levels without causing damage to people and institutions using a particular software.

Measures to reduce risk ought to be based on the understanding of the possible avenues that could be used in the attack [5]. Risk calculation is essential because it informs how or when a particular attack could be carried out [3]. It also enables the acquisition of knowledge and skills to prevent and mitigate those attacks through measures that are timely and proportionate to the amount of threat anticipated. Various terminology is used with regard to software security [18]. Confidentiality entails having the system being able to hold information secret, integrity entails having the system being able to guarantee the safety of information and availability entails the system being able to stay available and accessible to its legitimate users. Secure software systems ought to fulfill integrity, accessibility, and confidentiality requirements [18].

On top of that, the system must keep records and logs of all different kinds of activities that have happened through it. It should be able to trace the activities done through it and the people who did them, which is a feature commonly referred to as non-repudiation. Software security is a concept that means providing engineering measures to ensure that a software retains integrity, confidentiality, accessibility and non-repudiation characteristics to guarantee functionality in the midst of numerous attacks that are witnessed in the modern-day. Figure 1 shows the general model of the Secure Software Development Life Cycle (SSDLC) and security components with correspondence to phases.

2. Best Practices for Secure Software Development

Software security is endowed with several concepts [13]. It is important that people using and developing software systems are aware of the basic concepts and principles that promote integrity, accessibility, the confidentiality of their software systems. Some of the concepts such as security perimeter and attack surfaces [10] among others are presented in the following discussion. The security perimeter is not a new terminology but a common practice. Buildings are secured using a security perimeter wall as the first line of defense against external threats. The concept is replicated in all key installations such as airports, military barracks, and schools among others as the priority measure in creating a secured zone within the perimeter wall. A single entry is created such that all those who access the secured zone are adequately verified, their intention is known and the integrity of the secured zone is not interfered with. In software security development, it is paramount to establish a security perimeter to protect key sections of the software against authorized and malicious access. However, the ever-changing nature of the business, global networking and increased openness to stakeholders and customers have made it challenging to clearly define the location of the security perimeter.

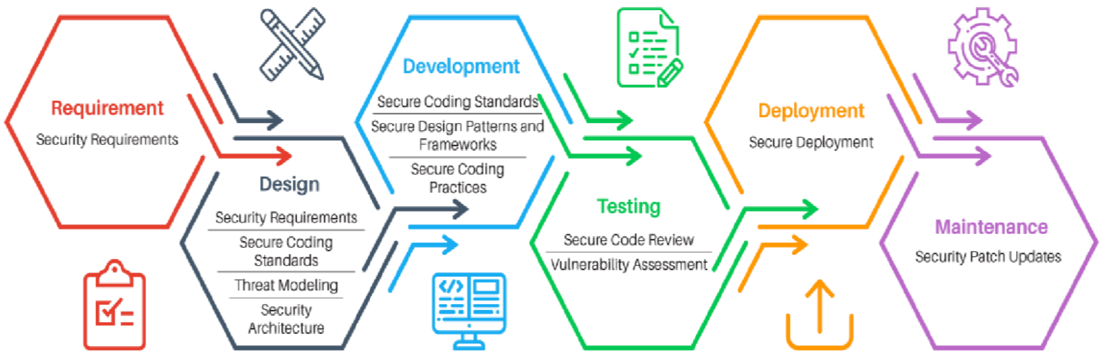
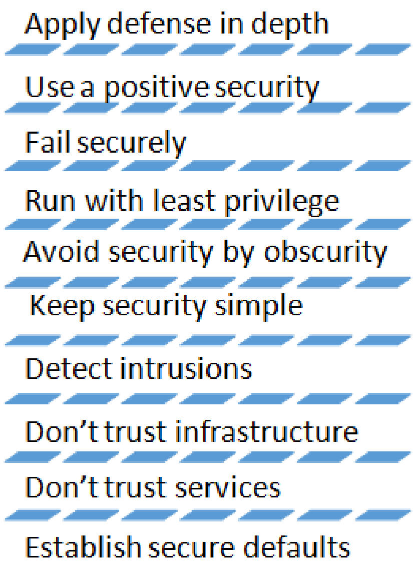


Figure 1. General Secure Software Development Life Cycle Phases Model

The attack surface is another concept that might be applied in order to enhance software systems security. It is largely applied in the assessment of risk and establishment of measures aimed at not only eliminating the risk but also mitigating the effect arising from a successful attack. The attack surface defines all feasible entry points, which a malicious person might utilize to access the security zone of a software. It means the total areas of possible attack irrespective of whether the attacker is working from the external environment or from the internal environment. The concept recommends a reduction of the attack surface to the minimum possible because the higher the number of entry points, the more difficult it is to guarantee the integrity of the software system. Besides, secure software development requires more than design but also the application of ethical behaviors in the development, installation, and utilization of software systems. The principles of managing software are helpful in that they help make decisions that promote the security of the system. With the application of secure software system principles, system requirements are established well, the design is made with focus to reducing the attack surface and the system is implemented in a manner that takes note of new developments that require regular system updates.

The Open Web Application Security Project (OWASP) [1] discussed 10 principles as listed in Figure 2:


Apply defense in depth

Use a positive security

Fail securely

Run with least privilege

Avoid security by obscurity

Keep security simple

Detect intrusions

Don’t trust infrastructure

Don’t trust services

Establish secure defaults

Figure 2. Software Security Principles and Best Practices

2.1. Practice 1 Apply Defence in Depth

This principle states that you can increase security by structuring your security controls as a sequence of overlapping layers to deliver the needed three essentials to secure assets (prevention, detection, and response). In software development, countermeasures and security controls should be layered to protect, detect, and respond to possible security attacks. Defence-in-depth guides the selection of controls for an application to ensure its resilience against different types of attacks and reduces the likelihood of a single point of failure.

2.2. Practice 2 Use a Positive Security Model

Using a positive security model-also called whitelisting-determines what can be allowed and rejects everything else. The other available model is a" negative" (or" blacklist") security model, which determines what can be disallowed and allowing everything else. One of the issues in software development is the need to" enumerate badness" or begin using a blacklist which is an infinite process. Whitelisting focuses on" enumerating goodness," which is an easier and more efficient task. Developers can focus only on acceptable formats and reject anything else which will lead to the prevention of new attacks-including zero-day attacks-that have not been anticipated by developers.

2.3. Practice 3 Fail Securely

One of the key aspects of secure and resilient applications is handling errors and exceptions securely. Two main types of errors need special care namely exceptions that happen while processing a security control and code exceptions that are not" security-relevant". The most important thing is that these errors and exceptions do not allow behavior that security countermeasures would normally not allow. Developers should know that there are always three possible outcomes from any security countermeasure: allow, disallow the operation, or exception. In case of a failure in a security mechanism, the execution path should always go to disallowing the operation by default. The other type of security-relevant exception is security-relevant if they affect whether the application properly invokes the security control or not. An exception might cause a security method not to be invoked when it should, or it might affect the initialization of variables used in the security control.

2.4. Practice 4 Run with Least Privilege

This principle states that users should always have the minimum privileges that will allow them to their basic business needs. This principle is commonly accepted as an important design decision that enhances the protection of data and functionality from faults and malicious behavior. The principle of least privilege has sometimes a different name which is the principle of least authority (POLA).

2.5. Practice 5 Avoid Security by Obscurity

Security by obscurity is trying to achieve the security of a software system by relying on the difficulty in finding or understanding the security methods, mechanisms, and controls. Relying on the secrecy of the implementation of a system or controls is considered to be a weak security control that will always fail. The psychological effect of keeping the secrecy of the implemented algorithm will lead to not doing a good job in implementing them. Relying on secrecy only may lead to theoretical or actual security vulnerabilities. Security should be achieved by security by design, not by secrecy.

2.6. Practice 6 Keep Security Simple

Keeping security simple is applying simple things rather than complicating things. Be it architecture, code, security control or any artifact should be relatively straightforward and simple for someone to read and understand it. Developers should always strive to achieve simplicity and avoid complexity. Keeping security simple is related to a number of other resilience principles, and using it as a principle or guideline will help you to meet the spirit of several of the other principles. One of the ways to keep security simple is to break security functions down into these discrete objectives:

- Keep services running and information away from attackers-related to denying access by default.
- Allow the right users access to the right information related to the least privilege.
- Defend every layer as if it were the last layer of defense-related to defense in depth.
- Keep a record of all attempts to access information (logging).
- Compartmentalize and isolate resources.

2.7. Practice 7 Detect Intrusions

Usually, software engineers can discover an issue with software by studying the log entries that you cant discover at runtime. Detecting intrusions in software needs at least three main elements: the capability to log security-relevant events, procedures to ensure that logs are monitored regularly, and procedures to respond properly to an intrusion once it has been detected. Particularly, any use of security controls should be logged, with enough detailed information to aid in tracking down an attacker. Being not able to look through the event logs to conclude which events are actionable, then logging events have no added value. Logging can be seen as a forensic function for your software. Detecting intrusions limit the time for an attacker to perfect their attacks. If you detect intrusions perfectly, then an attacker will get only one attempt before he is detected and prevented from launching more attacks.

2.8. Practice 8 Do Not Trust Infrastructure

You will never know exactly what hardware or operating environment your applications will run on. Relying on a security process or function that may or may not be present is a sure way to have security problems. Make sure that your application security requirements are explicitly provided through application code or through the explicit invocation of reusable security functions provided to application developers to use for the enterprise (e.g., OWASP Enterprise Security API).

2.9. Practice 9 Do Not Trust Services

Services can refer to any external system. Many businesses utilize third-party processing powers which by default enforce different security policies that are not under these businesses. Therefore, the implicit trust of externally run systems is not warranted. All external systems should be treated in a similar fashion.

2.10. Practice 10 Establish Secure Defaults

Every application should be delivered securely by default out of the box! You should leave it up to users to decide if they can reduce their security if your application allows it. Secure by default means that the default configuration settings are the most secure settings possible-not necessarily the most user- friendly.

3. SECURITY REQUIREMENTS ENGINEERING

A software system is expected to have key properties such as doing what is it is meant to do, defining who interacts with it and it should be highly domain-specific among other requirements. It is evident that a system requirement can be categorized as a non-functional and/or functional. Security requirements differentiate a system that is functional and non-functional. A secure system ought to provide services in the midst of malicious attacks without losing its integrity and functionality. Therefore, the design of software should be based on security requirements that have been established.

The process of analyzing system requirements is important too. Stakeholders should be involved in the process of identifying system requirements. Adequate time is important too, to ensure analysis is not done hurriedly with errors. Moreover, the analysis should be done in the entire system and must be documented in statements that provide an adequate and accurate description of the system. A mnemonic SMART (Specific, Measurable, Attainable, Realistic, Traceable) [16] is a simple approach to ensure those requirements are documented adequately. In addition to that, the analysis should provide use and misuse/abuse cases [17] situations that may arise to ensure the design takes into consideration all avenues of a possible attack. There should be a simulation of how the system can respond during a use case as well as during a misuse case. In summary, the process of security requirements engineering can be done in a stepwise manner. The first step is having prior knowledge of attack surfaces in other systems similar to the one in use. Thereafter, establish attacker awareness of the possible entry points and simulate how the attacker would leverage on the weaknesses of the system. A similar simulation should be done on the use case so that system requirement engineering can fill all the gaps. In order to elicit security requirements efficiently, here are some recommended steps to follow:

- Start with doing your homework by studying common security issues for systems that are similar to your system, and see if an attacker might exploit such issues in the system being developed. After that, try to describe how the attacker would leverage the problem if it exists.
- Brainstorm on the basis of a list of system resources. For each resource, attempt to construct misuse cases in connection with each of the basic security services: authentication, confidentiality, access control, integrity, and availability.
- Brainstorm on the basis of a set of existing use cases. This may be useful for identifying representative risks and for ensuring that the rest two approaches did not overlook any obvious threats. Misuse cases derived in this fashion are often written in terms of valid use and then annotated to have malicious steps.

Security requirements should go through different phases, from elicitation to analysis and modeling and then prioritization and documentation. Security experts, business analysts, designers, architects, developers, testers, maintenance teams should all be involved in collecting and analyzing security requirements. The challenges posed by the process of evaluating system requirements include costs, emerging threats, and failure to understand threats during the design process. Lack of legal and adequate political framework governing software systems development and use is part of the weaknesses that are influencing security software engineering. Various comprehensive approaches have been recommended in software security requirements engineering namely STORE [7] which stands for Security Threat Oriented Requirements Engineering Methodology, SQUARE which stands for Security QUAlity Requirements Engineering [20] and MSRE (Metamodel of Security Requirements Engineering) [24].

4. SECURITY ARCHITECTURE AND DESIGN

In software development, architecture is considered to be design but not all design is considered to be an architecture [15]. Architecture indicates the important design decisions that form a system, where important is measured by the cost of change [4]. Software architecture and design should clear ambiguities and answer important questions. The architecture and design phase is concerned with answering the question" how". Clear ideas are translated and transformed into reality. Looking at the architecture and design from a functional perspective, this transformation from idea to actual form is very essential to the overall quality and success of the delivered software. In developing secure software systems, architecture and design are considered the most critical phase of the SDLC. Good decisions made during this phase will not only yield an approach and structure that are more resilient and resistant to attack but will often also help to prescribe and guide good decisions in later phases such as coding and testing. Bad decisions made during this phase can lead to design flaws that can never be overcome or resolved by even the most intelligent and disciplined code and test efforts [8].

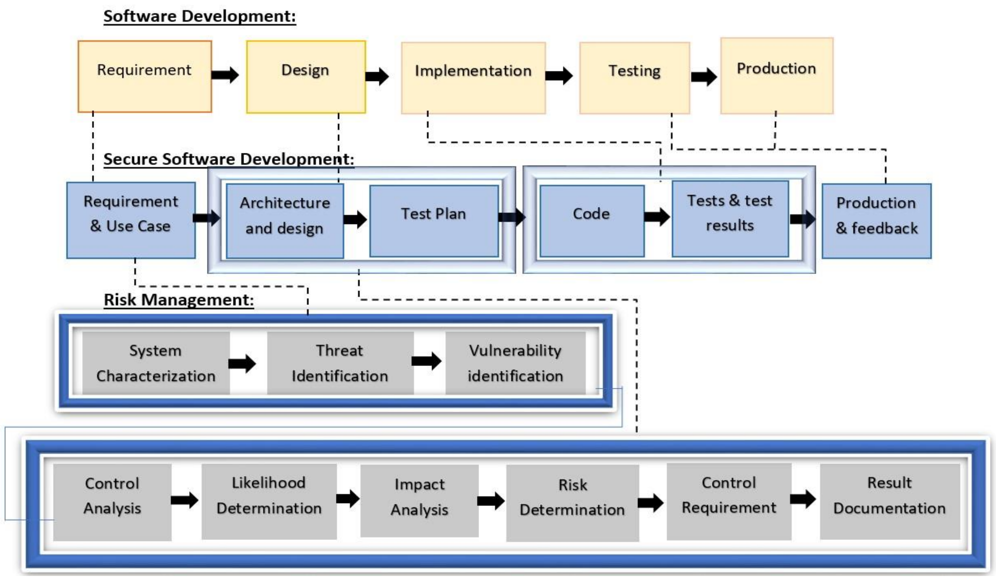


Figure 3. Risk Management in Secure SDLC

Security-specific objectives of software architecture and design are comprehensive functional security architecture (security features and capabilities are fully enabled), attack resistance (contains minimal security weaknesses that could be exploited), attack tolerance (while resisting attack, software function, and capability are not unduly affected), attack resilience (in the face of successful attack, the effects on the software are minimized).

There are different techniques and activities that can be done at this phase. These are Threat and Risk Modelling, design to meet security requirements, security design patterns, design guidelines for secure software development, security design and architecture checklist, and security design and architecture review. See Figure 3 for more information in Risk Management in Secure SDLC.

Threat modeling [26] comprises defining the software attack surface by exploring its functionality for trust boundaries, entry points, data flow, and exit points. Threat models are usually done when functional requirements are complete. Threat modeling ensures that the design complements the security objectives making trade-off and prioritization-of-effort decisions, and reducing the risk of security issues during development and operations. One of the most effective activities done at this stage is threat modeling. Risk modeling discusses risks, their rank, and mitigations. Ranking threats are usually achieved by looking at the organization's business objectives, compliance and regulatory requirements, and security exposures. A threat model is considered to be a specification-just like a functional specification-that defines the architecture needed to implement the functional specification and a test that defines how you plan to ensure that the design as implemented meets the requirements of the functional specification. Just like other specifications, a threat model is a living document-as you change the design, you need to go back and update your threat model to see if any new threats appear. A threat modeling is considered the most effective way of handling risks at the architecture and design phase [26] but it is also the most cost technique [25].

Most software security vulnerabilities are design issues rather than coding issues. Design security issues like business logic flaws cannot be spotted in code and need to be examined by carrying threat modeling and abuse case modeling during the design stage. Threat modeling is by nature an iterative endeavor used to identify threats. It starts by identifying the security objectives of the software as described in the security requirements. Threat modeling breaks the software into physical and logical constructs, generating software artifacts that include data flow diagrams, end-to-end deployment scenarios, documented entry, and exit points, protocols, components, identities, and services.

5. SECURE IMPLEMENTATION

The software security four pillars are secure by design, secure by default, secure by the implementation, and secure by communication. Secure By Design can be achieved by establishing trust boundaries, not reinvent the wheel, building the economy of mechanism, applying trust reluctance, achieving open design, minimizing the attack surface, and securing the weakest link. Secure by default can be achieved by using least privilege, using default deny, and failing securely. Secure by the implementation can be achieved by psychological acceptability, applying the least common mechanism, validating inputs, securing data at rest, preventing bypass attacks, auditing and verifying, and applying defense in depth. Secure by Communication can be achieved by securing trust relationships and applying good practices of secure communication.

Gary McGraw stated that complexity is regarded as the enemy of security [18]. Every software has a different level and structure of complexity [6]. Complexity in the structure of software that can be found because of the cluster of program elements contained by a program includes:

- Architectural complexity arises because of a high number of interconnected subsystems.
- Cyclomatic complexity arises because of a high number of loops and if statements.
- Cognitive complexity can be discovered by observing the behavior of users finding a solution to a particular task or problem. Its main focus is on the understandability aspect of the code.

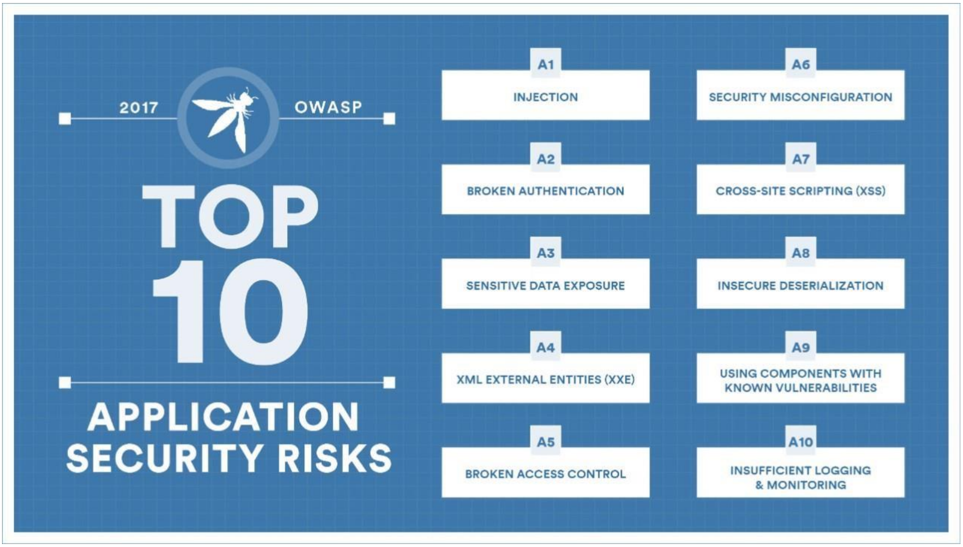

2017
OWASP
TOP 10
APPLICATION SECURITY RISKS
A1 INJECTION
A2 BROKEN AUTHENTICATION
A3 SENSITIVE DATA EXPOSURE
A4 XML EXTERNAL ENTITIES (XXE)
A5 BROKEN ACCESS CONTROL
A6 SECURITY MISCONFIGURATION
A7 CROSS-SITE SCRIPTING (XSS)
A8 INSECURE DESERIALIZATION
A9 USING COMPONENTS WITH KNOWN VULNERABILITIES
A10 INSUFFICIENT LOGGING & MONITORING

Figure 4. Open Web Application Security Project (OWASP) released Top 10 Most Critical Web Application Security Risks

Complexity in the inputs can also be a cause of introducing vulnerabilities in the system and huge security risks, such as application to the operating system or pages to web browsers.

To start secure software development, you need to understand insecure coding practices, and how these may be exploited [21]. Insecure designs can lead to" intentional errors", which is, the code is correctly implemented but the resulted software has a security vulnerability. Secure designs need an understanding of both types of requirements. Secure coding requires an understanding of the specifics of the implementation. Risk analysis is all about domain, assets, threats, and what-ifs. It is more global-minded and prioritization is critical. Whereas defensive coding is a defensive design form meant to make sure the continuation of the functioning of software under unexpected circumstances. The idea can be viewed as reducing or eliminating the prospect errors. Defensive programming techniques are used especially when a piece of software could be misused. In the case of having one small change in code that results in a big change in risk analysis. Figure 4 shows the Open Web Application Security Project (OWASP) [1] released the Top 10 Most Critical Web Application Security Risks.

There is a wide range of platforms, frameworks, and development tools, which cover some aspects of secure coding, but essential security controls are frequently missing, incomplete, or simply wrong. These gaps sometimes force developers to design and build home-grown security mechanisms, leading to wasted time and certain security issues and bugs. To use security controls is totally different than building them. Most developers should focus on using security controls rather than developing them. Enterprise Security API (ESAPI) is designed to include aspects of application security automatically, making these issues transparent to developers. The Enterprise Security API (ESAPI) project of the OWASP helps engineers guard against security-related design and implementation flaws. There is also the Top 10 Secure Coding Practices from CERT as language-independent recommendations [21].

6. SECURITY TESTING

Software testing can be defined by running a program or system with the intent of finding errors [23]. It is also can be seen as evaluating the capability of the system and determining that its requirements are met. Testing has different purposes improving quality, verification, and validation, and reliability estimation. Normally, software testing happens in multiple iterative phases. Each phase includes some security testing activities and is described within each phase: Unit testing, Integration testing, Quality assurance testing, User acceptance testing.

Security testing is to test security requirements with regard to security properties (i.e. integrity, confidentiality, availability, authorization, authentication, and nonrepudiation). Security testing tests if the intended security properties are correctly implemented or not. It shows conformance with security properties. Regular testing aims to ensure that the program meets customer requirements in terms of features and functionality. Usually tests normal use cases and with regards to common expected usage patterns. Security testing aims to ensure that the program fulfills security requirements which are often non-functional and more interested in misuse cases. The key steps to a clear successful security testing program are:

1. Identifying the Scope of Security Testing
2. Test Case Generation and Execution

The main objectives of the first step are: validate and verify that the applications meet the security requirements and find vulnerabilities in the given environment of the application. Conducting a rigorous security evaluation of an application is a complex job that should be approached like any other software analysis task-with a methodology, testing procedures, set of helpful tools, skills, and knowledge. Automated and manual penetration testing might be used to discover critical security vulnerabilities. Each phase of software development should have its own security testing strategy.

The security of an application is tested by attempting to violate built-in security controls [11]. This technique ensures that the protection mechanisms in the system are adequate enough to secure the application from improper and unauthorized access. The tester overloads the system with continuous requests, thereby denying service to others. The tester may deliberately cause system errors to violate security during recovery or may browse through insecure data to find the key to system entry. The following areas need to be tested for security, access controls, authentication, password management, input validation, exception handling, secure data storage and transmission, logging and monitoring and alerting, change management, and periodic security assessments and audits.

Buffer overflow, SQL injection, cross-site scripting, parameter tampering, cookie poisoning, hidden fields, debug options, unvalidated input, broken authorization, broken authentication, and session management are some of the areas around which the test cases should be generated for security testing. Ideally, security testing should be performed at the end of functional integration testing and performance testing [11]. This helps to detect hidden security threats in the application. After completing security testing, the findings should be summarized in a report. The summary report should contain details such as the types of testing conducted and the security risks identified, with ratings, which helps the business take a decision on deployment of the application. Developers can also conduct direct security testing using fuzzing techniques. Fuzzing is a method for discovering faults in software by providing unexpected input and monitoring for exceptions. Fuzzing, in simplest terms, is sending random data to the application program interfaces (APIs) that the program relies on and determining whether, when, and how it might break the software.

Risk-based testing is one of the common techniques of security testing [12]. The risks and their probability and impact guide the security efforts. Test effort is prioritized based on risk scores. High-risk areas are tested first, then medium-risk, and finally, low-risk areas. The main objective is to reach an acceptable level of risk. In other attempts to test the security of a software system, manual source code review can start when there is enough code to review. Manual source code review usually focuses on finding code-level security vulnerabilities. Manual source code review is also useful for no security flaws that can affect the overall code quality. Bug reports often contain a specific remediation recommendation by the reviewer so that the developer can fix it appropriately. Manual code reviews are expensive because they involve many manual efforts and often involve security specialists to assist in the review. However, manual reviews have proven their value repeatedly when it comes to accuracy and quality. They also help identify logic vulnerabilities that typically cannot be identified by automated static code analyzers.

Medium to large enterprises cannot afford to complete a manual code review on every single application every single time. Instead, many rely on automated source code analyzers to help. Typical software development priorities are schedule, cost, features and then quality-in most cases, in that order. The pressure from a time-to-market perspective can negatively affect software quality and resilience and sometimes causes the postponement of adding features to the software. Organizations with a mature SDLC process usually face little extra overhead because of software quality and resilience requirements, and the corresponding cost savings from process improvements far exceed the cost of added developer activities. They provide a wide variety of views/reports and trends on the security posture of the code base and can be used as an effective mechanism to collect metrics that indicate the progress and maturity of the software security activities. Source code analyzers operate in astonishingly quick time frames that would take several thousand man-hours to complete if they were done manually.

Although automated source code analyzers are strong at performing with low incremental costs, are good at catching the typical low-hanging fruits, have an ability to scale to several thousands of lines of code, and are good at performing repetitive tasks quickly, they also have a few drawbacks. Automated tools tend to report a high number of false positives. Sometimes it will take an organization several months to fine-tune the tool to reduce these false positives, but some level of noise will always remain in the findings. Source code analyzers are poor at detecting business logic flaws. Some of the other types of attacks that automated analysis cannot detect are complex information leakage, design flaws, subjective vulnerabilities such as cross-site request forgery, sophisticated race conditions, and multi-step-process attacks.

A well-known form of black-box security testing is Penetration Testing [9]. In a penetration test, an application or system is tested from the outside in a setup that is comparable to an actual attack from a malicious third party. Penetration testing is security testing in which evaluators attempt to bypass the security features of a system on the basis of their understanding of the system design and implementation. It is important to determine how vulnerable an organization's network is and the level of damage that can occur if the network is compromised. A penetration test can simulate both inside or outside attacks. Penetration testing is done not to only find possible vulnerabilities but to find the attack exploitability and the degree of the business impact of in case of a successful exploit.

7. MODELS FOR SECURE SOFTWARE DEVELOPMENT

These models can broadly be categorized as prescriptive and descriptive models. Prescriptive models provide guidance on what people should do while descriptive models for security software development provide a comparison of what one company is doing compared to the others. Descriptive models are used as benchmarks with other companies. Prescriptive models such as Microsoft Software Development Lifecycle, Comprehensive, Lightweight Application Security Process (CLASP) [14], Development, Security, and Operations (DevSecOps) (Figure 5) and descriptive models such as The Open Software Assurance Maturity Model (OpenSAMM) [2] and Building Security in Maturity Model (BSIMM) [19]. Microsoft SDL distributes security practices to different software development phases of the SDLC. CLASP is a predefined process element, which might be injected into any software development process while DevSecOps [22] has five main steps, Plan for Security, Engage the Developers and Be Engaged, Arm the Developers, Automate the Process, and Use Old Tools Wisely.

Among the descriptive models, the Building Security In Maturity Model (BSIMM) model encourages the stimulation of cultural change in creating secure software. The main objective of the project is to build a maturity model according to the collection of actual data from nine large-scale software development initiatives. OpenSAMM shows a well-defined maturity model (with tools of self-assessment and planning) for secure software development and deployment.

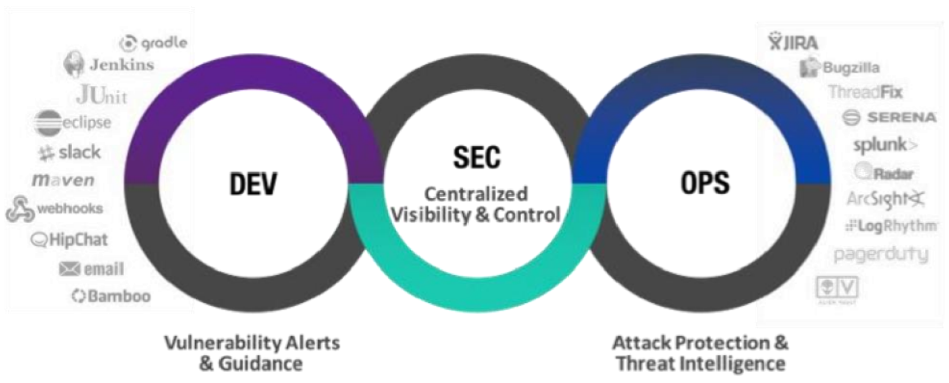


Figure 5. The DevSecOps Model

8. Conclusions

Many people and institutions have put their resources into various software products. Developing secure software is a necessity that can be achieved by implementing the discussed basics of secure software development as well as being in the lookout to counter new methods of breaching software security. The guide has clearly demonstrated that secure software development is not an extremely difficult field to be left to the IT (Information Technology) specialists only. The guide has provided easy to understand presentation of software systems security in a simple way that enables every stakeholder to those basic but vital measures. Researchers have a basis to develop further information based on the findings of the presented work. It has identified the available information, strategies, principles, and models that can be utilized in secure software development. Further research work can be done on the basis of the effectiveness of each of the principles, strategy and model presented.

References

[1] The Open Web Application Security Project (OWASP), 2020 (accessed February 12, 2020). https://owasp.org/.

[2] The OWASP SAMM Project, 2020 (accessed February 12, 2020). https:// wiki.owasp.org/index.php/OWASP_SAMM_Project.

[3] Thamer Al Hamed and Mamdouh Alenezi. Business continuity management & disaster recovery capabilities in saudi arabia ict businesses. International Journal of Hybrid Information Technology, 9(11):99–126, 2016.

[4] Mamdouh Alenezi, Alka Agrawal, Rajeev Kumar, and Raees˜Ahmad Khan. Evaluating performance of web application security through a fuzzy based hybrid multi-criteria decision-making approach: Design tactics perspective. IEEE Access, 8:25543–25556, 2020.

[5] Mamdouh Alenezi and Sadiq Almuairfi. Security risks in the software development lifecycle. International Journal of Recent Technology and Engineering (IJRTE), 8(13), 2019.

[6] Mamdouh Alenezi and Mohammad Zarour. On the relationship between software complexity and security. International Journal of Software Engineering & Applications (IJSEA), 11(1), 2020.

[7] Md Tarique˜Jamal Ansari, Dhirendra Pandey, and Mamdouh Alenezi. Store: Security threat oriented requirements engineering methodology. Journal of King Saud University-Computer and Information Sciences, 2018.

[8] Ivan Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, et˜al. Avoiding the top 10 software security design flaws. IEEE Computer Society Center for Secure Design (CSD), Tech. Rep, 2014.

[9] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. IEEE Security & Privacy, 3(1):84–87, 2005.

[10] Alexandre Bartel, Jacques Klein, Yves Le˜Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 274–277. IEEE, 2012.

[11] Michael Felderer, Matthias Bu¨chler, Martin Johns, Achim˜D Brucker, Ruth Breu, and Alexander Pretschner. Security testing: A survey. In Advances in Computers, volume 101, pages 1–51. Elsevier, 2016.

[12] Michael Felderer and Ina Schieferdecker. A taxonomy of risk-based testing. International Journal on Software Tools for Technology Transfer (STTT), 16(5):559– 568, 2014.

[13] Lynn Futcher and Rossouw von Solms. Guidelines for secure software development. In Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, pages 56–65, 2008.

[14] Johan Gregoire, Koen Buyens, Bart De˜Win, Riccardo Scandariato, and Wouter Joosen. On the secure software development process: Clasp and sdl compared. In Third International Workshop on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007), pages 1–1. IEEE, 2007.

[15] Naufil Kazi, Deepa Parasar, and Fauzia Siddiqui. A guide for security design. CYBERNOMICS, 1(6):37–40, 2019.

[16] Mike Mannion and Barry Keepence. Smart requirements.ACM SIGSOFT Software Engineering Notes, 20(2):42–47, 1995.

[17] John McDermott and Chris Fox. Using abuse case models for security requirements analysis. In Proceedings 15th Annual Computer Security Applications Conference (ACSAC'99), pages 55–64. IEEE, 1999.

[18] Gary McGraw. Software Security: Building Security In. Addison-Wesley Professional, 2006.

[19] Gary McGraw and Brian Chess. The building security in maturity model (BSIMM). In 18th USENIX Security Symposium (USENIX Security '09), Montreal, Quebec, August 2009. USENIX Association.

[20] Nancy R Mead and Ted Stehney. Security quality requirements engineering (square) methodology. ACM SIGSOFT Software Engineering Notes, 30(4):1–7, 2005.

[21] Mark S Merkow and Lakshmikanth Raghavan. Secure and resilient software development. Auerbach Publications, 2010.

[22] Havard Myrbakken and Ricardo Colomo-Palacios. Devsecops: a multivocal literature review. In International Conference on Software Process Improvement and Capability Determination, pages 17–29. Springer, 2017.

[23] Bruce Potter and Gary McGraw. Software security testing. IEEE Security & Privacy, 2(5):81–85, 2004.

[24] Sven Turpe. The trouble with security requirements. In 2017 IEEE 25th International Requirements Engineering Conference (RE), pages 122–133. IEEE, 2017.

[25] Elaine Venson, Xiaomeng Guo, Zidi Yan, and Barry Boehm. Costing secure software development: A systematic mapping study. In Proceedings of the 14th International Conference on Availability, Reliability and Security, pages 1–11, 2019.

[26] Wenjun Xiong and Robert Lagerstrom. Threat modeling–a systematic literature review. Computers & Security, 2019.