Al-Quds University

Deanship of Graduate Studies

Improving Software Security in Software Life Cycle Models

Ahmad Jamel Fahel

M.Sc. Thesis

Jerusalem – Palestine

1430 / 2010

Al-Quds University

Deanship of Graduate Studies

Computer Science Department

Improving Software Security in Software Life Cycle Models

Prepared By:

Ahmad Jamel Fahel

Supervisor:

Dr. Raid AL-Zaghal

Thesis Submitted in Partial fulfillment of requirements for the Master

Degree of Computer Science from Computer Science department of Al-Quds University

Jerusalem – Palestine

1430 / 2010

Al-Quds University

Deanship of Graduate Studies

Computer Science Department

Thesis Approval

Improving Software Security in Software Life Cycle Models

Prepared By: Ahmad Jamel Fahel
Registration No: 20714277

Supervisor: Dr. Raid AL Zaghal

Master thesis submitted and accepted. Date: 23/6/2010

The names and signatures of the examining committee members are as follows:

1- Head of Committee: Dr. Raid al-Zaghal	Signature: ……………..
2- Internal Examiner: Dr. Nidal Kafri	Signature: ……………..
3- External Examiner: Dr. Osama Marie	Signature: ……………..

Jerusalem – Palestine
1430 / 2010

Declaration

I certify that this thesis submitted for the degree of master of computer science is the result of my own research, except where otherwise acknowledged, and that this thesis (or any part of it) has not been submitted for a higher degree to any other university or institution.

Signature:

Ahmad Jamel Fahel

Acknowledgments

Praises and thanks always are to Allah, The creator, and the teacher.

Then, I would like first of all to state my thanks to my supervisor Dr. Raid Al-Zaghal for his great support, effort and advice during my study period and especially the course of my thesis.

Also, I am grateful to all teachers at Al-Quds University/Computer Science Department. This thesis would not have been possible unless their encouragement.

Our thanks are also extended to doctor Nidal Kafri, Dr. Osama Amin and all our academic staff.

And thanks to my mother, brothers and sisters. I will never forget the support and encouragement from my wife; my great thanks to them for their love and inspiration.

Dedication

To my parents, wife, brothers and sister

To all Alquds University friends and colleagues

To all postgraduate students at Al-Quds University

To all those who help me during my study

Abstract

Software security is a major issue in software engineering, and the principles of software security are very clear to understand, but they are usually hard to implement. This is due to many security vulnerabilities that deter achieving a high level of security in software systems.

In this thesis, I have collected information on relevant security vulnerabilities; I described and classified them into levels according to their risk degrees. To do that, I have built a model based on different stages: (1) a learning stage to give the system engineer full and clear information about these security vulnerabilities, (2) a prediction stage that depends on the collected information to predict the possibility of each vulnerability and its effect (harm level) on the system, (3) in the scenario stage, the system engineer writes one or more scenarios to describe the circumstances (how and where) that would lead for each vulnerability and then suggests a preventive plan to avoid that vulnerability, (4) in the testing stage, the software is tested with all predictions on spot by running a fuzzy test to be sure that the software is secure against known vulnerabilities, (5) and in the final stage, I write the implementation for the system auditor to check the overall security level of the software.

We have suggested a plan to integrate this model into the four common phases of the software development lifecycle.

ملخص الرسالة

تعد حماية البرامج من الثغرات من اهم الامور التي تناقش مرارا وتكرارا نتيجة ظهور أنواع متعددة من الثغرات, ويعد العمل على حماية البرامج من الامور الصعبة في بعض الميادين لان بعض البرامج تتعرض لثغرات نتيجة لغة البرمجة المستخدم والبيئة اللتي يستخدم بها البرنامج , اضافة الى الفهم الصحيح للثغرات الامنية اللتي لا تطرح عادة ضمن مراحل حياة البرنامج على اختلاف انواعها.

إبتدأت في هذه الرسالة بجمع الثغرات الامنية وجميع المعلومات عنها من مصادر مختلفة , و قمت بتصنيف هذه الثغرات بحسب اهميتها ونوع لغات البرمجة اللتي تستطيع هذه الثغرة اختراقها او الانظمة المستضيفة للبرامج وكيفية الوقاية من الثغرات او تجنب حدوثها.

وبعد دراسة مستفيضة للثغرات الامنية قمت ببناء نموذج مبسط لتسهيل تجنب الثغرات الامنية ولتعليم المستخدم لهذه النموذج الثغرات الامنية التي قد تواجهه اثناء تطويره نظام معين ويتكون هذا النموذج من المراحل التالية :

1- التعلم : يتم من خلاله المرور على جميع الثغرات الامنية ودراستها جيدا
2- التوقع : يتم جمع جميع الثغرات الامنية المتوقع حدوثها بناءً على معطيات البرنامج المنوي عمله
3- كتابة طريقة تجنب حدوث الثغرة
4- تطبيق الطريقة المكتوبة في الفقرة السابقة
5- فحص البرنامج بناءً على التوقعات المحتملة
6- كتابة تقرير بالثغرات اللتي تم التنبه لها

وبعد ان قمت ببناء النموذج اقترحت الية دمجه مع آليات تطوير البرامج المستخدمة من قبل المطورين , والية فحص البرنامج من ناحية امان البرنامج , والية قياس النموذج المقترح بناء على المراحل اللتي يتم بها تطوير البرنامج.

Structure of Thesis

This research contains six chapters: the first chapter presents an introduction on software security and gives a short problem description, the second chapter gives a background on software security and illustrates relevant definitions and concepts, the third chapter discusses related works and other modules on the security lifecycles, the forth chapter presents our model and how to test software security and test security models, and also contains data on how to integrate our model with other software lifecycles models. The fifth chapter presents a real case on how to use this model with web applications, and the sixth chapter presents future work.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 Purpose Statement

This work aims at giving more focus on software security vulnerabilities and to enable software engineers and system developers to manage the issue of software security in all the phases of the software development lifecycle in a systematic way based on concrete knowledge. Towards that end, I have developed a new model that employs risk analysis techniques and rigorous testing rather than mere expectation and intuitive decisions.

1.2 Thesis Target Audience

Since software security is relatively a new field, it keeps attracting both researchers and developers, so this work can benefit system architects, system analysts, programmers, testers, and quality assurance personnel.

This work is also suitable for academic institutions and software engineering educators, since it can help them understand the concepts of the software security and be able to predict security vulnerabilities. It also presents a simple and clear model that enables software engineers to solve security vulnerabilities during software design and development.

1.3 Objectives

Towards the general aims outlined above, we try to achieve the following objectives:

- Identify the security concepts
- Identify the need for secure software
- Identify the missing parts of related research work
- Identify the types of security vulnerabilities
- Introduce simple security vulnerabilities preventive model
- Integrate this model into the four common software design models (Waterfall, Agile, Extreme Programming, and Iterative)
- Present a testing measurement for software and security model.
- Give examples of how to use this model with real example

1.4 Problem Description

Usually developers identify the security as an authentication and authorization issue, and they mix between security needs and system's requirements, and when I started to identify the security concepts for this thesis and its vulnerabilities, I found that I missed most of the part regarding security and its meaning.

Software security problems appear in small software applications and even in enterprise applications. Similar mistakes are repeated by many developers. Most developers use one of the four common lifecycle models, and yet they have faced buffer overflows for example, and therefore many scenarios were put to solve this problem.

Many developers ignore security requirements since they are hard to describe and complex to implement, especially when the software becomes very large, and with distributed systems the problem becomes even more difficult.

Another point to mention is that Security is hard to be measured after the software has been delivered as a closed box, and it will be difficult to test and modify the software by another person.

By looking at OWASP “yearly top ten most critical Web application security risks” report [34], we found that 8 out of 10 are still in the most top 10 and the awareness of them is not increased. The graph below from this report shows the differences between 2007 to 2010 top 10 most critical web application security risks

OWASP Top 10 - 2007 (Previous Version)	OWASP Top 10 - 2010 (Current Version)
A2-Injection Flaws	A1-Injection
A1-Cross Site Scripting (XSS)	A2-Cross Site Scripting (XSS)
A7-Broken Authentication and Session Management	A3-Broken Authentication and Session Management
A4-Insecure Direct Object Reference	A4-Insecure Direct Object References
A5-Cross Site Request Forgery (CSRF)	A5-Cross Site Request Forgery (CSRF)
(was T10 2004 A10 - Insecure Configuration Management)	A6 Security Misconfiguration (NEW)
A8-Insecure Cryptographic Storage	A7-Insecure Cryptographic Storage
A10-Failure to Restrict URL Access	A8-Failure to Restrict URL Access
A9-Insecure Communications	A9-Insufficient Transport Layer Protection
(not in 2007 Top 10)	A10-Unvalidated Redirects and Forwards (NEW)
A3-Malicious File Execution	<dropped from 2010 Top 10>
A6-Information Leakage and Improper Error Handling	<dropped from 2010 Top 10>

Table 1.1 Top 10 security risks

1.5 The Need for Secure Software

Usually system developers, designers, architectures and requirement analysts are unaware of the concepts of software security and give little or no consideration to security during the development process. This has caused a large number of security problems within the software.

CERT Coordination Centre [2] mentions that 90% of posted software security problems took place because of mistakes on the design phase of that software, bad coding style, or misunderstanding of the environment that the software will be deployed in.

Security problems cause customers a huge loss of data, money and even the trust of the company for which they have developed the software.

And from my real life experience, I found that any breach in software security causes the customers a big loss of money that affects the company’s reputation and usually other related customers as well. I experienced many cases that have occurred due to software security issues that caused a huge loss and would have been easily avoided if minor mistakes in deployment were discovered.

1.6 Required Qualities of Security

Secure software can be defined as the “software that is resistant to intentional attack as well as unintentional failures, defects and accidents” [10].

From this definition we have to be aware that each software should be secure as a means of prevention from all available kinds of vulnerabilities. And to reach that goal we have to collect all available information about security vulnerabilities and put forward suitable plans so as to avoid them and make our software immune against attackers. We have to keep in mind that the attackers also collect this information and are usually aware of these vulnerabilities, and our mission is to protect our software against their knowledge.

Another quality that we have to take into consideration is the performance of our software, if security overheads will reduce software performance significantly, then we are going in the wrong direction! Security measures that are built on concrete knowledge and planning won’t affect performance significantly, but if we look at the security issue after the software has been completed and delivered, this will definitely reduce software performance significantly.

Measurement of security is another key of quality, if we can explain the level that we reach in building this software this means that we admit that we provide all our best to protect our software and we don’t hide any information about the level of security we reach.

Availability in system represents the service or functionality that is available when it is required, but as for security, it means that it is available every time not just when it is required, so we can measure the quality of security availability by the following rule: if security is missed once in the software, then it does not exist at all!

2. Background

This chapter presents the concept of software security stated in the literature. This chapter will serve as a context building for upcoming discussion about improving security in the next chapters.

2.1 Background:

Software engineers start to think of the importance of the security issue when security vulnerabilities where found in their software. When customer's private data is attacked and stolen by hackers, they blame software engineers for being less educated and cannot present secure software to their customers. So software engineers start to learn from their mistakes to avoid attacks and start writing scenarios on how to avoid security vulnerabilities and keep their customers satisfied with their work in order to keep giving life to the software development.

The current development lifecycle views security as a useful case, procedures and testing but not as logical thinking to avoid vulnerabilities. Experts develop checklists, training programs, how to test manuals that handles few cases of security, but not whole phases on software lifecycle.

2.2 General Concepts of Software Security

Definition of software security is all about Confidentiality, Integrity and Availability (CIA), in addition to, Authenticity, on-repudiation and Risk management and that stem from the software security definitions [5]:

- **Confidentiality** is the property of preventing disclosure of information to unauthorized individuals or systems.
- **Integrity** means that data cannot be modified without authorization, and maintain consistency of data.
- **Availability** : mean that the information must be available when it is needed Security Vulnerabilities resources
- **Authenticity** : mean to ensure that the data, transactions, communications or documents (electronic or physical) are genuine
- **Non-repudiation**: implies one's intention to fulfill their obligations to a contract. It also implies that one party of a transaction cannot deny having received a transaction nor can the other party deny having sent a transaction.
- **Risk management**: A comprehensive treatment of the topic of risk management is beyond the scope of this article. However, a useful definition of risk management will be provided as well as some basic terminology and a commonly used process for risk management.

2.3 Software Security Definitions

Here we present some of the definitions of software security as they appear in the literature:

- “Software Security is the ability of the software to resist, tolerate, and recover from events that intentionally threaten its dependability” [2]
- “Software Security is about building secure software: designing software to be secure, making sure that software is secure, and educating software developers, architects, and users about how to build secure things” [3]
- “The idea of engineering software that continues to function correctly under malicious attack” [5]
- “The process of designing, building, and testing software for security” [8]
- “Defends against software exploit by building software to be secure in the first place, mostly be getting the design right (which is hard) and avoiding common mistakes (which is easy)" [5]
- “Software Security is system-wide issue that takes into account both security mechanisms (such as access control) and design for security (such as robust design that make software attacks difficult)”[5]

According to all above definitions of software security we can conclude that they all focus on making the software more robust that can resist attacks, while keeping private data away from attackers.

Therefore we can define software security as follows: ***"The ability to analyze and measure the level of security that has been reached during the development lifecycle to avoid any security vulnerabilities and risks"***

2.4 Resources of Security Vulnerabilities

In order to provide a new approach for integrating the security requirement in the software lifecycle – contrary to what other researchers have been doing in the past few decades, I have been working for two years trying to collect and identify all kinds of security vulnerabilities that affect software systems in general. As a result, I was able to identify 160 different security vulnerabilities in different types and flavors. Most of these can be considered as deadly (or catastrophic) vulnerabilities! I have included a list of all these vulnerabilities in the appendix. I summarized below the sources of these vulnerabilities:

1- **CERT** (Computer Emergency Response Team) [2]: is considered as an American governmental institute that is responsible for helping other governmental organizations on security vulnerabilities.

2- **OWASP** (Open Web Application Security Project) [6]: Worldwide free and open community focused on improving the security of application software. I have obtained most of the security vulnerabilities information from this resource.

3- **19 Deadly Sins of Software Security** [36]: a book that contains information about 19 vulnerabilities. This book had helped me with its idea about discovering the vulnerabilities before software failures.

2.5 A Taxonomy of Software Security Terms

Here I summarize most of the software security terms and vocabulary that are used in this thesis and in many software security books and the literature in general, and hereby I list the common ones and their meanings to make them clear and easier for the reader to understand. [1]

Access Control List (ACL): A data structure or list that is maintained to track what users or groups have permissions to perform what actions. *This is a Windows term.*

Attack: A particular instance of an attempted introduction of one or more exploits to a system.

Attacker: Someone who is trying to bypass the security of one or more pieces of the software in order to carry out some malicious agenda.

Backdoor: A piece of malicious software that is installed and left running to provide a way for an attacker to regain system access at a later time.

Cracker: Someone who "cracks" through software security, particularly licensing and copy protection. It is thought to have its roots in "safe cracker." This term isn't often used, in part because it is more narrowly focused and in part because it is just not as widely known and the differentiation between a hacker and a cracker is not clear yet.

Cracking: The act of circumventing the copyright protection, licensing, or the registration functionality of the software.

Daemon: A piece of software running in the background, usually as a process. Sometimes used interchangeably with "demon" in Unix® (The Open Group, San Francisco, California) term.

Denial of Service (DoS): Where legitimate users are prevented from accessing services or resources they would normally be able to access.

Distributed Denial of Service (DDoS): Where legitimate users are prevented from accessing services or resources by a coordinated attack from multiple sources.

Escalation of Privilege: When attackers illegitimately gain more functionality or access than they are authorized to have.

Ethical Hacker: One that performs penetration tests. Sometimes ethical hackers are also called "white hats."

Exploit: A code, a technique, or a program that takes advantage of a vulnerability to access an asset.

Firewall: An application or hardware appliance designed to diminish the chances of an attack by limiting specific types of information that can pass into or out of a system or a network.

Hacker: Someone who "hacks" programs, i.e., writes them in a particularly haphazard or unorganized manner. This wasn't originally a term that was specific to attackers, but in the last few years it has become an often-used synonym for attackers, especially in the press.

Hijacking: A situation when an attacker takes over control of one side of a two-sided conversation or connection.

Hub: A networking device that repeats the network packets on the physical network layer among many devices.

Information Disclosure: A situation when an attacker is able to access information he or she shouldn't be able to.

Intrusion Detection System: An application that monitors a system or network and reports if it recognizes that the signs of an attack are present.

Leets peek: The stereotypical sign of a script kiddie where text is written with numbers substituted for letters. The name comes from "elite." For example, "leet" is often written as "1337" or "l33t." It's also seen a lot in gaming communities.

Media Access Control (MAC) Address: Also called the Physical Address, it is physically embedded in every network interface card (NIC) during the manufacturing process. MAC addresses are often treated as unique, although that is not actually guaranteed.

OSI Network Model/OSI Seven Layer Model: The Open Systems Interconnection

Reference Model. This is commonly used to explain at what point certain processes are taking place and how information travels.

Personally Identifiable Information (PII): Information that is private to the user or machine. Disclosing PII is a violation of user privacy and can be a part of identity theft problems.

Phishing: Social engineering on a large scale, usually to obtain things like login information, credit card numbers, etc.

Protocol Stack: A system that implements protocol behavior based on a series of the OSI Network Model.

Reverse Engineering: The act of wholly or partially recreating the algorithms or designs used in software. This is usually done without source code access.

Rootkit or Root Kit: A set of tools and scripts that an attacker installs after successfully compromising a system. These are designed to automate additional tasks including installing additional programs like key loggers, remote administration tools, packet sniffers, backdoors, etc. Kernel Rootkits are rootkits that hide themselves within the

Operating system's kernel, making them a lot more difficult to detect.

Router: A hardware device that routes traffic between two networks. It can also disguise the traffic from the network behind it to make it appear as if all traffic comes from a single system.

Script kiddie: The somewhat derogatory term for an attacker who primarily downloads and uses exploit code designed and written by others. "Script kiddie" tends to be used to signify a copy-cat type of attacker that is not particularly skilled or creative on his or her own. A script kiddie is also considered to be young, cocky, and brash.

Social Engineering: The process of tricking or convincing a user into volunteering information the hacker can later use. This is often focused on things that are either finance related or material for identity theft.

Spoofing: Impersonating someone or something else — such as another user or machine — in order to trick software security checks or users.

Switch: A hardware device similar to a hub but which knows the hardware (MAC) addresses of each machine connected to it. This is so it can transmit packets only to the individual machine it is addressed to. This has the positive side effect of reducing network traffic and noise.

Threat: A possible path to illegitimate access of an asset.

Trojan Horse: A piece of malicious software designed to deceive the victims by appearing to be a benign program that they may wish to use and thus are willing to download or install.

Virus: A piece of malicious software that is capable of spreading itself, typically as part of a piece of software or a file that is shared between users.

Vulnerability: A bug in the software that would allow an attacker to make use of a threat to illegitimately access an asset. All vulnerabilities are threats, but only unmitigated threats are vulnerabilities.

Zero-Day Exploit: A vulnerability that is exploited immediately after its discovery, often before the software company or the security community is aware of the vulnerability.

3. Related Work

3.1 Misuse Cases

The concept was created in the end of the 1990s by Guttorm Sindre of the Norwegian University of Science and Technology and Andreas L. Opdahl of the University of Bergen, Norway [35], we found a complete framework; Strategic Modeling Technique, which covers in details both analysis and modeling in terms of security improvements. The framework introduces the definition of misuse cases of technique for many developers. This technique expects developers to be experts in security issues and have good experience of software development analysis and modeling in order to be used correctly.

The following graph shows a scenario of misuse case and how the system analyst writes a scenario to counter attack for each one of the misuse
cases.

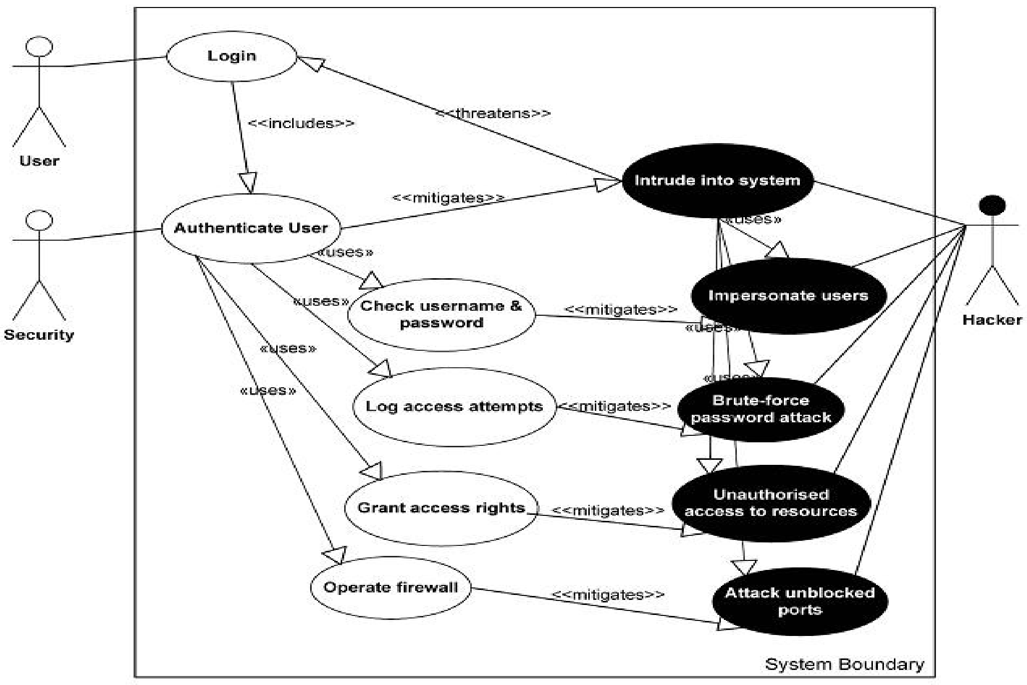


Figure 3.1 misuse cases

3.2 Nonfunctional Requirements

Another work comes from Chung et al [37], who classifies security requirements as non-functional requirements and present a general framework to deal with non-functional requirements to express them explicitly in the software life cycle. Chung believes that non-functional requirements are often subjective and relative. They introduced a set of sub-goals in order to satisfy a given security goal where the relationship between the sub-goals and the goal is either AND or OR relationships.

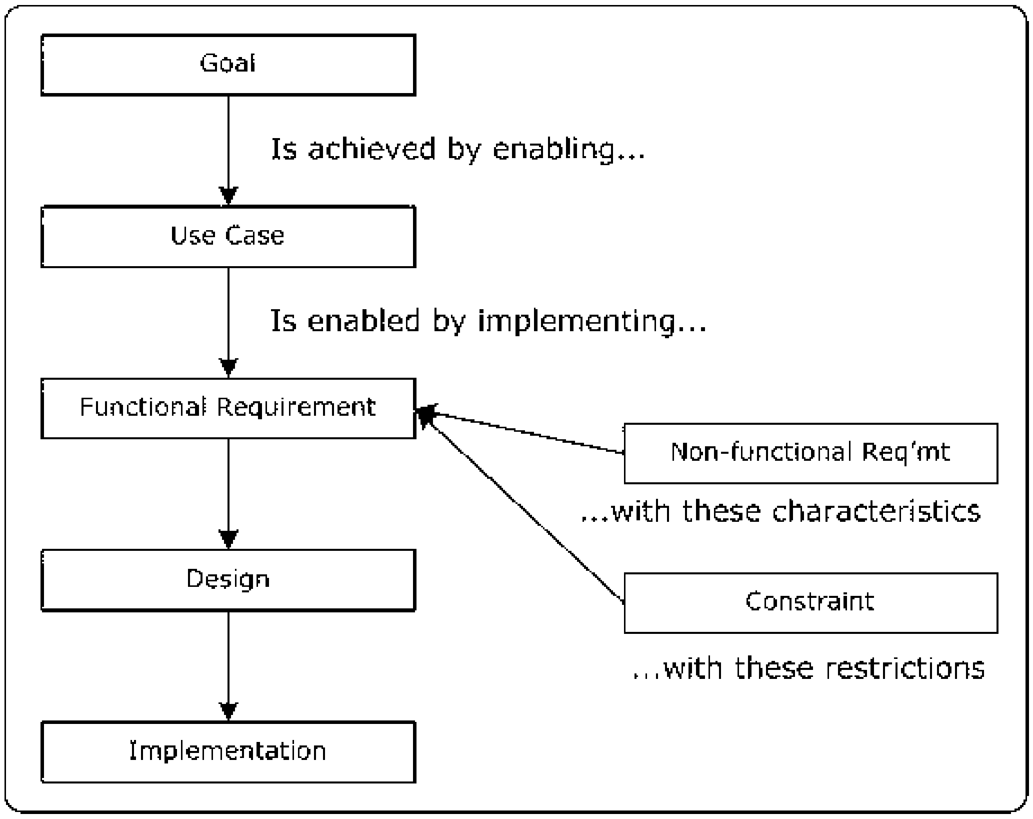

Goal
Is achieved by enabling...
Use Case
Is enabled by implementing...
Functional Requirement
Non-functional Req'mt
...with these characteristics
Design
Constraint
...with these restrictions
Implementation

Figure 3.2 Non Functional requirements

3.3 Spiral Model

This model was proposed by [37], and it is considered an extension of the iterative model assumes to have 4 phases, the first one is planning, the second is risk management, the third is development and testing and finally a plan for the next iteration.

This model wants the developer to check in each phase the possible risk that could happen, without showing the system architecture the potential problem.

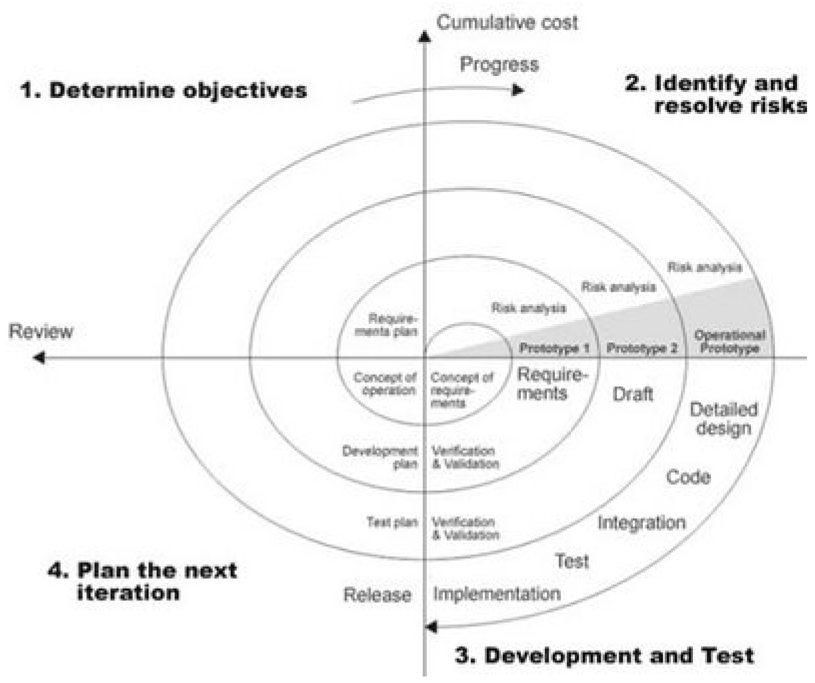
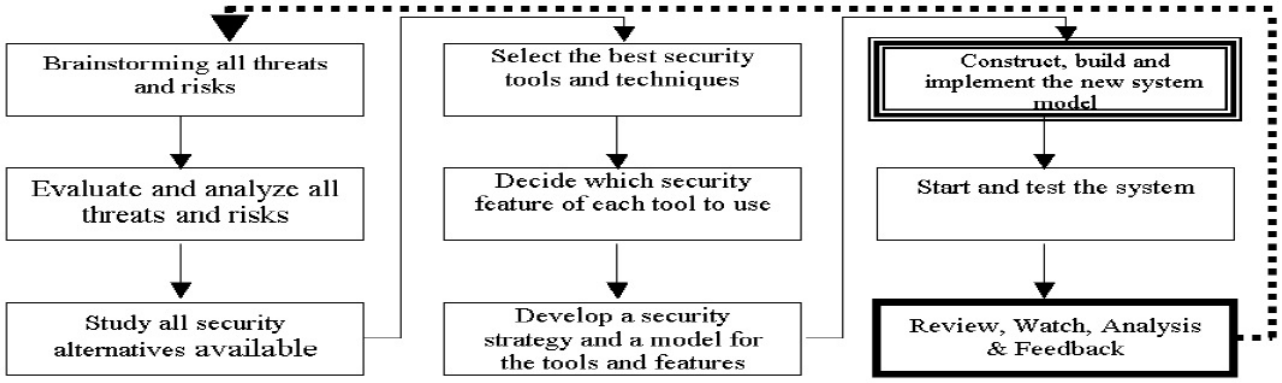


Figure 1.3 Spiral model

3.4 Security Model for E-Education Process

This research was presented in 2009 by [39], and provided a way of thinking of security vulnerabilities by using brainstorming to discuss system requirements, and try to figure out what could attack the system without providing previous knowledge about the security vulnerabilities.

Another point concerning this model that it depends on tools that are not mentioned in the model context and here is that what is considered as missing point to the developer.



Brainstorming all threats and risks
Evaluate and analyze all threats and risks
Study all security alternatives available
Select the best security tools and techniques
Decide which security feature of each tool to use
Develop a security strategy and a model for the tools and features
Construct, build and implement the new system model
Start and test the system
Review, Watch, Analysis & Feedback

Figure 3.2 Security Model for E-Education Process

3.5 Microsoft Security Development Lifecycle

Microsoft also provides its own security model to counter attack risk based on core training phase [28], even Microsoft has a long history in security vulnerabilities that attacks its products, bud this model is good for Microsoft developers.

This phase focuses on Microsoft products only and discusses the vulnerabilities that may attack Microsoft products only, and that is ignoring large amount of security vulnerabilities for other technologies and products.

Another point concerning Microsoft model that does not exist on our model, is that we create counter attack scenario and test this scenario to investigate these scenarios, whether it can prevent these vulnerabilities or not.

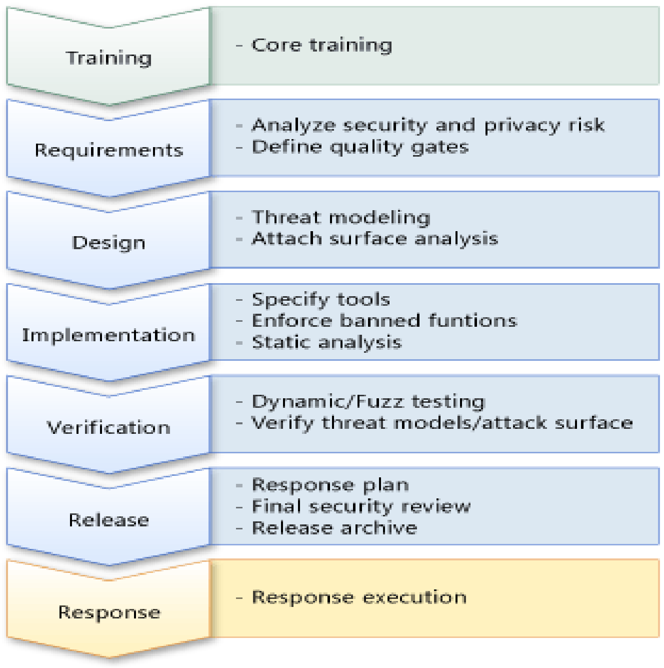


Figure 3.3 Microsoft security development lifecycle

3.6 Other Research Behaviors:

Software security researchers have written books on software security, solutions for many security problems, how to avoid them and how it could damage the software [8, 10]. Others wrote on how to test the software against security vulnerabilities, on risk management and how to keep the software in a stable status after being attacked [19, 32].

In my search for security concepts I found many security modules that can be followed to ensure security in software's based on good knowledge for security in the developers minds.

4. Our Model

4.1 Our Contribution

The main difference between my work and other works is that I am working first to collect all available information about security vulnerabilities and study them well, then start my model with a learning phase that provides the developer with the needed information about how to start thinking about security seriously and how to be aware of all possible vulnerabilities then to share information on the last stage to the next responsible person.

Another difference between my works and other software life cycle models is that I first focus on the security vulnerabilities and then I evolve my model into another model and I will explain later how to integrate my six layers into each model of the most known four models.

Our Model

In this chapter I will introduce my model depending on the works I have done in analyzing hundreds of security vulnerabilities supported from CERT.

In this thesis we define six stages for the security lifecycle model:

1- **Learning stage** : in this stage target audience will take the knowledge of the security Vulnerabilities that may occur in each phase of the software life cycle

2- **Predicting**: in this stage target audience will predict all possible vulnerabilities that may occur in each phase.

3- **Write scenario**: in this stage target audience will write the best scenario to complete this phase in order to completely pass all security vulnerabilities.

4- **Implementation:** in this part target the audience will do the required work for this part of work such as analysis or implementation … etc.

5- **Apply all tests**: in this stage target audience will apply all the related tests for his stage as I will explain how to get the test from the sheet that contains all security vulnerabilities.

6- **Documentations:** writing notes to next stage implementer is very meaningful for security life cycle since most of vulnerabilities occur due to miss information between people in each stage.

And in the rest part of this chapter I will write details for each phase of security life cycle with an example for each.

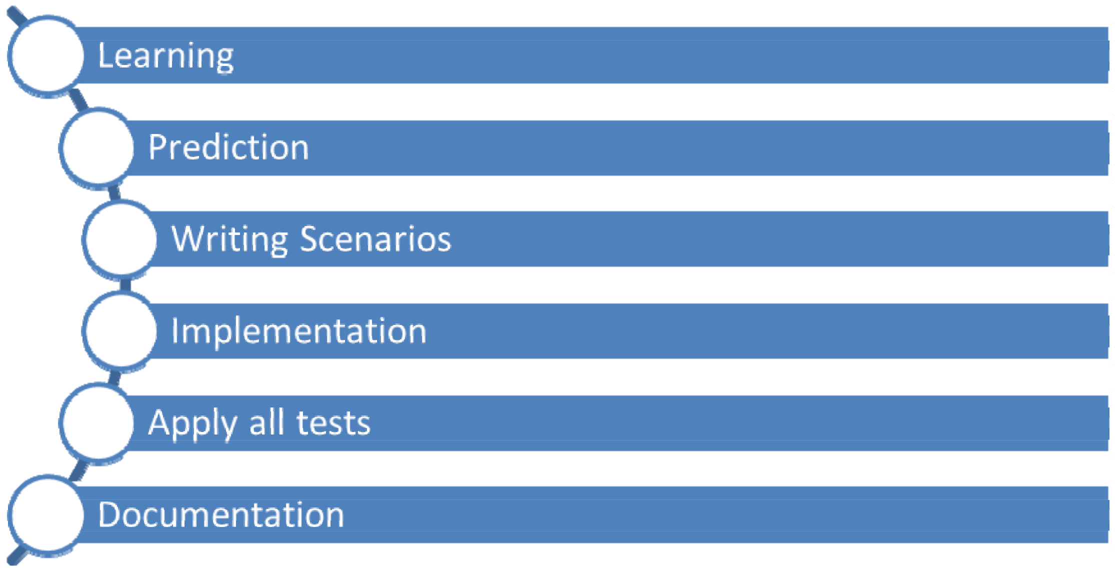


Figure 4.1 Model basic graph

4.1.1 Learning Stage:

In this stage we will take the knowledge and learn about the security vulnerabilities that may occur in the stage of implementation, this knowledge comes from a complete list of all security vulnerabilities provided by CERT.

Taking knowledge from the list will make it easy to target audience since this sheet provides the list with description of each problem and in which stage each vulnerability may occur, so it needs just to look at your required part and take knowledge of its information and will study the provided case.

The information listed in the sheet required only one time and then the knowledge is provided to each time it’s needed.

And if the sheet is updated, it will be easy to target audience to learn the new vulnerability and to take it into consideration since most of the vulnerabilities are dependent on the environments that are used.

4.1.2 Prediction Stage

After target audience take the knowledge of all above vulnerabilities it will be easy to predict what could happened during the work depending on his own style.

Many system architects don’t fall in any of those vulnerabilities without knowing the risk that may occur if they fail in one of them.

Predicting stage can be included into two stages in the software life cycle: the stage of writing the system requirements and analysis requirements. So the target audience will take in consideration all the system requirements and analysis in order to predict all the possible failure.

4.1.3 Writing Scenario

Writing scenario is the core stage of the security life cycle , and this means the how much we learn from the learning stage ,so the target audience will do the best to pass through all points in the list above and write complete system requirements and security requirements to avoid any chance of failure. And if we review the list in the prediction stage we will find that all of them are of type of warning and all are easy to take in consideration. Then the target audience will write system analysis and showing more details in writing each point to clarify to the system implementer all the warning as points so as to consider them as functional requirements.

Here, the meaning of writing scenario is to write all possible user workflow control and to insure that all vulnerabilities are prevented and none of them could occur to the system target that audience working on.

Scenario writer must write security vulnerabilities according to their reference and level, the reference means where this vulnerability comes from and the level estimates the level of damage that may occur if this security vulnerability is left without handling.

4.1.3.1 Security Vulnerabilities and References:

a. Technology security vulnerabilities
b. Implementation security vulnerabilities
c. Environment security vulnerabilities
d. User security vulnerabilities
e. Third party system security vulnerabilities
f. Integration security vulnerabilities
g. Business rules security vulnerabilities
h. Anonymous user vulnerabilities

4.1.3.2 Security Vulnerability Level:

I choose the name of the vulnerability level based on old phonetic alphabets just because old communication via radio signal used this name to give the listener the value of thing they agree on and so I choose the developer that agrees on such name to be common to the level of vulnerability

1- **Delta** (Severe) :this is the most dangerous level and this means it will be hard to restore system or system data after security vulnerabilities are attacked
2- **Charlie** (High) : this means that danger could cause loss the data, or either copy the data where it's considered as damage the customer business
3- **Bravo** (Elevated) : high dangerous, could cause stealing the customer business and make the customer lose money after this vulnerability is being attacked
4- **Alpha** (Controlled) : this level means that if these vulnerabilities attacke, system or data will still be safe but system behavior will not work as expected

5- **Echo** (system workflow hit): this is the least level in this scale and it means that everything will be kept as expected and there will be no loss of data or system, and the system behavior will be kept as expected ,but strange behavior may occurs in third party software or hosting environment.

And from the above specifications scenario writer can use the following template in order to fill the scenario:

#	Name	Description	Scenario steps	Reference	Level

Table4.1:Writing the scenario

This stage of the security life cycle for the first time seems very hard to the target audience since he has to think badly, and try to write all bad scenario that may affect the security of the system. Most system hacker use the knowledge in order to damage any system, they have to get a clear guess of how the system is built so as to find the system security vulnerabilities.

There are a lot of books that describe how to attack systems, all of them depend on knowing the system behavior and if they don't know how the system is built, they guess and build in their mind a similar system so they can guess the mistakes that the system developers failed to prevent.

Any other suggested form or sheet is not bad but I just try to provide applicable form to the scenario writer and he can write his scenario using his own words , and this means that the writing phase fully describes the level or education that the writer reach, his words give us clear vision of what he means, and for another purpose since some cases are different from each other, some of them can be written as notes ,others require full description and graph to be illustrated like Buffer Overflow and Buffer underwrite.

4.1.4 Implementation

Implementation here means that implementing security requirements are not the software implementation, and the system developer who works to implement this requirements has to take care of the written scenario and refer to the predicting scenario and get vulnerabilities information from the vulnerabilities sheet.

The main goal of this model is to make it easy for the implementer to apply all security requirements; this goal requires us to provide system developer with all required information, scenarios, and predicting all information for any vulnerability.

Implementation steps:

- Review all the available vulnerabilities and review that developer is aware of each one
- Review all predicting vulnerabilities
- Review and Pass through each vulnerabilities scenario
- Start to implements with the knowledge of the above steps

And when developer starts the implementation phase and code writing he will be more aware of the security vulnerabilities after being predicted and written as scenario from more than one person or more than one step.

4.1.5 Apply All Tests

Software security testing in other models require the tester to have more experience, more work to do, and to expect the unexpected vulnerabilities. While searching for testing security in software, many books described the security tester as a long experienced and very good knowledgeable person, who can discover security vulnerabilities that system developer is now aware of, but in this model, security tester has to follow the sheet provided by another phase to generate a list of tests that need to be done, and here the main advantage of my model is to make it straight forward for each phase in the model

After implementation we have to test whether the implementation blocks any security vulnerabilities or not, this test must be classified and sorted out so as not to ignore any vulnerability.

Test phase must be done by a person who must have the knowledge and the ability to rearrange the entire tests scenario to apply them with the order that does not ignore any vulnerability.

The meaning of ignoring vulnerabilities is that some vulnerability occurs as a result of vulnerabilities, such as “Using freed memory” that can cause to “Unintentional pointer scaling”, and serious tester must be aware of the meaning of vulnerabilities that make vulnerabilities.

And security test must not be ambiguous that’s to say it must be different from software test and each one has its own goals and its reference to depends on, even if the same person does both tests he should separate from the system functional tests.

4.1.5.1 Security Testing Requirements

1- Technology knowledge : tester must have a good experience in the technology used in this system, this is because new technologies are different from each other and each one has framework and vulnerabilities that can’t be applied to other technologies

2- Aware of Code Complexity: The more complex the code, the more likely it is to have security vulnerabilities as well.

3- Code Coverage: this means that tester has to pass over all the code that the software has, the percentage of code coverage is an important issue to determine the coverage of test and the security of the software.

4- Test environment :it is similar or close to the production environment

5- Describe your attacker: tester must have a good idea of just who these attackers are and what their skills and motivations are.

6- Define attacker’s goals: this means that any attackers have a goal to reach in order to hack your software, and the testers have to identify their goals to prevent them from attacking your software.

4.1.5.2 Testing Plan

As mentioned before, the security testing in this model is completely straightforward and needs the test just to implement using the available test that comes from our research about all security vulnerabilities, predicting phase and written scenario and collecting all these data and exposing them to testing matrix.

Testing matrix is a matrix that combines between the predicting phase of the written scenario and surely the implementation itself.

So all tests must be available to be done, the tester has to order test and make sure that the order is meaningful and not to ignore any test because of the order.

4.1.5.3 Time Plane for the Test

Tester must put test plan for the time of the tests so as to determine the time spent for executing each test in order not to have a conflict between the two tests and not waste the testing time.

Tester has to determine the required time to investigate each vulnerabilities, and since this requires the tester to be aware of the vulnerability itself and then to pass through the software code so as to check whether this vulnerability is handled with the code written or not.

The time needed for running all tests can be modify according to the test results, if tests start to succeed one after another then there is no need to stick completely to the time plan and test can go faster.

4.1.5.4 Fuzz Testing (Fuzzing)

The term Fuzz originated from Prof. Barton Miller's student assignment at the University of Wisconsin in the Fall of 1988, titled "Operating System Utility Program Reliability - The Fuzz Generator".[40] In quality assurance and testing, the same approach (using unexpected data or syntax) has been called robustness testing, syntax testing or negative testing. Even white-noise testing can be thought of as fuzzing.

Fuzz testing is mostly used to test the stability of the software since if you can post the software entry points data larger than what he expects, that may cause the software to hang-up and may cause to stop the software from being served to the customer, and this is considered as security vulnerability since it’s used to prevent the user from accessing the service.

And to insure that tester has done the fuzz testing in the testing phase, every security vulnerabilities that may occurs as result of fuzz testing must be included in the learning phase and during learning phase and expectation phase they can see that it’s available to be taken into consideration.

4.1.5.5 Result of the Tests

Result of the test should be clear with scenario of the fail, this means that the tester has to complete the scenario of the test, and write the scenario to the implementer showing how to do each test.

Result of the test also should have the reference of the test, this means that the tester must get to know where this test is from, and here in this model; is it from the expected vulnerabilities or from the learning stage and the expectation stage, and here the tester should be aware of the missing test from the expectation stage.

Table 4.2 describes show a form of the table result sheet.

Test order	**Test name**	**Test scenario**	**Test reference**	**Test time**	**Test result**	**Test note**

Table 4.2 : Result of the tests

4.1.6 Documentations

Writing notes in this security model gives a security auditor or any security test to know the ability of the level of the security in this software and how to add new vulnerability blocker to the software.

Writing notes is not the result of the security test; it is the description of what that software can apply of rules to prevent any security vulnerability that may attack the software.

4.1.6.1 Documentations Elements

Security notes contains the following elements

1- System description
2- Description of used technology
3- Environment developed on.
4- Environment deploy on.

5- Security vulnerability expected
6- Expected Security vulnerability blocker
7- Implementer notes on vulnerabilities
8- Tester notes

System Description

This section contains a description of the system and its aim and this must contain information on the analysis document that builds the system

Description of Used Technology

Systems may contain several technologies in the same project and the combination between technologies to build this system, since combining or connecting technologies presents security vulnerabilities.

Environment Developed On

Describing the development environment is a key issue to describe the changes in the development environment and deployment environment to keep the auditor aware of the vulnerabilities that may not checked during development.

Environment Deploy On.

To describe the target environment that the system will be deployed on, and to investigate the changes between development environment and deployment environment.

Security Vulnerability Expected

List of all expected vulnerabilities in the expectation stage to show the level of expectation reached during development

Expected Security Vulnerability Blocker

Describe the work done to prevent security vulnerabilities from attacking our software ability to improve blocker

Implementer Notes on Vulnerabilities
Notes of the system developer and system implementer about the security vulnerabilities that have been expected.

Tester Notes
Notes of the tester and notes about the result of security test, and this is to show to the auditor the level of acceptance that the tester reached.

4.1.7 Review of Our Model

During this chapter we passed through my model and showed the stage that I assumed will prevent security vulnerability based on knowledge before implementation.

- Learning is the phase where system engineer explores new information about security vulnerabilities.
- Prediction is the phase where system engineer expects security vulnerabilities based on the information.
- Writing scenario is the stage where system engineer describes how many security vulnerabilities occur.
- Implementation phase shows where prevention and blocking of the security vulnerabilities are done.
- Applying all test phase makes test to all security vulnerabilities and ensures that the system is secure for all expected vulnerabilities.
- Writing the note phase is the final stage and here we write to the IT auditor what we do on each environments and how we do it.

The main goal of this model is to facilitate the process of the security testing of software since most system engineer does not have good knowledge of security vulnerabilities and its level of danger to the system they develop and what the meaning of hacking system is.

Most people think that the hacker is a genius one who can discover attack, stall and destroy systems based on his intelligence and here we clear that all security vulnerabilities occur due to lake of knowledge and attacker got that knowledge.

4.2 Security Measurements

In this section we will discuss security measurement, how to measure my model according to common security measurement and how to measure any security software built on this model.

4.2.1 Software Security Measurable Entities

Before starting measure my model, we must agree on the entity to measure security at any software and then put the entity to measure my security model.

1- **Provided Level Of Protection**: this entity get its information from the result test and from the notes written on the software.

2- **Applying Customer Internal Policy**: if the security of the software violates the customer policy this means that this software is not fit for the requirements.

3- **System Performance**: the affected performance due to security of this software.

4- **Cost**: the extra money needed to secure this software against software budget.

5- **Time-Orientation**: needed time to secure the system: the extra time needed to secure software against the time needed to develop the software.

6- **Software Modification**: this issue reflects the level of security when modification is applied on the software.

7- **Integration with Other System**: this issue reflects the change on level of security due to integration of this software with other software's.

4.2.2 Security Model Measurable Entities

Measuring the model is different from measuring software, here we are talking about model and we have to measure it against other models, and rank it with models entities, and for that I listed my own entities that reflect the goals of this models

1- **Security Level Reached**: the security level reached in this model is high since I collect all kinds of security vulnerabilities and classified them into categories and provided them to the developer to help in protecting software.

2- **Usability of This Model :** as you can see in the next chapter I merged this model into the four common life cycle and made it usable and meaningful for the developer

3- Cost Saving: saving money that may be spent on security of the software after being deployed is greater rather than spending it during development.

4- **Time Saving:** secure software against all kinds of vulnerabilities is saving time in consideration to the time needed if software is attacked after being deployed where it’s possible to lose data and customers privacy. and after using this model for the first time, developer learning time is reduced and after a while it goes to zero time.

5- **Portability:** this model is portable for all kind of technologies and all kinds of programming language since it collects information for all kinds of security vulnerabilities.

6- **Documentation :** in this model each phase is concluded with documentation that help other phase until it reaches the last phase where documentation and notes deal with IT auditor level and represent the level of security that this software reaches.

4.3 Integrating with Common Software Lifecycle Models

In this chapter I will inject the security model in the four software lifecycle that are mostly used and I will show how to include it in any software lifecycle

4.3.1 Waterfall Lifecycle

The waterfall model is a sequential software development process, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing and Maintenance.

Waterfall life cycle is the most common used way of software engineering process and it is considered the simplest one in any other software lifecycles.

And figure 4.2 shows the steps those bases during this lifecycles

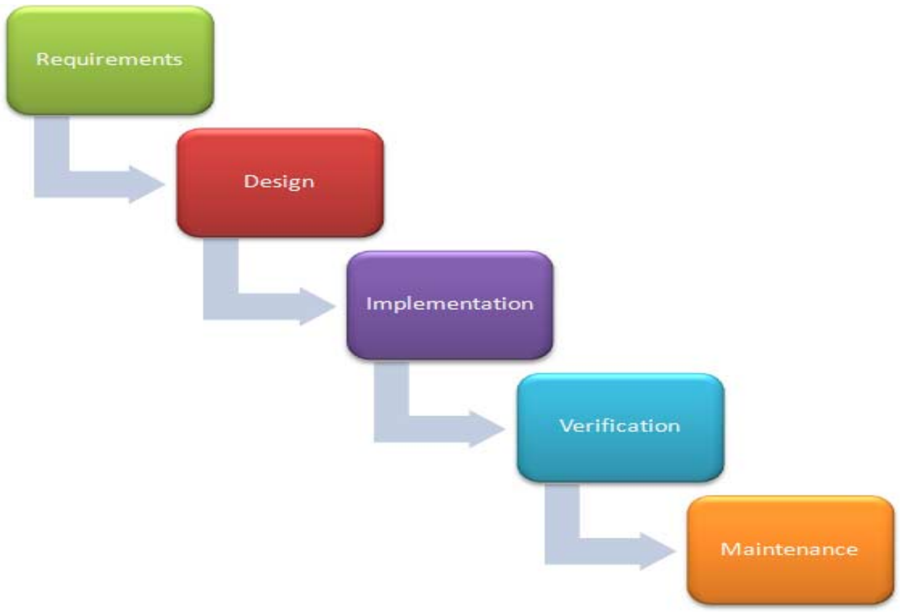


Figure 4.2 waterfall lifecycle

And to inject my model into this lifecycle I do the following:

1- Learning and predicting stages added to the requirements phase
2- Writing scenario added to the design stage
3- Implementation phase added to implementation phase
4- Applying all test added to the verification phase
5- Writing notes added after the software is launched

And graph 4.3 shows the new design of the waterfall that contains my module and each step inside it, and here I created a new step for software launch where writing notes is embedded in.

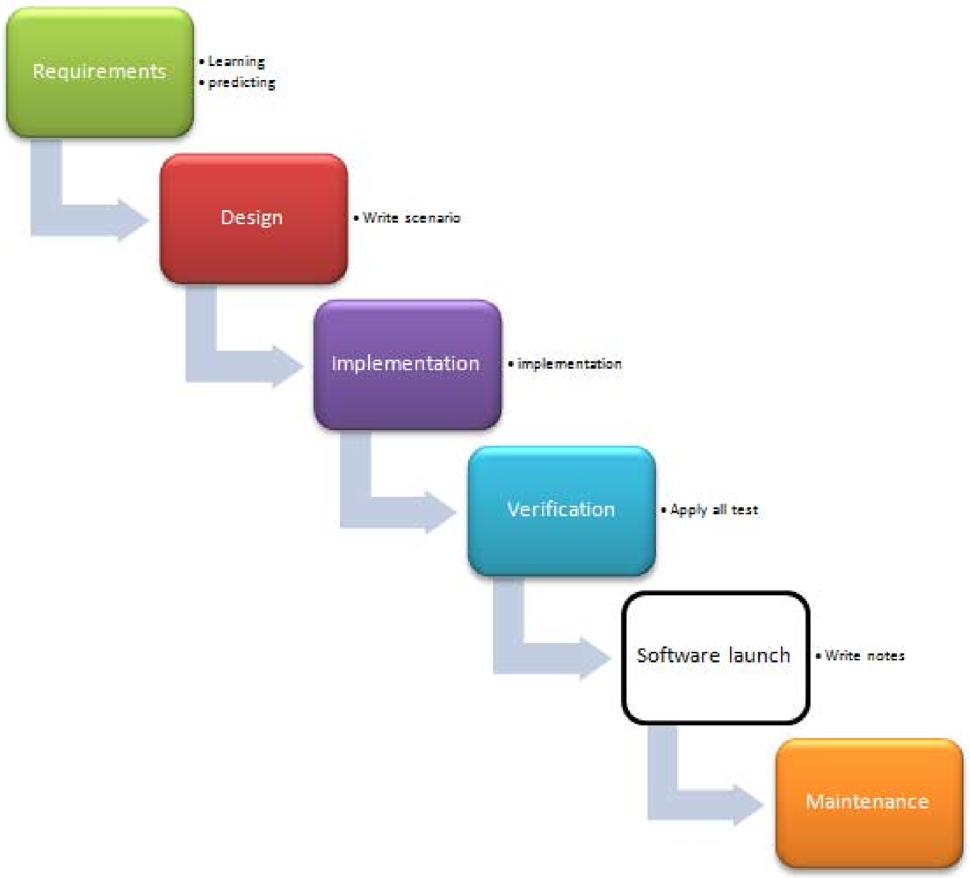


Figure 4.3 Waterfall with security model

4.3.2 Agile Software Development

Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

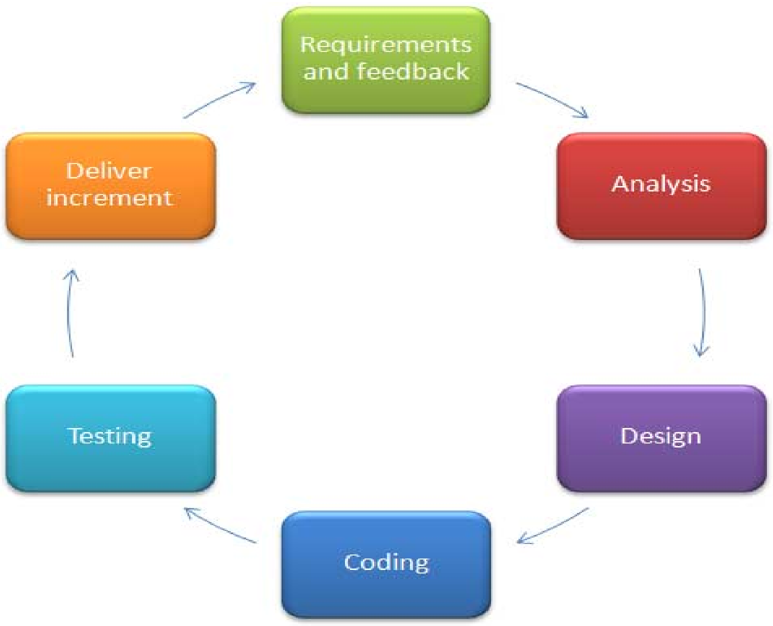


Figure 4.4 Agile development lifecycle

Agile processes use feedback, rather than planning so this can make a lot of security vulnerabilities and resolve them more quickly than waterfall model. And to add my model to this iterative model I did the following:

1- Learning phase before the iteration start.
2- Prediction and in the requirements and feedback.
3- Write scenario in the analysis stage.
4- Implementation in the coding stage
5- Apply all tests in the Testing stage.
6- Write notes in the delivers increment stage.

And graph 4.5 shows how the security models are embedded into the agile software lifecycle

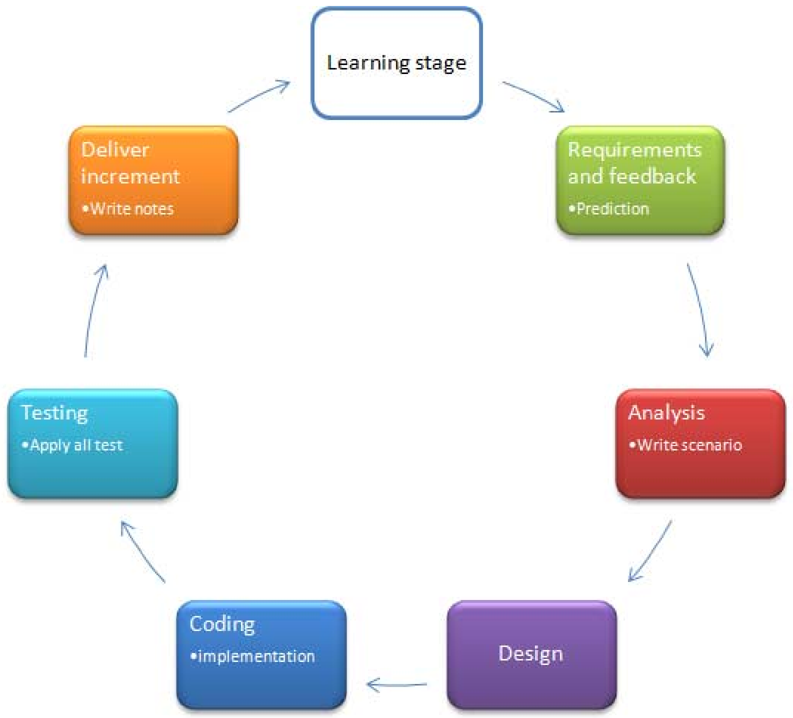


Figure 4 Agile with security model

4.3.3 Iterative and Incremental Development

Iterative and Incremental development is a cyclic software development process developed in response to the weaknesses of the waterfall model. It starts with an initial planning and ends with deployment with the cyclic interaction in between.

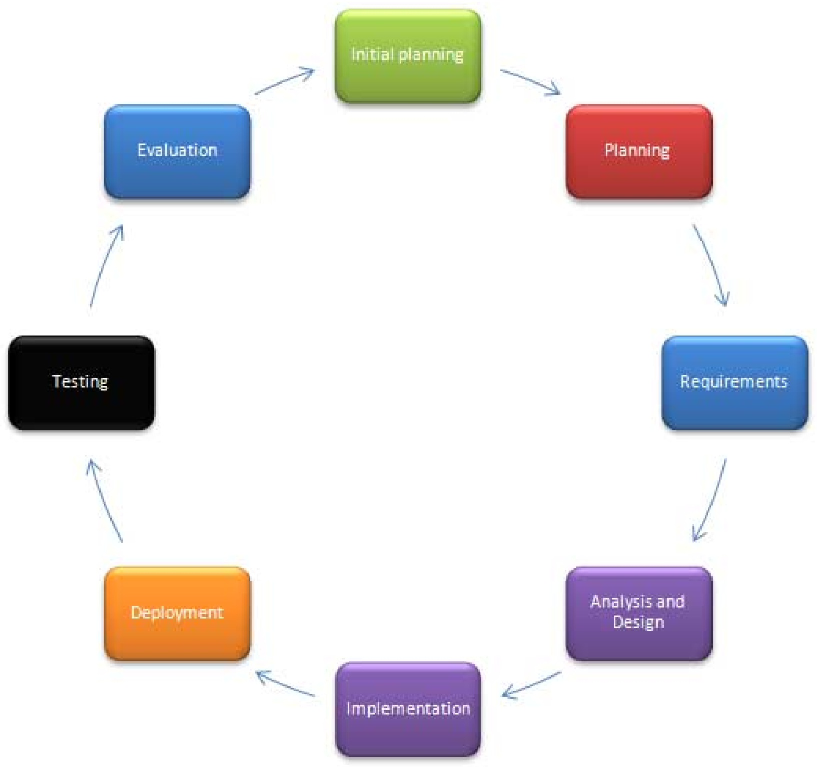


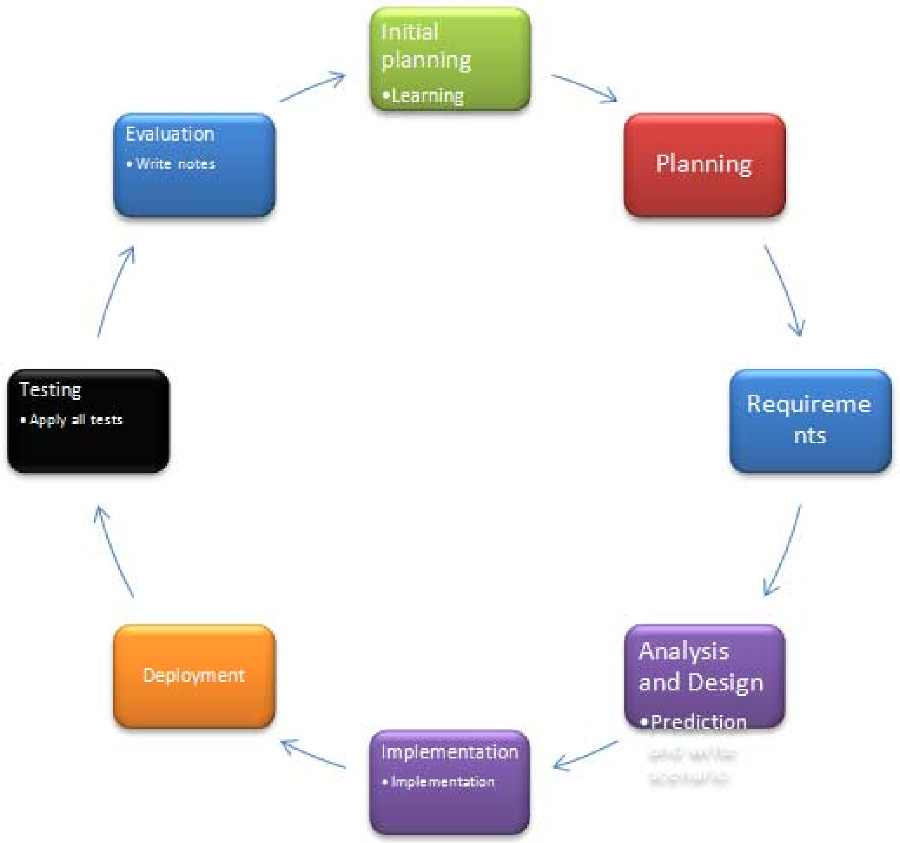
Figure 5 Iterative and Incremental Development

I found that this model is acceptable to my model more easily in its stage since it does evaluation phase then goes to another level and here they can apply all test and write notes to the next iteration.

And I add my model as the following

1- Learning stage comes with the initial planning.
2- Prediction and write scenario in the analysis & design stage.
3- Implementation phase in the implementation phase.
4- Apply all tests in the testing phase.
5- Write notes in the evaluation phase.

And graph 4.7 shows how the security models is embedded into the iterative and incremental



model

Figure 4.7 Iterative and Incremental Development with security model

4.3.4 XP: Extreme Programming

Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles (time boxing), which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

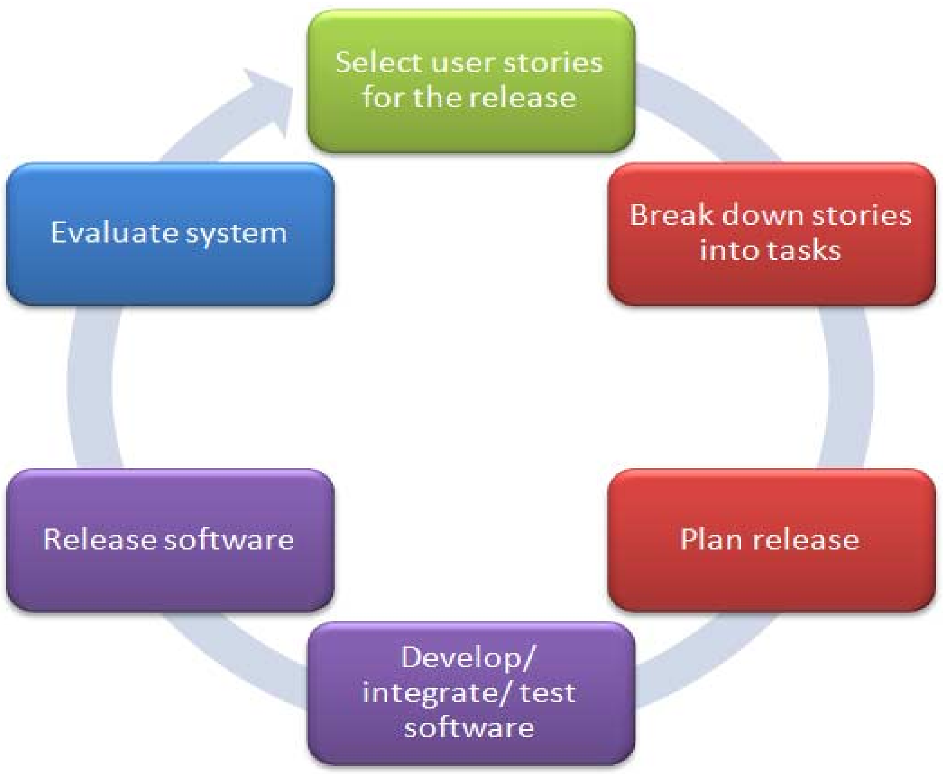

Select user stories for the release
Break down stories into tasks
Plan release
Develop/ integrate/ test software
Release software
Evaluate system

Figure 4.8 XP: Extreme Programming

Extreme programming is the best iterative process to work on continuous work that keeps and adds new feature and new requirements.

And to add my model into the XP model I do the following:

1- Learning stage comes before the iteration start
2- Prediction in the break down stories to tasks stage
3- Write scenario in the plan release
4- Implementation in the development stage
5- Apply all test comes before release software
6- Write notes comes in the evaluate system.

And graph 4.9 shows the XP model with security model embedded into it where you will find that learning stage is before the iterative start

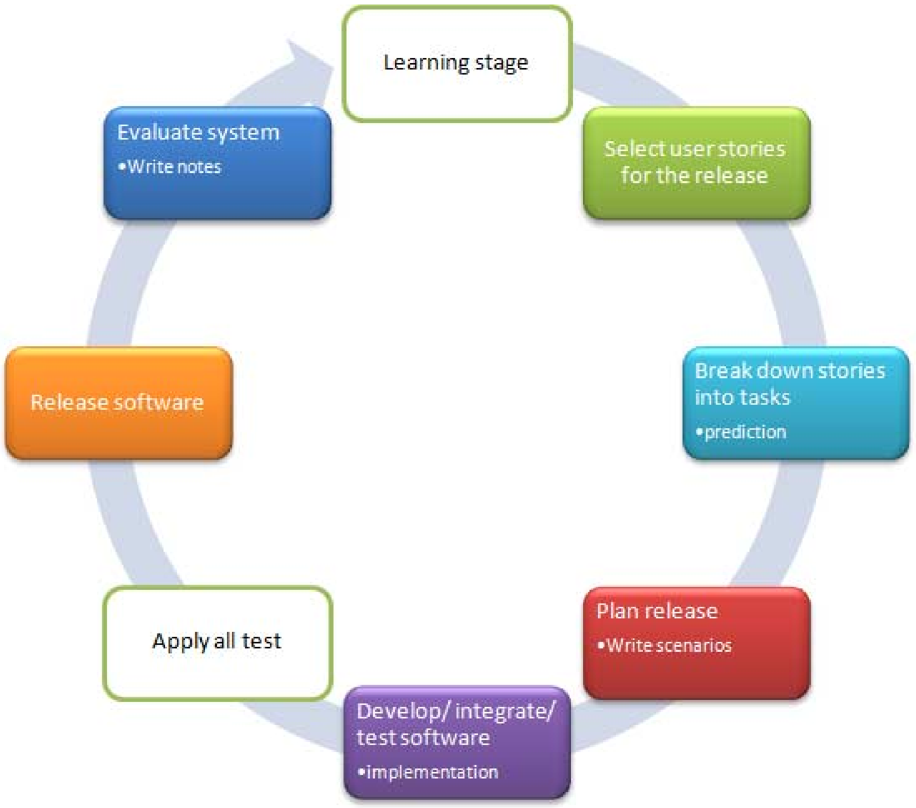


Figure 4.9 XP: Extreme Programming with security model

5. Experiment

Here in this chapter we will pass through real life example, and try to implement our security model, and explain in some points how to use this model, how to write scenarios and how to test our products.

We take an example of building Web application using ASP.net with SQL server; this application is social community web application that enables visitors to contact each other.

And here are some requirements of this application:

1- Users will have to register in order to create an account

2- After registration, activation of the email will be sent to users account

3- User will have to log in after they creating the account

4- Users can upload their photos, videos to the site

5- User can comment on their photos, videos or their friends.

6- Users can add other users as friends and also they can delete friends

If we look at these customer requirements we will finds that they are simple and common in web application, and here we will try to break security threat into expected vulnerabilities.

5.1 Learning Stage:

Here after reviewing and learning from the vulnerabilities sheet, we must take knowledge of the below related vulnerabilities, since they all related to the following area:

1- Web Application
2- ASP.net technologies
3- SQL server
4- Windows environment
5- Amateur users
6- Anonymous users

And we found that the below list is joining these areas:

- Addition of data-structure sentinel
- Allowing password aging
- ASP.NET Misconfigurations
- Business logic vulnerability
- Catch NullPointerException
- Comparing classes by name
- Cross Site Scripting Flaw
- Deserialization of untrusted data
- Empty Catch Block
- Empty String Password
- Failure of true random number generator
- Failure to add integrity check value
- Failure to drop privileges when reasonable
- Failure to encrypt data
- Failure to protect stored data from modification
- Failure to provide confidentiality for stored data
- Failure to validate host-specific certificate data
- Format String
- Hard-Coded Password
- Ignored function return value
- Injection problem
- Insecure Randomness

- Insecure Third Party Domain Access
- Insufficient Session-ID Length
- Log Forging
- Missing XML Validation
- Not allowing password aging
- Often Misused: Authentication
- Often Misused: Privilege Management
- Often Misused: String Management
- Open redirect
- Password Management: Weak Cryptography
- Password Plaintext Storage
- Privacy Violation
- Session Fixation
- Storing passwords in a recoverable format
- String Termination Error
- Truncation error
- Trust of system event data
- Trusting self-reported DNS name
- Trusting self-reported IP address
- Uncaught exception
- Unreleased Resource
- Unrestricted File Upload
- Using password systems
- Write-what-where condition

The list seems to be very long and it requires long time to be learned and that's the point we need, the target audience needs to know all the potential vulnerabilities before starting to work on the project, and this knowledge is not needed to be transferred every time the target audience will develop a new web application, and this will reflect the rest parts in the application so it becomes easier and more trusted every time it's checked.

And if we look at the list one more time we see that it is divided into all phases in the software life cycle, and let's say to the system analyst that there are 12 points, for developer there are 19 points, for deployed about 7 points … etc.

5.2 Predicting Stage

As a system analysis and system requirements collector, I will write the following predictions according to the requirements and to the above list:

1- Allowing password aging: since visitor will be able to log in using their own credentials.
2- Business logic: this is ways of using the legitimate processing flow of an application in a way that results in a negative consequence to the organization.
3- Empty String Password: this validation should be written in the system analysis documents.
4- Failure to add integrity check value: this validation should be written in the system analysis documents.
5- Failure to drop privileges when reasonable: this validation should be written in the system analysis documents.
6- Failure to provide confidentiality for stored data: this validation should be written in the system analysis documents.
7- Failure to validate host-specific certificate data: this validation should be written in the system analysis documents.
8- Hard-Coded Password: this validation should be written in the system analysis documents.
9- Insecure Third Party Domain Access: this validation should be written in the system analysis documents.
10- Log Forging: this validation should be written in the system analysis documents.
11- Often Misused: Authentication: this validation should be written in the system analysis documents.
12- Often Misused: Privilege Management: this validation should be written in the system analysis documents.
13- Password Management: Weak Cryptography: this validation should be written in the system analysis documents.
14- Password Plaintext Storage: this validation should be written in the system analysis documents.
15- Privacy Violation: this validation should be written in the system analysis documents.
16- Trusting self-reported DNS name: this validation should be written in the system analysis documents.

The above 16 points are taken from the 45 points in the learning stage where I predict these fail to occur so I write my system analysis and collect system requirements and take them all into consideration.

5.3 Write Scenario

Let’s take one example of security scenario on our web application and show how this list help system developer to check their software against security falls.

#	Name	Description	Scenario steps	Reference	Level
1	SQL Injection	User attempt to log in using other user accounts depends on miss apply to SQL query sent from web application	Try to offer a complete string in the password field to be concatenating with the passed query	implementation	Bravo
2	...	...	...	..	...

Table 5.1: Case study scenario

And after writing all scenarios this will be considered as part of the tests in the test phase,

5.4 Implementation

Here in this part we will skip it in order to avoid writing application in our thesis, but we just give knowledge of each risk.

5.5 Apply All Tests

After finishing implementation, we have to build a test sheet that contains all expected vulnerabilities and their counter attack scenario.

Test order	Test name	Test scenario	Test reference	Test time	Test result	Test note
1	SQL injection	Try to pass true condition value to the login page	Technologies (ASP.net)	00:00	Pass	Ok
2	Weak password	Try to update user password with the password (123456)	Implementation	00:12	System request to have solid password	-
3	-	-	-	-	-	-

Table 2.2 Apply all Tests

5.6 Documentation

Documentation here will include the following information as an example of the required entity for this case study

1- System description: Web application using ASP.net with SQL server, this application is social community web application that enables visitor to contact with other.

2- Description of used technology :

   a. ASP.net as presentation layer,

   b. SQL server as data layer,

   c. .net classes as business logic layer

3- Environment developed on:

   a. windows XP service pack 3 as hosted developed environment

   b. visual studio 2008 as development IDE

   c. SQL server 2005 express edition as database

   d. IIS 6

4- Environment deploy on:

   a. Windows server 2008

   b. IIS 7

   c. SQL server 2008

   d. .net framework 2,3,3.5,4 all enabled

5- Security vulnerability expected : all the 16 points in the prediction layer

6- Expected Security vulnerability blocker : all the points in the write scenario tables

7- Implementer notes on vulnerabilities : these points come with the implementation codes

8- Tester notes: the result table of applying all test phase.

6. Conclusion and Future Work

6.1 Conclusion

Based on my previous experience in software development, I found that most of the software lifecycle models and phases lack the necessary mechanisms for software security, and mainly focuses on the mechanisms to distribute authentication and authorization information which is far away from the real or genuine meaning of security.

In this work I have collected many security vulnerabilities that I found in trusted resources, illustrated and classified them in a clear way to create a model that can be used by software engineers and researchers to predict and resolve security issues during the software development lifecycle.

The proposed model is applied in 6 stages and starts with a learning stage; this division provides greater understanding of potential security vulnerabilities in all lifecycle stages and up to the level of authentication phase (IT auditors). We have followed a comprehensive approach to better serve the developer, by showing how to integrate the proposed model into the four common software lifecycles models by including an appropriate security phase in each model.

The first three stages of the model (Learning, Prediction, and Writing Scenarios) can be applies in the earlier stages of the software lifecycle such as requirements collection, analysis, and design stages. This distribution gives the developer the chance to learn about security vulnerabilities early in the lifecycle and allows him to take the required measurements to avoid any security risks.

The last three stages of the model (Implementation, Applying all tests, and Documentation), represent the natural place where the security scenarios are applied, and then test the required functionality of this software and the basic authentication and authorization for that software. Adding security test to the testing phase will enrich the testing phase and help testers in formatting their tests according to the provided forms. Documentation is also a very important phase in delivering the application and collecting information about what security auditor ask for and how to help security auditors do their testing.

We have included a simple application scenario that passes through each stage, and show how the model can applied in a real life application. The example shows what vulnerabilities we need to take care of, and how to write a scenario to counter attack each one of them, and last it shows how to document our work for delivery to the end user.

And after this example we arrive to the following conclusions:

1- Software security vulnerabilities occur due to lack of information about them.
2- Software engineering lifecycles should include security risk handling mechanisms in all phases not only as a risk analysis phase.
3- Work on security vulnerabilities based on concrete and complete knowledge minimizes the required time to counter attack and avoid serious damage scenarios.
4- Risk management for software security should be the last choice to secure our software systems—planning for security should start at the first stage of information collection.

5- Writing scenarios on counter attacks can be reused for other similar software pieces and this minimizes time.
6- Integrating this model into software lifecycle models is easy and manageable and does not require significant overhead.
7- The documentation to describe software security handling process is an important and integral part in delivering the software and testing it.

6.2 Future Works

Based on the proposed model, we can facilitate a better understanding and awareness about the many risks and vulnerabilities that can appear in software security. We hope that this model will provide better protection to software programs and their state of continuity as a whole.

To move forward in this area of research, I will try to achieve an intelligent working system based on the analysis data to offer a process-centric system for detecting all possible security vulnerabilities and providing context-sensitive mechanisms to avoid them. This can be achieved by doing a thorough examination of each security vulnerability in advance and according to the program requirements and then submitting a report on each vulnerability.

Most of security vulnerabilities come from the state of ignorance of relevant security principles, the lack of knowledge on this area, and the lack of interest on this issue. In addition, people tend to believe that the running environment or the firewall can secure their systems from attackers, ignoring the fact that these environments and firewalls themselves face a lot of security vulnerabilities.

6.3 Collected Vulnerabilities

Here in this section we provide the collected vulnerabilities with their description, software stage and vulnerabilities emergency level for each one of them.

Access control enforced by presentation layer	Enforcing access control in the presentation layer means that the developer does not show buttons and links for functions and assets that are not authorized for the user. An attacker, however, is not constrained by the buttons nd links presented, and can forge requests for those functions and assets. Forced browsing is one attack that targets this type of vulnerability.	Implementation	Bravo
Addition of data-structure sentinel	The accidental addition of a data-structure sentinel can cause serious programming logic problems.	Design	Alpha
Allowing password aging	Allowing password aging to occur unchecked can result in the possibility of diminished password integrity.	Implementation	Echo
ASP.NET Misconfigurations	Debugging messages help attackers learn about the system and plan a form of attack	Design	Alpha
Assigning instead of comparing	In many languages, the compare statement is very close in appearance to the assignment statement and are often confused	Implementation	Alpha
Authentication Bypass via Assumed-Immutable Data	Assumed-immutable authentication data can be modified by attackers to bypass the authentication. Most of the time, this vulnerability results from inappropriate session management, i.e., important data that is used for authentication decisions is sent to the client side and subject to user modification. This kind of data should be stored in the server-side session as much as possible.	Implementation	Alpha
Buffer Overflow	Buffer overflow is probably the best known form of software security vulnerability. Most software developers know what a buffer overflow vulnerability is, but buffer overflow attacks against both legacy and newly-developed applications are still quite common. Part of the problem is due to the wide variety of ways buffer overflows can occur, and part is due to the error-prone techniques often used to prevent them.	Implementation	Bravo

Buffer underwrite	A buffer underwrite condition occurs when a buffer is indexed with a negative number, or pointer arithmetic with a negative value results in a position before the beginning of the valid memory location.	Implementation	Alpha
Business logic vulnerability	Most security problems are weaknesses in an application that result from a broken or missing security control (authentication, access control, input validation, etc...). By contrast, business logic vulnerabilities are ways of using the legitimate processing flow of an application in a way that results in a negative consequence to the organization.	Design	Bravo
Capture-replay	A capture-relay protocol flaw exists when it is possible for a malicious user to sniff network traffic and replay it to the server in question to the same effect as the original message (or with minor changes).	Installation and deployment(deployment)	Alpha
Catch NullPointerException	It is generally a bad practice to catch NullPointerException.	Implementation	Alpha
Comparing classes by name	The practice of determining an object's type, based on its name, is dangerous since malicious code may purposely reuse class names in order to appear trusted.	Implementation	Alpha
Comparing instead of assigning		Implementation	Echo
Comprehensive list of Threats to Authentication Procedures and Data		Implementation	Bravo
Covert timing channel	Unintended information about data gets leaked through observing the timing of events.	Installation and deployment(deployment)	Bravo
CRLF Injection	The term CRLF refers to Carriage Return (ASCII 13, \r) Line Feed (ASCII 10, \n). They're used to note the termination of a line, however, dealt with differently in today's popular Operating Systems. For example: in Windows both a CR and LF are required to note the end of a line, whereas in Linux/UNIX a LF is only required.	Implementation	Charlie

Cross Site Scripting Flaw	Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.	Implementation	Charlie
Dangerous Function	Functions that cannot be used safely should never be used.Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account.	Testing and debugging (validation)	Bravo
Deletion of data-structure sentinel	The accidental deletion of a data structure sentinel can cause serious programing logic problems.	Implementation	Bravo
Deserialization of untrusted data	Data which is untrusted cannot be trusted to be well formed.	Design	Alpha
Directory Restriction Error	Improper use of the chroot() system call may allow attackers to escape a chroot jail.The application fails to enforce the intended restricted directory access policy. By using relative paths or other path traversal attack mechanisms, an attacker can access unauthorized files outside the restricted directory.	Installation and deployment(deployment)	Echo
Double Free	Double free errors occur when free() is called more than once with the same memory address as an argument.Calling free() twice on the same value can lead to a buffer overflow. When a program calls free() twice with the same argument, the program's memory management data structures become corrupted. This corruption can cause the program to crash or, in some circumstances, cause two later calls to malloc() to return the same pointer. If malloc() returns the same value twice and the program later gives the attacker control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack.	Implementation	Echo
Doubly freeing memory	Freeing or deleting the same memory chunk twice may - when combined with other flaws - result in a write-what-where condition.	Implementation	Echo
Duplicate key in associative list	Associative lists should always have unique keys, since having non-unique keys can often be	Implementation	Echo

(alist)	mistaken for an error.		
Empty Catch Block	Ignoring an exception can cause the program to overlook unexpected states and conditions.When an exception is thrown and not caught, the process has given up an opportunity to decide if a given failure or event is worth a change in execution.Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break.Two dubious assumptions that are easy to spot in code are "this method call can never fail" and "it doesn't matter if this call fails". When a programmer ignores an exception, they implicitly state that they are operating under one of these assumptions.	Implementation	Alpha
Empty String Password	Using an empty string as a password is insecure.It is never appropriate to use an empty string as a password. It is too easy to guess. Empty string password makes the authentication as weak as the user names, which are normally public or guessable. This make a brute-force attack against the login interface much easier.	Design	Bravo
Failure of true random number generator	True random number generators generally have a limited source of entropy and therefore can fail or block.	Implementation	Echo
Failure to account for default case in switch	The failure to account for the default case in switch statements may lead to complex logical errors and may aid in other, unexpected security-related conditions.	Implementation	Alpha
Failure to add integrity check value	If integrity check values or "checksums" are omitted from a protocol, there is no way of determining if data has been corrupted in transmission.	Implementation	Echo
Failure to check for certificate revocation	If a certificate is used without first checking to ensure it was not revoked, the certificate may be compromised.	Implementation	Alpha
Failure to check integrity check value	If integrity check values or "checksums" are not validated before messages are parsed and used, there is no way of determining if data has been corrupted in transmission.	Implementation	Echo

Failure to check whether privileges were dropped successfully	If one changes security privileges, one should ensure that the change was successful.	Implementation	Alpha
Failure to deallocate data	If memory is allocated and not freed the process could continue to consume more and more memory and eventually crash.	Implementation	Echo
Failure to drop privileges when reasonable	Failing to drop privileges when it is reasonable to do so results in a lengthened time during which exploitation may result in unnecessarily negative consequences.	Design	Echo
Failure to encrypt data	The failure to encrypt data passes up the guarantees of confidentiality, integrity, and accountability that properly implemented encryption conveys.	Design	Bravo
Failure to follow chain of trust in certificate validation	Failure to follow the chain of trust when validating a certificate results in the trust of a given resource which has no connection to trusted root-certificate entities.	Design	Alpha
Failure to follow guideline/specification		Implementation	Bravo
Failure to protect stored data from modification	Data should be protected from direct modification.	Design	Bravo
Failure to provide confidentiality for stored data	Non-final public fields should be avoided, if possible, as the code is easily tamperable.	Design	Bravo
Failure to validate certificate expiration	The failure to validate certificate operation may result in trust being assigned to certificates which have been abandoned due to age.	Implementation	Bravo
Failure to validate host-specific certificate data	The failure to validate host-specific certificate data may mean that, while the certificate read was valid, it was not for the site originally requested.	Implementation	Echo
File Access Race Condition: TOCTOU	The window of time between when a file property is checked and when the file is used can be exploited to launch a privilege escalation attack.	Implementation	Alpha

Format String	Allowing an attacker to control a function's format string may result in a buffer overflow.Format string vulnerabilities occur when:Data enters the application from an untrusted source.The data is passed as the format string argument to a function like sprintf(), FormatMessageW(), or syslog().Format string problems occur when a user has the ability to control or write completely the format string used to format data in the printf style family of C/C++ functions.	Implementation	Alpha
Guessed or visible temporary file	On some operating systems, the fact that the temp file exists may be apparent to any user.	Implementation	Bravo
Hard-Coded Password	A hard-coded password vulnerability occurs when usernames and passwords are included in HTML comments. Because HTML comments are not displayed, it was often the mentality that normal users would not see them. It can also occur when a specific username (usually unique) does not require a password.	Design	Charlie
Heap Inspection	Do not use realloc() to resize buffers that store sensitive information.Heap inspection vulnerabilities occur when sensitive data, such as a password or an encryption key, can be exposed to an attacker because they are not removed from memory.The realloc() function is commonly used to increase the size of a block of allocated memory. This operation often requires copying the contents of the old memory block into a new and larger block. This operation leaves the contents of the original block intact but inaccessible to the program, preventing the program from being able to scrub sensitive data from memory. If an attacker can later examine the contents of a memory dump, the sensitive data could be exposed.	Implementation	Bravo
Heap overflow	A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as the POSIX malloc() call.	Implementation	Alpha

Ignored function return value	If a functions return value is not checked, it could have failed without any warning.Ignoring a method's return value can cause the program to overlook unexpected states and conditions. Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break.Two dubious assumptions that are easy to spot in code are "this function call can never fail" and "it doesn't matter if this function call fails". When a programmer ignores the return value from a function, they implicitly state that they are operating under one of these assumptions.	Implementation	Alpha
Illegal Pointer Value	This function can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer may have unintended consequences.This function can return a pointer to memory outside the bounds of the buffer to be searched under either of the following circumstances:An attacker can control the contents of the buffer to be searched,An attacker can control the value for which to search	Implementation	Alpha
Improper cleanup on thrown exception	Causing a change in flow, due to an exception, can often leave the code in a bad state.	Implementation	Alpha
Improper Data Validation		Implementation	Alpha
Improper error handling	Sometimes an error is detected, and bad or no action is taken.	Implementation	Echo
Improper string length checking	Improper string length checking takes place when wide or multi-byte character strings are mistaken for standard character strings.	Implementation	Echo
Improper temp file opening	Tempfile creation should be done in a safe way. To be safe, the temp file function should open up the temp file with appropriate access control. The temp file function should also retain this quality, while being resistant to race conditions.	Implementation	Alpha
Incorrect block delimitation	In some languages, forgetting to explicitly delimit a block can result in a logic error that can, in turn, have security implications.	Implementation	Alpha

Information Leakage	Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack. An information leak occurs when system data or debugging information leaves the program through an output stream or logging function.	Requirements specification (AKA Verification)	Alpha
Information leak through class cloning	Cloneable classes are effectively open classes since data cannot be hidden in them.	Implementation	Alpha
Information leak through serialization	Serializable classes are effectively open classes since data cannot be hidden in them.	Implementation	Alpha
Injection problem	Injection problems span a wide range of instantiations. The basic form of this flaw involves the injection of control-plane data into the data-plane in order to alter the control flow of the process.	Implementation	Bravo
Insecure Compiler Optimization	Improperly scrubbing sensitive data from memory can compromise security. Compiler optimization errors occur when: Secret data is stored in memory. The secret data is scrubbed from memory by overwriting its contents. The source code is compiled using an optimizing compiler, which identifies and removes the function that overwrites the contents as a dead store because the memory is not used subsequently.	Maintenance	Alpha

Insecure Randomness	Standard pseudo-random number generators cannot withstand cryptographic attacks. Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in security-sensitive context. Computers are deterministic machines, and as such are unable to produce true randomness. Pseudo-Random Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated. There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts.	Implementation	Bravo
Insecure Temporary File	Creating and using insecure temporary files can leave application and system data vulnerable to attacks. Applications require temporary files so frequently that many different mechanisms exist for creating them in the C Library and Windows® API. Most of these functions are vulnerable to various forms of attacks.	Implementation	Alpha
Insecure Third Party Domain Access	Occurs when an application contains content provided from a 3rd party resource that is delivered without any type of content scrub. Environments Affected Web servers Application servers Client Machines	Integration	

Insecure Transport	The application configuration should ensure that SSL is used for all access controlled pages. If an application uses SSL to guarantee confidential communication with client browsers, the application configuration should make it impossible to view any access controlled page without SSL. However, it is not an uncommon problem that the configuration of the application fails to enforce the use of SSL on pages that contain sensitive data. There are three common ways for SSL to be bypassed: A user manually enters the URL and types "HTTP" rather than "HTTPS". Attackers intentionally send a user to an insecure URL. A programmer erroneously creates a relative link to a page in the application, failing to switch from HTTP to HTTPS. (This is particularly easy to do when the link moves between public and secured areas on a web site.)	Installation and deployment(deployment)	Bravo
Insufficient Entropy	When an undesirably low amount of entropy is available. Psuedo Random Number Generators are susceptible to suffering from insufficient entropy when they are initialized, because entropy data may not be available to them yet.	Implementation	Alpha
Insufficient entropy in pseudo-random number generator	The lack of entropy available for, or used by, a PRNG can be a stability and security threat.	Implementation	Alpha
Insufficient Session-ID Length	Session identifiers should be at least 128 bits long to prevent brute-force session guessing attacks.	Implementation	Alpha
Integer coercion error	Integer coercion refers to a set of flaws pertaining to the type casting, extension, or truncation of primitive data types.	Implementation	Alpha
Integer overflow	An integer overflow condition exists when an integer, which has not been properly sanity checked, is used in the determination of an offset or size for memory allocation, copying, concatenation, or similarly. If the integer in question is incremented past the maximum possible value, it may wrap to become a very small, or negative number, therefore providing a very incorrect value.	Implementation	Alpha
Invoking untrusted mobile code	This process will download external source or binaries and execute it.	Implementation	Alpha

J2EE Misconfiguration: Unsafe Bean Declaration	Entity beans that expose a remote interface become part of an application's attack surface. For performance reasons, an application should rarely use remote entity beans, so there is a good chance that a remote entity bean declaration is an error.	Implementation	Alpha
Key exchange without entity authentication	Performing a key exchange without verifying the identity of the entity being communicated with will preserve the integrity of the information sent between the two entities; this will not, however, guarantee the identity of end entity.	Implementation	Alpha
Least Privilege Violation	The elevated privilege level required to perform operations such as chroot() should be dropped immediately after the operation is performed.When a program calls a privileged function, such as chroot(), it must first acquire root privilege. As soon as the privileged operation has completed, the program should drop root privilege and return to the privilege level of the invoking user.	Implementation	Alpha
Leftover Debug Code	Debug code can create unintended entry points in a deployed web application. A common development practice is to add "back door" code specifically designed for debugging or testing purposes that is not intended to be shipped or deployed with the application. When this sort of debug code is accidentally left in the application, the application is open to unintended modes of interaction. These back door entry points create security risks because they are not considered during design or testing and fall outside of the expected operating conditions of the application.	Implementation	Alpha

Log Forging	Writing unvalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs. Log forging vulnerabilities occur when: Data enters an application from an untrusted source. The data is written to an application or system log file. Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information. Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act [1]. In the worst case, an attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility .	Implementation	Echo
Log injection	Log injection problems are a subset of injection problem, in which invalid entries taken from user input are inserted in logs or audit trails, allowing an attacker to mislead administrators or cover traces of attack. Log injection can also sometimes be used to attack log monitoring systems indirectly by injecting data that monitoring systems will misinterpret.	Implementation	Alpha

Member Field Race Condition	Servlet member fields may allow one user to see another user's data. Many Servlet developers do not understand that, unless a Servlet implements the SingleThreadModel interface, the Servlet is a singleton; there is only one instance of the Servlet, and that single instance is used and re-used to handle multiple requests that are processed simultaneously by different threads. A common result of this misunderstanding is that developers use Servlet member fields in such a way that one user may inadvertently see another user's data. In other words, storing user data in Servlet member fields introduces a data access race condition.	Implementation	Alpha
Memory leak	A memory leak is an unintentional form of memory consumption whereby the developer fails to free an allocated block of memory when no longer needed	Implementation	Echo
Miscalculated null termination	Miscalculated null termination occurs when the placement of a null character at the end of a buffer of characters (or string) is misplaced or omitted.	Implementation	Alpha
Misinterpreted function return value	If a function's return value is not properly checked, the function could have failed without proper acknowledgement.	Implementation	Alpha

Missing Error Handling	A web application must define a default error page for 404 errors, 500 errors, and to catch java.lang. Throwable exceptions prevent attackers from mining information from the application container's built-in error response. When an attacker explores a web site looking for vulnerabilities, the amount of information that the site provides is crucial to the eventual success or failure of any attempted attacks. If the application shows the attacker a stack trace, it relinquishes information that makes the attacker's job significantly easier. For example, a stack trace might show the attacker a malformed SQL query string, the type of database being used, and the version of the application container. This information enables the attacker to target known vulnerabilities in these components. The application configuration should specify a default error page in order to guarantee that the application will never leak error messages to an attacker. Handling standard HTTP error codes is useful and user-friendly in addition to being a good security practice, and a good configuration will also define a last-chance error handler that catches any exception that could possibly be thrown by the application.	Implementation	Alpha
Missing parameter	If too few arguments are sent to a function, the function will still pop the expected number of arguments from the stack. Potentially, a variable number of arguments could be exhausted in a function as well.	Implementation	Alpha

Missing XML Validation	Failure to enable validation when parsing XML gives an attacker the opportunity to supply malicious input. Most successful attacks begin with a violation of the programmer's assumptions. By accepting an XML document without validating it against a DTD or XML schema, the programmer leaves a door open for attackers to provide unexpected, unreasonable, or malicious input. It is not possible for an XML parser to validate all aspects of a document's content; a parser cannot understand the complete semantics of the data. However, a parser can do a complete and thorough job of checking the document's structure and therefore guarantee to the code that processes the document that the content is well-formed.	Implementation	Alpha
Mutable object returned	Sending non-cloned mutable data as a return value may result in that data being altered or deleted by the called function, thereby putting the class in an undefined state.	Implementation	Alpha
Non-cryptographic pseudo-random number generator	The use of Non-cryptographic Pseudo-Random Number Generators (PRNGs) as a source for security can be very dangerous, since they are predictable.	Implementation	Bravo
Not allowing password aging	If no mechanism is in place for managing password aging, users will have no incentive to update passwords in a timely manner.	Implementation	Alpha
Not using a random initialization vector with cipher block chaining mode	Not using a random initialization vector with Cipher Block Chaining (CBC) Mode causes algorithms to be susceptible to dictionary attacks.	Implementation	Alpha

Null Dereference	The program can potentially dereference a null pointer, thereby raising a NullPointerException. Null pointer errors are usually the result of one or more programmer assumptions being violated. Most null pointer issues result in general software reliability problems, but if an attacker can intentionally trigger a null pointer dereference, the attacker might be able to use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks. A null-pointer dereference takes place when a pointer with a value of NULL is used as though it pointed to a valid memory area.	Implementation	Alpha
Object Model Violation: Just One of equals() and hashCode() Defined	This class overrides only one of equals() and hashCode(). Java objects are expected to obey a number of invariants related to equality. One of these invariants is that equal objects must have equal hashcodes. In other words, if a.equals(b) == true then a.hashCode() == b.hashCode(). Failure to uphold this invariant is likely to cause trouble if objects of this class are stored in a collection. If the objects of the class in question are used as a key in a Hashtable or if they are inserted into a Map or Set, it is critical that equal objects have equal hashcodes.	Implementation	Alpha
Often Misused: Authentication	Attackers can spoof DNS entries. Do not rely on DNS names for security. Many DNS servers are susceptible to spoofing attacks, so you should assume that your software will someday run in an environment with a compromised DNS server. If attackers are allowed to make DNS updates (sometimes called DNS cache poisoning), they can route your network traffic through their machines or make it appear as if their IP addresses are part of your domain. Do not base the security of your system on DNS names.	Implementation	Bravo

Often Misused: Exception Handling	The _alloca() function can throw a stack overflow exception, potentially causing the program to crash. The _alloca() function allocates memory on the stack. If an allocation request is too large for the available stack space, _alloca() throws an exception. If the exception is not caught, the program will crash, potentially enabling a denial of service attack. _alloca() has been deprecated as of Microsoft Visual Studio 2005®. It has been replaced with the more secure _alloca_s().	Implementation	Alpha
Often Misused: File System	Passing an inadequately-sized output buffer to a path manipulation function can result in a buffer overflow. Windows provides a large number of utility functions that manipulate buffers containing filenames. In most cases, the result is returned in a buffer that is passed in as input. (Usually the filename is modified in place.) Most functions require the buffer to be at least MAX_PATH bytes in length, but you should check the documentation for each function individually. If the buffer is not large enough to store the result of the manipulation, a buffer overflow can occur.	Implementation	Bravo
Often Misused: Privilege Management	Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities. Programs that run with root privileges have caused innumerable Unix security disasters. It is imperative that you carefully review privileged programs for all kinds of security problems, but it is equally important that privileged programs drop back to an unprivileged state as quickly as possible in order to limit the amount of damage that an overlooked vulnerability might be able to cause. Privilege management functions can behave in some less-than-obvious ways, and they have different quirks on different platforms. These inconsistencies are particularly pronounced if you are transitioning from one non-root user to another. Signal handlers and spawned processes run at the privilege of the owning process, so if a process is running as root when a signal fires or a sub-process is executed, the signal handler or sub-process will operate with root privileges. An attacker may be able to leverage these elevated privileges to do further damage.	Implementation	Alpha

Often Misused: String Management	Functions that convert between Multibyte and Unicode strings encourage buffer overflows. Windows provides the MultiByteToWideChar(), WideCharToMultiByte(), UnicodeToBytes, and BytesToUnicode functions to convert between arbitrary multibyte (usually ANSI) character strings and Unicode (wide character) strings. The size arguments to these functions are specified in different units – one in bytes, the other in characters – making their use prone to error. In a multibyte character string, each character occupies a varying number of bytes, and therefore the size of such strings is most easily specified as a total number of bytes. In Unicode, however, characters are always a fixed size, and string lengths are typically given by the number of characters they contain. Mistakenly specifying the wrong units in a size argument can lead to a buffer overflow.	Implementation	Alpha
Omitted break statement	Omitting a break statement so that one may fall through is often indistinguishable from an error, and therefore should not be used.	Implementation	Alpha
Open forward	An open forward is an application that takes a parameter and forwards a user to another part of the application without any validation or access control checks. This may allow an attacker to bypass access control checks, especially those enforced externally, such as by a web server.	Implementation	Alpha
Open redirect	An open redirect is an application that takes a parameter and redirects a user to the parameter value without any validation. This vulnerability is used in phishing attacks to get users to visit malicious sites without realizing it.	Implementation	Alpha
Overflow of static internal buffer	A non-final static field can be viewed and edited in dangerous ways.	Implementation	Echo

Overly-Broad Catch Block	The catch block handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program. Multiple catch blocks can get ugly and repetitive, but "condensing" catch blocks by catching a high-level class like Exception can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. Catching an overly broad exception essentially defeats the purpose of Java's typed exceptions, and can become particularly dangerous if the program grows and begins to throw new types of exceptions. The new exception types will not receive any attention.	Implementation	Alpha
Overly-Broad Throws Declaration	The method throws a generic exception making it harder for callers to do a good job of error handling and recovery. Declaring a method to throw Exception or Throwable makes it difficult for callers to do good error handling and error recovery. Java's exception mechanism is set up to make it easy for callers to anticipate what can go wrong and write code to handle each specific exceptional circumstance. Declaring that a method throws a generic form of exception defeats this system.	Implementation	Alpha
Passing mutable objects to an untrusted method	Sending non-cloned mutable data as an argument may result in that data being altered or deleted by the called function, thereby putting the calling function into an undefined state.	Implementation	Echo
Password Management: Hardcoded Password	Hardcoded passwords may compromise system security in a way that cannot be easily remedied. It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.	Implementation	Bravo

Password Management: Weak Cryptography	Obscuring a password with a trivial encoding does not protect the password. Password management issues occur when a password is stored in plaintext in an application's properties or configuration file. A programmer can attempt to remedy the password management problem by obscuring the password with an encoding function, such as base 64 encoding, but this effort does not adequately protect the password.	Implementation	Bravo
Password Plaintext Storage	Storing a password in plaintext may result in a system compromise. Password management issues occur when a password is stored in plaintext in an application's properties or configuration file. A programmer can attempt to remedy the password management problem by obscuring the password with an encoding function, such as base 64 encoding, but this effort does not adequately protect the password. Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plaintext.	Implementation	Bravo
PHP File Inclusion	PHP, as many other languages, allows the inclusion of files in order to provide or extend the functionality of the current file.	Implementation	Bravo
Poor Logging Practice		Implementation	Echo
Portability Flaw	Functions with inconsistent implementations across operating systems and operating system versions cause portability problems. The behavior of functions in this category varies by operating system, and at times, even by operating system version. Implementation differences can include: Slight differences in the way parameters are interpreted, leading to inconsistent results. Some implementations of the function carry significant security risks. The function might not be defined on all platforms.	Implementation	Alpha

Privacy Violation	Mishandling private information, such as customer passwords or social security numbers, can compromise user privacy, and is often illegal.	Implementation	Bravo
PRNG Seed Error	The incorrect use of a seed by a Psuedo Random Number Generator. A seed error is usually brought on through the erroneous generation or application of a seed state.	Implementation	Alpha
Process Control	Executing commands from an untrusted source or in an untrusted environment can cause an application to execute malicious commands on behalf of an attacker. Process control vulnerabilities take two forms: An attacker can change the command that the program executes: the attacker explicitly controls what the command is. An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means. We will first consider the first scenario, the possibility that an attacker may be able to control the command that is executed. Process control vulnerabilities of this type occur when: Data enters the application from an untrusted source. The data is used as or as part of a string representing a command that is executed by the application. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.	Implementation	Bravo
Publicizing of private data when using inner classes	Java byte code has no notion of an inner class; therefore inner classes provide only a package-level security mechanism. Furthermore, the inner class gets access to the fields of its outer class even if that class is declared private.	Implementation	Alpha
Race Conditions	A race condition occurs when a pair of routine programming calls in an application do not perform in the sequential manner that was intended per business rules. It is a timing event within software that can become a security vulnerability if the calls are not performed in the correct order.	Implementation	Alpha
Reflection attack in an auth protocol	Simple authentication protocols are subject to reflection attacks if a malicious user can use the target machine to impersonate a trusted user.	Implementation	Alpha

Reflection injection	Reflection injection problems are a subset of injection problems, in which external input is used to construct a string value passed to class reflection APIs. By manipulating the value an attacker can cause unexpected classes to be loaded, or change what method or fields are accessed on an object.	Implementation	Alpha
Relative path library search	Certain functions perform automatic path searching. The method and results of this path searching may not be as expected. Example: WinExec will use the space character as a delimiter, finding "C:\Program.exe" as an acceptable result for a search for "C:\Program Files\Foo\Bar.exe".	Implementation	Alpha
Reliance on data layout	Assumptions about protocol data or data stored in memory can be invalid, resulting in using data in ways that were unintended.	Implementation	Echo
Relying on package-level scope	Java packages are not inherently closed; therefore, relying on them for code security is not a good practice.	Implementation	Alpha
Resource exhaustion	Resource exhaustion is a simple denial of service condition which occurs when the resources necessary to perform an action are entirely consumed, therefore preventing that action from taking place.	Implementation	Alpha
Return Inside Finally Block	Returning from inside a finally block will cause exceptions to be lost. A return statement inside a finally block will cause any exception that might be thrown in the try block to be discarded.	Implementation	Alpha
Reusing a nonce, key pair in encryption	Nonces should be used for the present occasion and only once.	Implementation	Alpha

Session_Fixation	Authenticating a user without invalidating any existing session identifier gives an attacker the opportunity to steal authenticated sessions. Session fixation vulnerabilities occur when: A web application authenticates a user without first invalidating the existing session ID, thereby continuing to use the session ID already associated with the user. An attacker is able to force a known session ID on a user so that, once the user authenticates, the attacker has access to the authenticated session. In the generic exploit of session fixation vulnerabilities, an attacker creates a new session on a web application and records the associated session identifier. The attacker then causes the victim to authenticate against the server using the same session identifier, giving the attacker access to the user's account through the active session.	Implementation	Alpha
Sign extension error	If one extends a signed number incorrectly, if negative numbers are used, an incorrect extension may result.	Implementation	Alpha
Signed to unsigned conversion error	A signed-to-unsigned conversion error takes place when a signed primitive is used as an unsigned value, usually as a size variable.	Implementation	Alpha
Stack overflow	A stack overflow condition is a buffer overflow condition, where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).	Implementation	Alpha
State synchronization error	State synchronization refers to a set of flaws involving contradictory states of execution in a process which result in undefined behavior.	Implementation	Alpha
Storing passwords in a recoverable format	The storage of passwords in a recoverable format makes them subject to password reuse attacks by malicious users. If a system administrator can recover the password directly, or use a brute force search on the information available to him, he can use the password on other accounts.	Implementation	Bravo
String Termination Error	Relying on proper string termination may result in a buffer overflow. String termination errors occur when: Data enters a program via a function that does not null terminate its output. The data is passed to a function that requires its input to be null terminated.	Implementation	Alpha

Symbolic name not mapping to correct object	A constant symbolic reference to an object is used, even though the underlying object changes over time.	Implementation	Alpha
Truncation error	Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion.	Implementation	Alpha
Trust Boundary Violation	Commingling trusted and untrusted data in the same data structure encourages programmers to mistakenly trust unvalidated data. A trust boundary can be thought of as line drawn through a program. On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy. The purpose of validation logic is to allow data to safely cross the trust boundary--to move from untrusted to trusted. A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted. The most common way to make this mistake is to allow trusted and untrusted data to commingle in the same data structure.	Implementation	Alpha
Trust of system event data	Security based on event locations are insecure and can be spoofed.	Implementation	Alpha
Trusting self-reported DNS name	The use of self-reported DNS names as authentication is flawed and can easily be spoofed by malicious users.	Implementation	Alpha
Trusting self-reported IP address	The use of IP addresses as authentication is flawed and can easily be spoofed by malicious users.	Implementation	Alpha
Uncaught exception	Ignoring an exception can cause the program to overlook unexpected states and conditions. When an exception is thrown and not caught, the process has given up an opportunity to decide if a given failure or event is worth a change in execution. Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break. Two dubious assumptions that are easy to spot in code are "this method call can never fail" and "it doesn't matter if this call fails". When a programmer ignores an exception, they implicitly state that they are operating under one of these	Implementation	Alpha

	assumptions.		
Unchecked array indexing	Unchecked array indexing occurs when an unchecked value is used as an index into a buffer.	Implementation	Echo
Unchecked Return Value: Missing Check against Null	Ignoring a method's return value can cause the program to overlook unexpected states and conditions. Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break. Two dubious assumptions that are easy to spot in code are "this function call can never fail" and "it doesn't matter if this function call fails". When a programmer ignores the return value from a function, they implicitly state that they are operating under one of these assumptions.	Implementation	Alpha
Undefined Behavior	The behavior of this function is undefined unless its control parameter is set to a specific value. The Linux Standard Base Specification 2.0.1 for libc places constraints on the arguments to some internal functions [1]. If the constraints are not met, the behavior of the functions is not defined.	Implementation	Alpha
Uninitialized Variable	Using the value of an unitialized variable is not safe.	Implementation	Bravo
Unintentional pointer scaling	In C and C++, one may accidentally refer to the wrong memory due to the semantics of when math operations are implicitly scaled.	Implementation	Alpha
Unreleased Resource	The program can potentially fail to release a system resource. Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service attack by depleting the resource pool. Resource leaks have at least two common causes: Error conditions and other exceptional circumstances. Confusion over which part of the program is responsible for releasing the resource.	Implementation	Echo

Unrestricted File Upload	Uploaded files represent a significant risk to applications. The first step in many attacks is to get some code to the system to be attacked. Then the attack only needs to find a way to get the code executed. Using a file upload helps the attacker accomplish the first step. The consequences of unrestricted file upload can vary, including complete system takeover, an overloaded file system, forwarding attacks to backend systems, and simple defacement. It depends on what the application does with the uploaded file, including where it is stored. There are really two different classes of problems here. The first is with the file metadata, like the path and filename. These are generally provided by the transport, such as HTTP multipart encoding. This data may trick the application into overwriting a critical file or storing the file in a bad location. You must validate the metadata extremely carefully before using it. The other class of problem is with the file content. The range of problems here depends entirely on what the file is used for. See the examples below for some ideas about how files might be misused. To protect against this type of attack, you should analyze everything your application does with files and think carefully about what processing and interpreters are involved.	Implementation	Bravo
Unsafe function call from a signal handler	There are several functions which - under certain circumstances, if used in a signal handler - may result in the corruption of memory, allowing for exploitation of the process.	Implementation	Bravo
Unsafe JNI	Improper use of the Java Native Interface (JNI) can render Java applications vulnerable to security flaws in other languages. Unsafe JNI errors occur when a Java application uses JNI to call code written in another programming language.	Implementation	

Unsafe Mobile Code	Mobile code, such as a Java Applet, is code that is transmitted across a network and executed on a remote machine. Because mobile code developers have little if any control of the environment in which their code will execute, special security concerns become relevant. One of the biggest environmental threats results from the risk that the mobile code will run side-by-side with other, potentially malicious, mobile code. Because all of the popular web browsers execute code from multiple sources together in the same JVM, many of the security guidelines for mobile code are focused on preventing manipulation of your objects' state and behavior by adversaries who have access to the same virtual machine where your program is running.	Implementation	Alpha
Unsafe Reflection	An attacker may be able to create unexpected control flow paths through the application, potentially bypassing security checks. If an attacker can supply values that the application then uses to determine which class to instantiate or which method to invoke, the potential exists for the attacker to create control flow paths through the application that were not intended by the application developers. This attack vector may allow the attacker to bypass authentication or access control checks or otherwise cause the application to behave in an unexpected manner. This situation becomes a doomsday scenario if the attacker can upload files into a location that appears on the application's classpath or add new entries to the application's classpath. Under either of these conditions, the attacker can use reflection to introduce new, presumably malicious, behavior into the application.	Implementation	Bravo
Unsigned to signed conversion error	An unsigned-to-signed conversion error takes place when a large unsigned primitive is used as an signed value - usually as a size variable.	Implementation	
Use of hard-coded password	The use of a hard-coded password increases the possibility of password guessing tremendously.	Implementation	Alpha

Use of Obsolete Methods	The use of deprecated or obsolete functions may indicate neglected code. As programming languages evolve, functions occasionally become obsolete due to: Advances in the language Improved understanding of how operations should be performed effectively and securely Changes in the conventions that govern certain operations Functions that are removed are usually replaced by newer counterparts that perform the same task in some different and hopefully improved way. Refer to the documentation for this function in order to determine why it is deprecated or obsolete and to learn about alternative ways to achieve the same functionality. The remainder of this text discusses general problems that stem from the use of deprecated or obsolete functions.	Implementation	Echo
Use of sizeof() on a pointer type	Running sizeof() on a malloced pointer type will always return the wordsize/8.	Implementation	Echo
Using a broken or risky cryptographic algorithm	Attempting to create non-standard and non-tested algorithms, using weak algorithms, or applying algorithms incorrectly will pose a high weakness to data that is meant to be secure.	Implementation	Alpha
Using a key past its expiration date	The use of a cryptographic key or password past its expiration date diminishes its safety significantly.	Implementation	Alpha

Using freed memory	Referencing memory after it has been freed can cause a program to crash. The use of heap allocated memory after it has been freed or deleted leads to undefined system behavior and, in many cases, to a write-what-where condition. Use after free errors occur when a program continues to use a pointer after it has been freed. Like double free errors and memory leaks, use after free errors have two common and sometimes overlapping causes: Error conditions and other exceptional circumstances Confusion over which part of the program is responsible for freeing the memory Use after free errors sometimes have no effect and other times cause a program to crash. While it is technically feasible for the freed memory to be re-allocated and for an attacker to use this reallocation to launch a buffer overflow attack, we are unaware of any exploits based on this type of attack.	Implementation	Alpha
Using password systems	The use of password systems as the primary means of authentication may be subject to several flaws or shortcomings, each reducing the effectiveness of the mechanism.	Implementation	Delta
Using referer field for authentication or authorization	The referrer field (actually spelled 'referer') in HTTP requests can be easily modified and, as such, is not a valid means of message integrity checking.	Implementation	Bravo
Using single-factor authentication	The use of single-factor authentication can lead to unnecessary risk of compromise when compared with the benefits of a dual-factor authentication scheme.	Implementation	Echo
Using the wrong operator	This is a common error given when an operator is used which does not make sense in context.	Implementation	Bravo
Validation performed in client	Performing validation in client side code, generally JavaScript, provides no protection for server-side code. An attacker can simply disable JavaScript, use telnet, or use a security testing proxy such as WebScarab to bypass the client side validation.	Implementation	Alpha
Wrap-around error	Wrap around errors occur whenever a value is incriminated past the maximum value for its type and therefore "wraps around" to a very small, negative, or undefined value.	Implementation	Echo

Write-what-where condition	Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow.	Implementation	Alpha

Table 3.1: Collected Vulnerabilities

6.4 References

1- Der Linde, M. (2007): Testing Code security. 1st edition. Auerbach Publications

2- Goertzel, K. M. (2009): Introduction to Software Security. 2nd edition.

3- McGraw, G. (2004): Software Security IEEE Security & Privacy, volume 2, issue 2, Mar-Apr, 2004, Page(s): 80-83.

4- Abu-Sheikh. (2007): Reviewing and Evaluating Techniques for Modeling and Analyzing Security Requirements, Master Thesis. Blekinge Institute of Technology.

5- Ahmed S. R.(2007) :,Secure Software Development - Identification of Security Activities and Their Integration in Software Development Lifecycle, Master Thesis, Blekinge Institute of Technology.

6- - Masalin S.(2007): Software Security Design and Testing, 2nd edition.

7- Schumacher m., Ackermann r., Steinmetz r. (2009): towards security at all stages of a system's life cycle.

8- Howard M. and LeBlanc D.(2009): Writing Secure Code ,Microsoft press.

9- Howard M. (2006): Process of Performing Security Code Reviews - - Security & Privacy Magazine, IEEE magazine.

10- - Dustin E. (2006): The Secure Software Development Lifecycle, 1st edition.

11- Howard M. (2004): Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users, November MSDN Magazine.

12- Lipner S., Howard M. (2005): The Trustworthy Computing Security Development Lifecycle, Microsoft Corporation.

13- Sindre G., Opdahl A. L. (2004): Eliciting Security Requirements by Misuse Cases, Springer-Verlag London Limited.

14- Carlsson B. , Baca, D. (2005): Software Security Analysis, Execution Phase Audit, School of Engineering, Blekinge Institute of Technology.

15- Crispin C., Wagle P., Calton P. (2005): Buffer Overflows: Attacks and Defense for the Vulnerability of the Decade, 1st edition.

16- Howard M.(2005): “A Look Inside the Security Development Lifecycle at Microsoft”, MSDN Magazine.

17- Kenneth R. (2005): Bridging the Gap between Software Development and Information Security, Security & Privacy Magazine, IEEE, volume 3, issue 5, Sep-Oct, 2005.

18- Mark E. (2006): The Open Vulnerability and Assessment Language (OVAL) Initiative, 1st edition, MITRE Corporation.

19- Martin, R. A.(2005) Transformational Vulnerability Management Through Standards, The Journal of Defense Software Engineering, May, 2005,

20- Martin A. (2006): The Common Attack Pattern Enumeration and Classification (CAPEC) Initiative, 1st edition , MITRE Corporation.

21- Kenneth G. (2007) :The Consensus Audit Guidelines (CAG), 2nd edition, SANS Institute.

22- Grimes R. (2004):The true extent of insider security threats, 1st edition

23- Chase S.G. and Thompson H.H. (2005): The Software Vulnerability Guide. Charles River Media, Hingham, MA.

24- Erickson, J. Hacking (2003): The Art of Exploitation. No Starch Press, San Francisco, CA.

25- Fadia A. (2006): The Unofficial Guide to Ethical Hacking, 2nd edition. Thomson Course Technology, Boston, MA.

26- Gallagher T., Jeffries B., Landauer, L.(2006) : Hunting Security Bugs, Microsoft Press, Redmond, WA.

27- McGraw, G. Software Security: Building Security In . Pearson Education, Boston, MA, 2006.

28- Swiderski, F.,Snyder, W.(2004): Threat Modeling. Microsoft Press, Redmond, WA.

29- SAN L. (1995): Standard for Developing Life Cycle Processes. IEEE magazine. Issue 1074.

30- Schneider B. (2000): The Process of Security. ICSA Information Security Magazine, April 2000.

31- Schneier B.(2004) : Software Complexity and Security. 1st edition.

32- Kuloor C.,Eberlein A.(2003): Aspect-oriented requirements engineering for software product lines, Engineering of Computer-Based Systems, IEEE, 2003.

33- Jürjens J.(2002):Using UMLsec and goal trees for secure systems development, Proceedings of the 2002 ACM, ACM Press, 2002

34- Owasp (2010): Top 10 the ten most critical web application security risks, 2010 annual report.

35- Sindre G.(2001) :Capturing Security Requirements through Misuse Cases, 1st edition.

36- Howard M., LeBlanc D., Viega J. (2005):19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them (Security One-off), 1st edition.

37- Ruth Malan and Dana Brede Meyer - Defining Non-Functional Requirements- 2001 – 1st edition

38- Sommerville I.(2006): Software Engineering- 8th edition.

39- Al-Shalabi R., Al-aani S., Titi A., Khader M.(2007) : Security Model For E-Education Process, MIT Learning International Networks Consortium,2007, Amman, Jordan.

40- Takanen A., DeMott J., Miller C. (2007) : Fuzzing for Software Security Testing and Quality Assurance (Artech House Information Security and Privacy) – 1st edition.