Best Practices for Software Security: An Overview

Ansar-Ul-Haque Yasar
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee Belgium
+32 16 32 75 35
ansarulhaque.yasar@cs.kuleuven.be

Davy Preuveneers
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee Belgium
+32 16 32 78 53
davy.preuveneers@cs.kuleuven.be

Yolande Berbers
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee Belgium
+32 16 32 76 36
yolande.berbers@cs.kuleuven.be

Ghasan Bhatti
Linkoping University
SE-58183 Linkoping Sweden
+46 76 23 23 375
shani572@hotmail.com

Abstract - **With the growth of software flaws there is a rise in the demand of security embedding to achieve the goal of secure software development in a more efficient manner. Different practices are in use to keep the software intact. These practices also meant to be scrutinized for better results on the basis of the level of security, efficiency and complexity they are providing. It may also be weighted on the basis of Confidentiality, Integrity and Availability (CIA). Software security is a step by step procedure which can not be achieved just at a specific level but it should be taken into account from the beginning of the Software Development Life Cycle (SDLC). In this paper, we have taken into account some of the best practices for secure software development and categorized them based on the phases in software development lifecycle. The results enable us to draw a clear picture of the best practices in software development which will enable a developer to follow them on a particular SDLC phase.**

Keywords: Software, Security, SDLC.

I. INTRODUCTION

With increasing technologies there is a more excess to the software which is a threat to the security of software. Hidden attacking factors from inside or outside the organization are increasing day by day. Intrusion and malicious software not only cause a financial loss but also the loss of credibility and integrity of organization data.

Software security issues directly affect the Confidentiality, Integrity and Availability (CIA). Security is not a feature it's a property of software which has to be taken care of during the complete software lifecycle. So the security model, which has to be implemented, must consider these issues effectively and efficiently. If the software is not secure, then all its operations are exposed to attacks.

As mentioned software security spread covers the whole Software Development Life Cycle phases. Each of the phases should be carefully observed for building secure software.

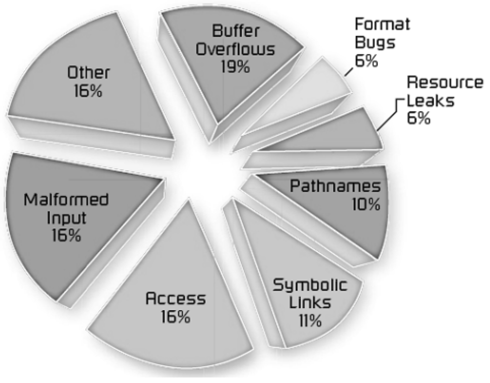

Figure-1: Summary of Vulnerability Types Found. [10]

II. BACKGROUND

While coding software every coder make some coding mistakes unintentionally, which introduce the majority of software vulnerabilities. Some of the examples of such vulnerabilities can be buffer overflows, integer overflow, format string vulnerabilities etc. Vulnerabilities in software can be classified as in figure 1.

"Software security is really risk management". So the risk must be analyzed first i.e. the analysis have to be made to evaluate the situations in which a specific or set of attacks initiate [3]. This can make easier to respond in a known situation. Risks can be appeared due to architectural problem or faults in implementation.

Software security practices involve dealing with risks associated with the operations and the effects of errors and vulnerabilities. The main aim of security for software is just to fall safely and decently in case of any crash with less or no damage done to the confidentiality integrity and availability.
Before going into any further details, the goals of software security must be clearly kept in the mind. Goals of security can be categorized as prevention, traceability and auditing, monitoring, privacy and confidentiality, multilevel security, anonymity, authentication, and integrity [3]. Any practice for software security can be weighted on the basis of achieving the

978-1-4244-2824-3/08/$25.00 ©2008 IEEE

above mentioned goals. The security countermeasures depend on the analysis of the threats and the vulnerabilities.

The threat analysis, the analysis of potential risks threatening the assets, must be practiced while planning the features and architecture the software. Threat analysis if performed with care and potential threads to the product can stabilize the system progress in the worst situation as well.

The response to the thread after the analysis is dependent on the scale of the risk incurring the thread. Once the threats are identified, there are three commonly recognized approaches [1]:

- *Mitigation of the risk*. The risk in this case is to be considered the harmful beyond the safety level so there is a response needed to reduce the risk up to a standard level.
- *Acceptance* i.e. the threat is accepted with its risk but not tackling the risk as it is but different plan of action designed to overcome the risk indirectly.
- *Insurance* this refers to as we redirect the vulnerability of a risk to a third party.

Similar to threat analysis for avoiding the potential problems we would also like to discuss about the basic principles for secure software according to John Viega and Gary McGraw[3]:

i. Secure the Weakest Link.
ii. Practice defense in depth.
iii. Fail Securely.
iv. Follow the principle of least privilege.
v. Compartmentalize.
vi. Keep it Simple.
vii. Promote privacy.
viii. Remember that hiding secrets is hard.
ix. Be reluctant to trust.
x. Use your community resources.

Viega and McGraw have explained that developers can avoid 90% of the problems using these principles [3].

a. Theoretical Methods

Methods like CIA, Risk Management and Formal Methods like Team Software Process (TSP), CLEANROOM Software Engineering (CRSE), Correctness-by-Construction (CC), Incremental Development (ID), Function Based Specification & Design (FBSD), Threat Modeling (TM), Attack Trees (AT) and Attack Patterns (AP) deal with the theoretical analysis. These methods may help implementing a particular practice for software security. We have not taken the theoretical methods into account for our analysis later on.

b. Practical Methods

We have assumed methods like automated verification and validation tools like ITS4, flawfinder, RATS, ESC/Java, PREfast, Rational Rose, Unified Modeling Language (UML) and SLAM as the practical methods. These methods may help implementing a particular practice for software security. We have shown a relationship of these methods with the best practices in a table later on in this paper.

III. Analysis and Comparison of the Best Practices

In this paper we have chosen three example set of best practices [1, 2, 11] and classify them based on the phases in software development process that these practices are applied.

a. Best Practices - Group 1

The first set of best practices we discuss here are introduced by Redwine and Davis [2]. These practices are:

i. Abuse Cases
ii. Security Requirement
iii. Risk Analysis
iv. External review
v. Risk-based security tests
vi. Static analysis
vii. Risk Analysis
viii. Penetration testing
ix. Security breaks

SDLC can be subdivided into some of the self-explanatory subgroups i.e. software artifacts according to the activities performed during that period. Software artifacts, as explained by [4], are *Requirements and use cases, Design, Test plans, Code, Test results, and Field Feedback.* At each specific artifact there are some software security practice are applied to build secure software. These can be summarized using the figure [2] below.

For each security artifact the practices need to be well understood and implemented in order to get the proper appropriate quality results.

At the requirement and use cases artifact the first practice is abuse cases which mainly focusing on the actions of the system in the presence of attack. At this point all the possible cases must be considered for which the system may work abnormally and marching to other steps after this consideration.

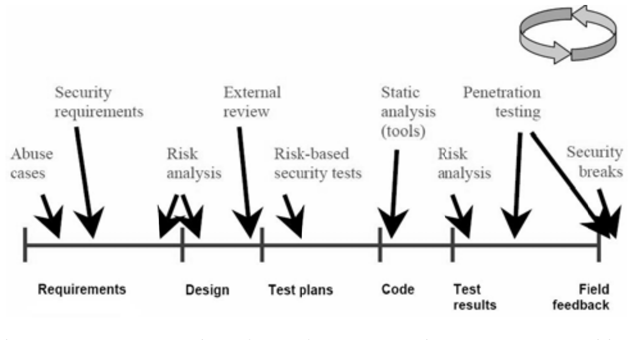

Figure – 2: Best Practices for Software Security are represented by the Arrows for a Specific Software Artifact [2].

After considering the abuse cases the next practice is security requirements i.e. to think about the security parameters that the system requires working securely. These may be practical, basic characteristics and features [2].

While at designing phase, a consistent approach is required as this phase mainly building the logical base of the whole software. Risk analysis requires all the known risk associated with any action or the attack must be clearly considered and a proposed plan of actions should be documented properly. Risk must be identified and the scale of its effect must be considered at this point so that the response associated with that risk can be clarified. Taking external help mostly helps positively as they may be the potential user with some practical knowledge. Clearly defined test plans on the basis of already considered risk are one of the best practices which enable the designer to justify his approach.

Majority of vulnerabilities are common and traceable so while coding there is a need of tools which act in a static or predefined way - static analysis tools. But not all the risks can be predefined so code review is not enough and further risk analysis is required on the basis of test results. There must be a continuous process defined through which testing of software security make possible.

Nothing is perfect in the world and software is a part of this world. There are always some security breaks. A feedback system should be planned such that after deployment enhance the database of known risks and propose solutions for future products [2].

b. Best Practices – Group 2

The second set of best practices we discuss here are introduced by P. Razvan [1]. These practices are:

i. Relevancy and Weakest Links
ii. Clarity
iii. Quality
iv. Involve all Stakeholders
v. Technology Processes
vi. Fail Safe Operation
vii. Defense in Depth
viii. Principle of Appropriate Privileges
ix. Interacting with Users
x. Trust Boundaries
xi. Third-Party Software
xii. Manipulation of Sensitive Data
xiii. Attack First. On Paper!
xiv. Think “Outside of the Box”
xv. Be Humble
xvi. Declarative vs. Programmatic
xvii. Reviews are Your Best Allies

Relevancy and Weakest Links concept relates to protect the related information and to secure the places which may be more prone to attacks.

Clarity says that the solution we provide should be clearly defined in every aspect.

Quality should be maintained in terms of the systems as well as the security features.

Involve all Stakeholders is a good concept of involving everyone who is some how related to the system being developed.

Technology Processes are related to each other and a technology should be able to invoke a process.

Fail Safe Operation is a concept of system or software being able to recover from the errors.

Defense in Depth means that to implement the defensive measures in a multi-layered environment.

Principle of Appropriate Privileges refers that everyone should be given privileges to operate the system accordingly as we also had to consider the development phase and not only the final product [1].

Interacting with the Users is an important practice to be considered as users tend to make mistakes and we can learn from them to develop countermeasures.

Trust Boundaries should be defined to enforce a reliable security mechanism. Trusted Computing Base and Trusted Path can be the subpart of this practice [1].

Third-Party Software which may be used while development of a totally new system should also be tested.

Manipulation of Sensitive Data should be handled with proper care.

Attack First. On Paper! It relates to the concept of trying to break the security architecture during the design phase [1].

Think “Outside of the Box” is an important concept which tells us that we should always try to find other possible ways of breaking into the system.

Be Humble is a concept related to continuously improving the system with time.

Declarative vs. Programmatic is related to the programming part of the system in which the security can be enforced either during the execution or outside of the program control [1].

Reviews are Your Best Allies is very much true and we should emphasis on design and code reviews for improvements.

c. Best Practices – Group 3

The third set of best practices we discuss here are introduced by D. Wheeler [11]. These practices are:

i. Validate All Input
ii. Avoid Buffer Overflow
iii. Secure the Interface
iv. Separate Data and Control
v. Minimize Privileges
vi. Minimize the Functionality of a Component
vii. Configure Safely and Use Safe Defaults
viii. Load Initialization Values Safely
ix. Fail Safe
x. Avoid Race Conditions

xi. Trust Only Trustworthy Channels
xii. Setup a Trusted Path
xiii. Use Internal Consistency
xiv. Self-limit Resources
xv. Prevent Cross Site Malicious Content
xvi. Foil Semantic Attacks
xvii. Be Careful with Data Types
xviii. Carefully Callout to Other Resources
xix. Send Information Back Judiciously

Validate All Input given to a system according to the rules defined.

Avoid Buffer Overflow as they may cause stack and heap smashing.

Secure the Interface explains that the interface should be simple enough to understand.

Separate Data and Control as an illegal execution may lead to loss of important data.

Minimize Privileges in order to limit the access as an intruder may able to get some information misusing the privileges.

Minimize the Functionality of a Component which we build to avoid misuse.

Configure Safely and Use Safe Defaults refers to making the initial system secure which can help to keep it secure while reconfiguration.

Load Initialization Values Safely as many programs read it to allow their defaults to be configured [11].

Fail Safe refers to a safe recovery from an error.

Avoid Race Conditions as they may reveal the shared resources.

Trust Only Trustworthy Channels as others may transmit unreliable data.

Setup a Trusted Path in order to avoid the leakage of secure information.

Use Internal Consistency to verify the basic assumptions we have made.

Self-limit Resources as abnormal termination may loose some information.

Prevent Cross Site Malicious Content as data from an unreliable source may lead to leak other user data.

Foil Semantic Attacks should be avoided as users may fall for a trap in which they may loose their personal information.

Be Careful with Data Types as using wrong data types may trigger buffer overflows.

Carefully Callout to Other Resources states that some resources like library routines may require some additional system information to operate. We should be careful as it may cause data loss.

Send Information Back Judiciously to minimize feedback for unreliable users.

d. Analysis of the Best Practices

In this section we have performed an analysis to classify the pre-mentioned set of best practices based on the phases in software development process that they are applied. The phases we assume here are based on the phases in traditional waterfall model: Requirement analysis, design, implementation, testing and maintenance. These phases can be more or less mapped to phases in other process models also. We are also introducing possible tools that support each of best practices.

TABLE - 1: Analysis of the Best Practices.

SDLC Phases/ Best Practices	**Best Practices (BP)**	**Tools Used**
Requirement Analysis	BP1.i*, BP1.ii*, BP1.iii	NA
	BP2.i*, BP2.ii*, BP2.iii	
	BP3.iii*, BP3.xvii*	
Design	BP1.iii*, BP1.iv*, BP1.v*	Rational Rose, UML
	BP2.iii*, BP2.v*, BP2.vi*, BP2.viii*, BP2.x*, BP2.xii, BP2.xiii*, BP2.xiv*	
	BP3.iii*, BP3.iv*, BP3.v*, BP3.vi*, BP3.ix*, BP3.x*, BP3.xi, BP3.xiii*, BP3.xviii	
Implementation	BP1.vi*	ITS4, RATS, FF, PREfast, ESC, SLAM
	BP2.iii*, BP2.v*, BP2.vi*, BP2.vii, BP2.viii*, BP2.xi*, BP2.xvi*	
	BP3.i*, BP3.ii*, BP3.v*, BP3.vii*, BP3.viii, BP3.ix*, BP3.xii*, BP3.xiv*, BP3.xv*	
Testing	BP1.vii*, BP1.viii*	ITS4, RATS, FF
	BP2.iv*, BP2.xvii	
	BP3.x*, BP3.xv*, BP3.xix	
Maintenance	BP1.viii, BP1.ix*	ITS4, RATS, FF
	BP2.ix, BP2.xv, BP2.xvii	
	BP3.vii, BP3.x*, BP3.xvi*	

Legend for Table – 1: Best Practices Group -1 (BP1), Best Practices Group -2 (BP2), Best Practices Group -3 (BP3), flawfinder (FF), Unified Modeling Language (UML).

(*) BP must be used otherwise may be used.

IV. Secure Software Testing

This section includes the testing factors and experiments which can deduce the desired results during the project's development life cycle. The testing factors [2] are:

i. Penetration testing
ii. Negative tests (fail safe)
iii. Fuzz (Black Box Testing)
iv. Risk Management
v. Ballista (an automated testing tool)

Penetration Testing plays an important part if the architectural risk analysis is the key factor behind the tests. It gives the clear picture of the software being tested in a real world scenario. It takes the software architecture into account which helps for a detailed software analysis.

Negative Tests (Fail Safe) deals with testing the software on negative cases and then try to understand the ability to recover.

Fuzz Testing is a black box testing that tests the software with pseudorandom inputs. In order to automate this black box testing *Ballista* an automated tool developed by CMU is being used [7].

Risk Management is actually not a part of the testing strategies but it was really important to mention about it in this paper. It involves threat identification, asset identification and quantitatively or qualitatively relating threats to assets. It may help the testing team to identify the real threats to software and then can enable them to rectify them according to the priority.

V. Conclusion and Recommendations

Based on the research for finding best practices of software security, we classified possible suggestions for improvement in existing practices which can make the software development more secure.

We would conclude that during our analysis of practices for building secure software, security is not a single step process; its span is all over the software development life cycle. Practices are mainly divided into different modules dealing with risks, management of risks i.e. identifying and scaling the risk along with proposing the plan of action for the specified vulnerability. The basic aim behind using the best practices is to provide the flexibility and improvement in the software quality and performance. Our classification of the best practices according to the SDLC phases will enable the stakeholders to emphasis on the specific practices in a particular phase.

We would like to recommend that the set of best practices classified according to the SDLC phases as mentioned earlier in *Table 1* must be properly documented as a set of rules to be followed. The standard which may be developed can help developers to ensure that the best practices are being followed for a secure development regardless of the scope of the software.

We also recommend that any best practices from the three groups we assumed can be used as a combination according to the scope of the software under development.
Emphasis on using the latest Languages which have a strong structural programming ability may also be used to overcome weaknesses.

Reference

[1] P. Razvan, "Best Practices for Secure Development", October 2001.

[2] S.T. Redwine, Jr. and N. Davis. "Process to Produce Secure Software", Nation Cyber Security Summit, March 2004.

[3] J. Viega and G. McGraw 2002, "Building secure software", Addison-Wesley, ISBN 0-201-72152-x.

[4] G. McGraw, "On the Horizon: The DIMACS Workshop on Software Security", IEEE Security and Privacy, March/April 2003.

[5] The Ballista Project, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/edrc-ballista/www/, visited on 20 April 2007.

[6] JTest, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, visited on 20 April 2007.

[7] SAP Analysis, "Open Source Static Analysis Tools for Security Testing of Java Web Applications", December 2006.

[8] P. Martin and C. Gail, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies", in the proceedings of ICSE 2002.

[9] H. Michael, "Secure Systems Begin with Knowing Your Threats – part I", http://security.devx.com/upload/free/Features/zones/security/articles/2000/09sept00/mh0900[2]-1.asp, visited on 20 April 2007.

[10] D. Evans and D. Larochelle, "Reported flaws in Common Vulnerabilities and Exposures Database", IEEE Software, January 2002.

[11] D.A. Wheeler, "Secure Programming for Linux and Unix HOWTO", GNU Free Documentation License, March 2003.