New Measurements for Building Secure Software

Khalid Ibrahim Sultan

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

June 2007

Your file Votre référence
ISBN: 978-0-494-34464-4
Our file Notre référence
ISBN: 978-0-494-34464-4

ABSTRACT

New Measurements for Building Secure Software

Khalid Ibrahim Sultan

Despite the increased focus of today's research towards improving security of the cyber infrastructure, there still exists room for improvement particularly in handling security during the software development life cycle (SDLC). Developing secure software requires that the developers should address security issues as part of each phase of the software development process.

Security metrics are powerful techniques that can assist software designers and developers integrate security features into their systems from the very beginning in the development lifecycle. However, it is worth mentioning that the idea of introducing such metrics in each phase of the SDLC has not appeared before.

To cope with the situation, we propose a new set of technical security metrics for building secure software. The proposed metrics are aimed to address the security related parameters throughout the entire SDLC. The focus of this research is to examine the concept "Design for Security" as part of research efforts and to incorporate technical security issues related to the development of software from the very beginning in the development process.

This set of metrics is further divided into subgroups where each subgroup corresponds to a particular phase of the SDLC. While describing each of these metrics, it has been specified whether a particular metric can be calculated automatically or manually. For calculating the automated metrics, we built a tool using JavaCC (Java Compiler Compiler) to do so. It takes a C/C++ source code files as input. The output of the tool is basically the automated security metrics during the implementation phase. We believe that considering these metrics will help people involved in the software development process improve their applications from the security point of view.

Keywords: *Software Security, Security Metrics, Software Development Lifecycle, Design for Security.*

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to all those who gave me the possibility to complete this thesis.

I am deeply indebted to my supervisor Dr. Abdeslam En-Nouaary for his suggestion of the research topic, the continuous help, and invaluable guidance through all the research stages.

I would also like to thank Dr. Olga Ormandjieva, Dr. Abdelwahab Hamou-Lhadj, and Dr. Rabin Raut for accepting to serve in the examination committee of my thesis defense.

Special thanks go to the Ministry of Higher Education and Al-Mergheb University (Libya) for their financial support.

Last, but not least, I would like to give my special thanks to my wonderful family for their patience, sacrifice, and support during all those years of my studies.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ACSAC	Annual Computer Security Applications Conference
BNF	Backus Nour Form
CERT/CC	Computer Emergency Response Team Coordination Center
DFD	Data Flow Diagram
DREAD	Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability
FTP	File Transfer Protocol
GUI	Graphical User Interface
IT	Information Technology
ITS4	It's the Software Stupid Security Scanner
JavaCC	Java Compiler Compiler
MS-DOS	Microsoft - Diskette Operating System
NIST	National Institute of Standards and Technology
RATS	Rough Auditing Tool for Security
SDLS	Software Development Lifecycle
SMART	Specific, Measurable, Attainable, Repeatable, and Time-dependent
Splint	Secure Programming Lint
STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of Privileges
UML	Unified Modeling Language
WU-FTPD	Washington University - File Transfer Protocol Daemon

Chapter 1: Introduction

Nowadays, security problems involving computers and software are frequent, widespread, and serious. Therefore, the production of secure software is a must in the interconnected electronic world of today. The number and variety of attacks by persons and malicious software from outside organizations, particularly via the Internet, are increasing rapidly. Over 90% of security incidents reported to the Computer Emergency Response Team Coordination Center (CERT/CC) results from defects in software requirements, design or code [1]. This is basically due to the lack of consideration for the security features during the development process. Part of the problem is that security teams are often called in to add security to software at a post development stage, rather than working alongside developers during the development process.

According to the literature reviewed for this research, it is obvious that ensuring software security needs to be considered from the very beginning in the SDLC. However, security as a non-functional requirement is often an afterthought in system design and often addressed late in the software development process. As a result, the security of such systems is poor and can lead to security compromises, increased cost of maintaining the software [2,3,4,5,6].

A secure system has security designed during initiation and implementation not during maintenance because patches applied after implementation can introduce other vulnerabilities, leading to a spiral of patching, fixing, and re-patching.

The software security research community believe that all the phases of the SDLC should participate in a common activity; security assurance. Therefore, we need to move beyond the firewall and build security into software as it is being created in order to achieve a more secure environment.

From the technical point of view, yet there is a significant lack in handling security during the SDLC. In fact, there is no such metrics for security in SDLC. The metrics being developed in this research help to judge the technical objects of the SDLC from security stand point which eventually helps to develop software products with security in mind.

1.1 Security Metrics - Genesis:

Metrics are the tools that enable stakeholders to make decisions based on quantitative or qualitative assessments rather than hunches or best guesses. Security metrics are the application of quantitative, statistical, and/or mathematical analyses to measuring security functional trends and workload. In other words, it is the tracking of what each function is doing in terms of level of effort, costs, and productivity [7,8].

The purpose of measuring security is to monitor the status of measured activities and facilitate improvement in those activities by applying corrective actions, based on observed measurements.

Security metrics must yield quantifiable information for comparison purposes, apply formulas for analysis, and track changes using the same points of reference. Percentages or averages are most common, and absolute numbers are sometimes useful, depending on the activity that is being measured [9].

Moreover, data required for calculating metrics must be readily obtainable, and the process that is under consideration needs to be measurable. Only processes that are consistent and repeatable should be considered for measurement. Although the processes may be repeatable and stable, the measurable data may be difficult to obtain.

Good metrics are those that are SMART, i.e. specific, measurable, attainable, repeatable, and time-dependent, according to George Jelen of the International Systems Security Engineering Association [10]. Here is the definition of these characteristics of metrics.

Specific	Metrics should be well defined, using unambiguous language that requires no judgment or interpretation by measurement takers.
Measurable	Metrics by definition must be quantitative in nature. If something can be counted or weighed, it is measurable.
Attainable	Some measurements are specific and theoretically measurable, but repeated measurement is not practical. Metrics, therefore, must be within both the budgetary and technical limitations of the measurement takers.
Repeatable	Metrics, by definition, consist of measurements. Those measurements are often gathered by different people at different times, and potentially, across many organizations. When two different measurement takers look at the same phenomenon, they should each record the same measurement.
Time-dependent	Time-dependence is particularly important when measuring a dynamic process, such as security. Metric contexts are typically time-dependent, both because the setting of baselines requires multiple time slices and because measurements themselves are only valid for finite periods of time.

Table 1.1: Characteristics of Good Metrics

1.2 Software Metrics:

Software measurement can be defined informally as a process of quantifying the attributes of software in such a way as to characterize them according to clearly defined rules [11].

Software engineering, as any engineering approach, requires a measurement mechanism to provide feedback and assist the software development, testing, and maintenance. However, software measurement requires a definition of the environment in which the measurement is expected to be performed.

1.2.1 Why use Metrics?

It is a widely accepted principle that an activity cannot be managed if it cannot be measured. Security metrics are powerful techniques that can help software designers and developers to integrate security features into their systems very early in the software development process [12]. They also help them to be able to develop mitigation strategies for the potential threats and vulnerabilities. However, Addressing security metrics as part of the life-cycle can be a cost-effective way to achieve overall goals and build secure software.

The use of metrics has recently received a lot of attention. They provide useful data that can be analyzed and utilized in technical, operational, and business decisions across an organization. Metrics, in general, assist developers in meeting overall mission goals such as continuity of operations, safety, reliability, and security.

There are four reasons for measuring software processes, products, and resources:

- to characterize

- to evaluate
- to predict
- to improve

We characterize to gain understanding of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments.

We evaluate to determine status with respect to plans. We also evaluate to assess achievement of quality goals and to assess the impacts of technology and process improvements on products and processes.

We predict so that we can plan. Measuring for prediction involves gaining understandings of relationships among processes and products and building models of these relationships, so that the values we observe for some attributes can be used to predict others. This can help in establishing achievable goals for cost, schedule, and quality so that appropriate resources can be applied. Predictive measures are also the basis for extrapolating trends, so estimates for cost, time, and quality can be updated based on current evidence. Projections and estimates based on historical data also help us analyze risks and make design/cost tradeoffs.

We measure to improve when we gather quantitative information to help us identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance. Measures also help us plan and track improvement efforts [13,14].

Though metrics may be required by laws, regulations, or administrative mandates, they should be an integral part of any information security program. Metrics are a necessary component of accountability policies. When implemented, they will change the behavior of an organization, especially when tied to compensation, bonuses, and budgets. Collecting and analyzing security-related metrics are a critical aspect of continuous improvement in both private and public organizations. When making a business decision, security metrics can provide a set of tangible data to support one course of action over another [15,16].

1.2.2 Metrics Types:

Metrics can be characterized by what they measure as follows [17]:

- **Organizational Metrics:**

Organizational metrics are used to describe, and to track the effectiveness of organizational programs and processes. They are used to support both strategic and tactical decisions regarding an organization's use of its resources. Strategic decisions involve investment in IT architectures or technologies as well as the creation, sustaining, and termination of security programs and program elements. Tactical decisions involve the allocation of already-budgeted resources among security program elements and activities. Moreover, Organizational metrics are often used in mandated performance or compliance reporting, and are generally tied to standards of good practice [17]. Shortly, we can say that organizational metrics assess the adequacy of standards, policies, and procedures adopted by an organization.

- **Operational Metrics:**

Operational metrics are designed to assess an organization's formal policies and procedures. They are used to describe, and hence manage the risks to operational environments. Most measures of risk, or of its component factors, are operational metrics.

Moreover, Operational metrics apply to systems and organizations in an operational environment [17]. Operational metrics for security include measures of operational readiness or security posture, i.e., how well can a system or organization be expected to perform given an assumed threat environment; measures used in risk management, including security performance, compliance, and risk metrics; metrics that describe the threat environment; metrics that support incident response and vulnerability management; and other metrics produced as part of normal operations that can be used directly as, or as input to, security metrics.

- **Technical Metrics:**

Technical metrics assess the adequacy of a system or component security. Technical metrics are used to describe, and hence compare, technical objects, particularly products or systems, against standards; to compare such objects; or to assess the risks inherent in such objects. Technical metrics could help industry select products that best meet their security requirements. Technical metrics are generally associated with technical standards.

1.2.3 Properties of Software Metrics:

According to Sellers, a software measure in general should be objective, reliable, valid, and robust [18]. Objectivity means that the measurement process should not depend on the subject that performs the measurement. The reliability requires the metrics to characterize in a unique way every entity measures. Equal entities should obtain equivalent measurement values, repeated measurement in equal conditions should give same values for the same entities. The robustness requires the ability of a measure to tolerate incomplete information.

1.2.4 Which Metrics to Use?

The Goal Question Metric paradigm (GQM) is one of the best-known and widely used mechanisms for defining measurable goals, and the appropriate measurements which would characterize the achievement of those goals [14]. GQM is a six-step methodology for setting a measuring framework within the context of an organization or a specific project. The steps are outlined below:

Step 1: (Conceptual Level) Develop a set of goals.

Step 2: (Operational Level) Develop an operational model.

Step 3: (Quantitative Level) Determine the measures needed.

Step 4: Develop a mechanism (tool) to collect and analyze the data.

Step 5: Collect measurement data, and empirically validate the measures.

Step 6: Analyze the data and feed back to the projects.

In this work the GQM approach has been applied to identify the appropriate measurable goals for achieving software security, and theoretically valid measurements for indicating the achievement of security goals.

1.2.5 Documentation of Metrics:

Once applicable metrics that contain the qualities described above are identified, they will need to be documented in the Metric Detail Form in Table 1.2.The technical security metrics form that is used in our thesis is based on the metrics development guidelines from NIST Security Metrics Guide for Information Technology Systems [8]. This form will be used to document all the security aspects that can eventually be used to calculate our metrics.

Performance goal	
Performance objective	
Metric	
Purpose	
Implementation evidence	
Frequency	
Formula	
Data source	
Indicator	

Table 1.2: Metrics Detail Form

Table 1.2 contains information that defines the goal, objective and purpose of the security metric. Multiple performance objectives can correspond to a single performance goal. In such a case a different table shall be used to document each performance objective. The implementation evidence serves for validating performance of security activities and pinpointing causation factors [8]

One thing should be mentioned here is that to obtain adequate results from our research; some modifications are done in this form as follows:

Performance goal	In the metrics we are developing in this research, we will be using security goal instead of performance goal. This is because the

	purpose of our work is to provide guideline how to take security considerations into account in the life cycle of the application. This field states the desired results of considering security aspects during a certain phase in the SDLC or for a component measured by the metric.
Performance objective	Here, we are using security objective instead of performance objective. This item will list one or more security questions. Multiple security objectives can correspond to a single security goal.
Metric	This field defines the metric by describing the quantitative measurements provided by the metric.
Purpose	The purpose describes the reason of collecting the metrics.
Implementation evidence	This field lists the security assertions that are performed as implementation evidence.
Frequency	The frequency is a suggested time frame when the security function testing is done. Repeating the calculation for a metric depends basically on the metric being calculated itself and/or sometimes on the SDLC model used to build the software. For example, when incremental model is used, the metrics in the implementation phase should be repeated for each build.
Formula	The calculation to be performed that results in a numeric expression is described in this field. The implementation evidence listing serves as an input in the formula to calculate the metric.
Data source	Data source in our case depends on at which phase of the application

	life cycle we are; for example: within the design phase, the risk analysis is most properly the main data source for collecting a metric.
Indicator	The indicator is used to describe the interpretation of the metric. If the metric is, for example, a percentage then the indicator will describe the implications when the metric is very low and when it converges to 100 percent.
Implementation	We added this row to describe whether the metric being calculated can be automatically calculated because not all metrics we proposed are automated. Therefore, this filed should be filled in by either: automatically calculated or Manual.

Choosing this form was based on the fact that this standard provides sufficient coverage of the required description of the metric. This is useful for integrating security into the development life cycle of an application as it provides clear way for developing secure software from the very beginning.

1.3 Thesis Objective:

The main objective of this research is to develop a set of security metrics which if focused during the SDLC will help building secure software. The importance of our contribution is that the idea of introducing metrics for security in each phase of SDLC has not appeared before, so our work will provide software developers with security guideline to help them improve the application being developed from the security stand point during each stage of the development life cycle.

Our research aims to ensure that security is fairly represented during each phase of the development lifecycle by making sure that security is kept in mind and considered as a property of the software from the requirements phase till the retirement.

1.4 Thesis Outline:

The rest of this thesis is organized as follows. In Chapter 2, the concept of software security is discussed. Chapter 3 presents the Security in SDLC. Some techniques used to provide security in SDLC along with a brief review of SDLC are presented in this chapter. Chapter 4 introduces new metrics for building security in SDLC. Our implementations to calculate the security metrics along with results discussion is described in chapter 5. Chapter 6 shows the summary of our research as well as the improvements that could be done in the future to enhance our work.

Chapter 2: Software Security Review

The software security field is a relatively new one. Until recently, security was an afterthought; developers were typically focusing on functionality and features, waiting to implement security at the end of development. Indeed, doing so helps to address some problems but fails to address the root causes of the real security problems. In this chapter, we are going to review the concept of the software security along with some other security concepts to give an idea about this field and how it can help in producing secure software.

2.1 Software Security Definition:

Software security is the idea of engineering software so that it continues to function correctly under malicious attack. More precisely, it is the process of designing, building, and testing software for security. Software security is mostly concerned with designing software to be secure [2,19]. Issues critical to this field include, but not limited to, software risk management, design for security, and security tests. Definitions of these terminologies are explained below to make the picture clearer.

- **Software Risk Management**: software risk management is a means of identifying, and mitigating software risks. In more details, it means dealing with a concern before it becomes a crisis. This improves the chance of successful project completion and reduces the consequences of those risks that cannot be avoided. Software risk management is becoming recognized as a best practice in the software industry for reducing the surprise factor.

- **Design for Security:** design for security is the notion that security should be considered during all phases of the development cycle and should deeply influence system's design. Our contribution comes under this subfield. It aims to examine the concept of "Design for Security" by introducing a new set of metrics to be considered when developing software in order to ensure that security is fairly represented during each phase of the development lifecycle.
- **Security Tests:** Security tests are meant to include all tests applied on software to ensure that it is secure enough. This includes static and dynamic testing, which applied on source code for detecting security problems such as buffer overflow vulnerabilities, and penetration testing which performed on a system in its final production environment.

2.2 Security Concepts:

This section briefly describes fundamental concepts of software security. Understanding these concepts is vital for people who deal with security issues of software systems.

- **Security policy:** A set of rules and practices that specify or regulate how a system provides security to protect sensitive data and critical system resources.
- **Security flaw**: A security flaw is a software defect that poses a potential security risk.
- **Risk**: flaws and bugs lead to risk. Risks capture the probability that a flaw or bug will impact the purpose of the software.

- **Vulnerability**: Vulnerability is a set of conditions that allows an attacker to violate an explicit or implicit security policy.
- **Exploit**: An exploit is a piece of software or technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy.
- **Mitigation:** Mitigations are methods, processes, tools, or runtime libraries that can prevent or limit exploits against vulnerabilities.
- **Attacker**: A malicious actor who exploits vulnerabilities to achieve an objective. These objectives vary depending on the threat. An attacker can also be referred to as the adversary, malicious user, hacker, or other alias [20].

The relationship between these security concepts and computer networks are shown in the following figure.

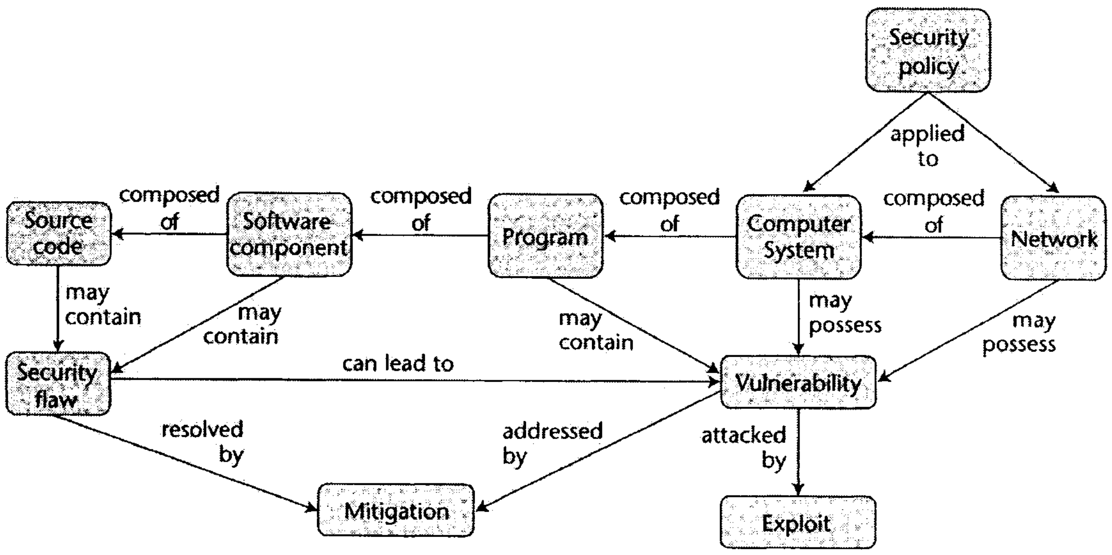


Figure 2.1: The relationship between security concepts and computer networks

As seen in figure 2.1, a network can be defined as a group of two or more computer systems linked together. A computer system is a complete, working computer.

The computer system includes not only the computer, but also any software and peripheral devices that are necessary to the computer function.

Programs are constructed from software components and custom developed source code. Software components are the elements from which larger software programs are composed [21]. Source code comprises program instructions in their original form. There should be a security policy to be applied to both network and computer systems. The network itself may possess vulnerability or more. Also, computer systems may possess vulnerability or more. Both Software component and source code may contain security flaw which might lead to vulnerability at the end. An attacker can exploit the vulnerability to attack the system. Therefore, mitigations are set to address the vulnerabilities as well as to resolve the security flaws.

2.3 Systems and Security

The problem with developing a secure system is that there are many conflicting goals to accommodate. These include functionality, usability, efficiency, scalability, simplicity, and modularity of a system. For an entity to complete a single transaction, several systems may be involved. The transacting processes can be affected by a fault that happens in one of the systems. Complex systems are made of many components that interact with each other and each component is the collection of thousands of lines of codes. A single erroneous line of code may cause a fault in the whole system. Practically, the implementation engineering process faces the reality of design tradeoff and imperfect configuration in the implementation process. That is why more secure systems can only be developed if the security engineering is part of development lifecycle [4,22].

2.4 Security services

Here in this section, some of the known security services are defined.

2.4.1 Confidentiality: Confidentiality refers to limiting information access and disclosure to authorized users "the right people", and preventing access by or disclosure to unauthorized ones "the wrong people". Usually this can be achieved by applying cryptographic functions.

2.4.2 Authentication: Authentication is a security feature that ensures authentic communication between entities. This security service provides proof of origin authentication between the sender and the responder. Human authentication factors are generally classified into three cases:

- Something the user is or does (e.g., fingerprint, DNA sequence, or signature or voice recognition)
- Something the user has (e.g., ID card, security token, or cell phone)
- Something the user knows (e.g., a password, or personal identification number (PIN))

2.4.3 Integrity: Integrity addresses the concept of trustworthiness of assets especially the data, message or a stream of data.

2.4.4 Availability: Availability refers to the availability of information resources; assuring information and communications services will be ready for use when expected. Resource unavailability is critical even if a single user uses the service [4].

2.5 Common Security Loop Holes:

Over the last years, the problem of security in software has grown exponentially [11,23,24]. Figure 2.2 shows the number of security-related software vulnerabilities reported to the CERT Coordination Center over several years. According to this graph, there is a clear and pressing need to consider security as a critical issue when developing software and to be integrated smoothly into early life-cycle activities [1].

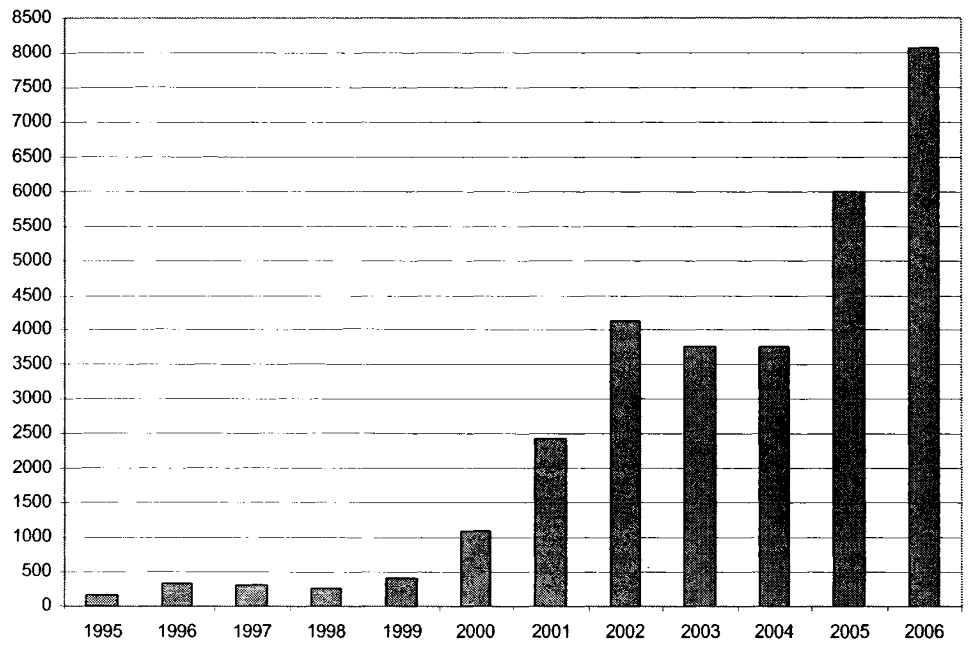


Figure 2.2: Security-related software vulnerabilities reported to CERT/CC

A lot of research has been done to address every single problem of them, we present a brief description of some common security problems that might lead to or make security vulnerabilities in software.

2.5.1 Buffer Overflow:

Buffer overflow is one of the most common problems of bad programming. It occurs when data is written outside of the boundaries of the memory allocated to a particular data structure. Depending on the location of the memory and the size of the overflow, a buffer overflow may go undetected, which can corrupt data, cause erratic behavior, or terminate the program abnormally [4].

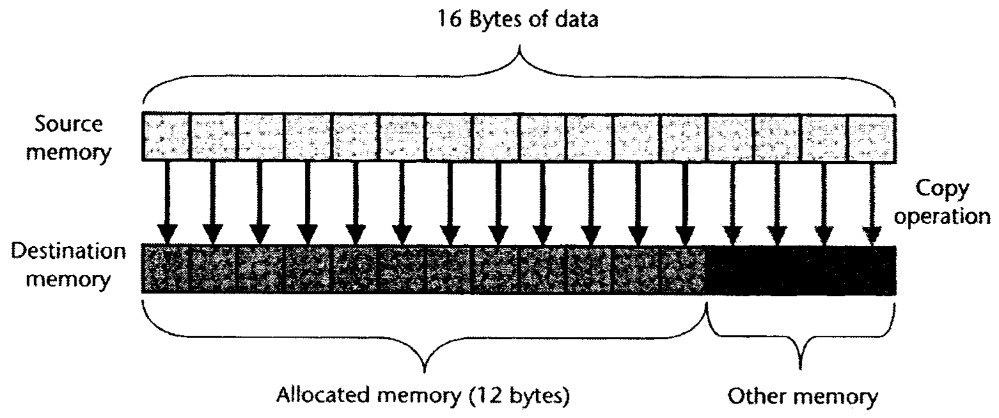


Figure 2.3: Writing beyond array bounds

Figure 2.3 depicts how buffer overflow occurs in memory by showing an example of copping 16 bytes of data into destination memory of 12 bytes. The result of this operation is that the 4 bytes just next to the destination memory will be used to store the last 4 bytes of the source.

To elaborate this point, let's consider the following C source code which exhibits a common programming mistake. Once compiled, the program will generate a buffer overflow error if run with a command-line argument string that is too long, because this argument is used to fill a buffer without checking its length.

```
/* C source code demonstrates a buffer overflow */
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
 char buffer[10];
 if (argc < 2)
 {
  fprintf(stderr, "USAGE: %s string\n", argv[0]);
  return 1;
 }
 strcpy(buffer, argv[1]);
 return 0;
}
```

Figure 2.4: C source code contains buffer overflow vulnerability

Strings of 9 or fewer characters will not cause a buffer overflow. Strings of 10 or more characters will cause an overflow: this is always incorrect but may not always result in a program error.

2.5.2 Race Condition:

Race condition is an undesirable situation that occurs when a system attempts to perform two or more operations at the same time, but because of the nature of the system, the operations must be done in the proper sequence in order to avoid problems.

For example, if one process writes to a file while another is reading from the same location then the data read may be the old contents, the new contents or some mixture of the two depending on the relative timing of the read and write operations [25].

2.5.3 Trust Management:

One of the biggest problems in software security is that developers often make poor decisions about whom and what to trust. Trust isn't something that should be extended lightly. In the case of software, this means we shouldn't even trust our own servers unless we absolutely must. Therefore, servers should only trust one another if absolutely necessary to meet requirements [4].

2.5.4 Badly Implemented Cryptography and Cryptographic Algorithms:

Badly implemented cryptographic process or algorithm with known flaws can both lead to severe vulnerabilities. Whenever possible, widely reviewed cryptographic libraries should be used. In addition, as cryptography relies on random numbers, without a good source of random numbers, the cryptographic output will be weaker.

2.5.5 Access Control:

Controlling which person, process or machine has access to which assets is one of the common security problems. Who should access what and when, should be well addressed to prevent the system from being compromised. Access control is a general way of talking about controlling access to system resources. It's the ability to permit or deny the use of an object (a passive entity, such as a system or file) by a subject (an active entity, such as an individual or process). Access control systems provide the essential services of identification and authentication, authorization, and accountability. Identification and authentication determine who can log on to a system while authorization determines what an authenticated user can do, and accountability identifies what a user did. [4]

2.6 Summary:

Though relatively new, software security is an important discipline [2]. As we have seen in the literature discussed in this chapter, security problems in software are common. The problem has multiplied exponentially over the last few years. We believe that the way security is approached has to be changed. Achieving this goal can only be met if security is considered in all SDLC phases, from requirements to product retirement.

Chapter 3: Security in Software Development Lifecycle

Security is not a feature that can be added to software once it is developed. Instead, it is an emergent property of a system. As security is not a feature, it can't be bolted on after other software features are codified. Nor it can be patched in after attacks have occurred in the field. Security must be built in from the ground up, considered a critical part of the design from the very beginning and included in every subsequent development phase all the way through fielding a complete system [26,27]. In this chapter, we review the SDLC phases and then we present some existing techniques used to provide security in the SDLC phases.

3.1 Software Development Life Cycle Background:

The SDLC is a conceptual model used to describe the stages involved in software development process from an initial feasibility study through maintenance of the completed application. Synonyms include software lifecycle and software process. Various methodologies for SDLC have been developed to guide the processes involved including the waterfall model, and Incremental Model. SDLC models describe phases of the SDLC and the order in which those phases are executed. Although there exist several such models and many companies adopt their own, all have very similar patterns. To give an idea about the lifecycle of software, we present some of these models with a brief illustration for each one of them [28,29].

3.2.1 The General Model

The general, basic model is shown below:

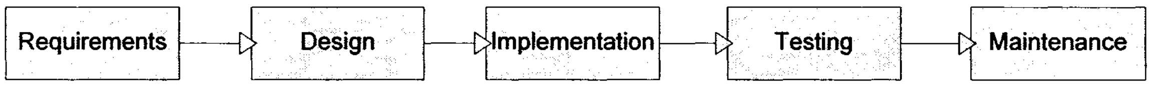


Figure 3.1: General Lifecycle Model

Each phase produces deliverables required by the next phase in the life cycle. Requirements are translated into design. Code is produced during implementation that is driven by the design. Testing verifies the deliverables of the implementation phase against requirements.

3.2.1.1. Requirements Phase:

This phase is the main focus of the project managers and stakeholders. It includes both eliciting the requirements as well as putting them into formal specifications. A requirement is a condition needed by a user to solve a problem or achieve an objective. A specification is a document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system, and often, the procedures for determining whether these provisions have been satisfied. Business requirements are gathered in this phase by meetings with managers, stakeholders and users. General questions answered during this phase are; who are going to use the system? How will they use it? What data should be input into the system? What data should be output by the system? This produces a nice big list of functionality that the system should provide.

The requirements phase is the opportunity for the product team to consider how security will be integrated into the development process, identify key security objectives, and otherwise maximize software security while minimizing disruption to plans and schedules.

3.2.1.2. Design Phase:

The software system design is produced from the results of the requirements phase. This is where the details on how the system will work are produced. Architecture, including hardware and software, communication, software design are all part of the deliverables of a design phase.

3.2.1.3. Implementation Phase:

Code is produced from the deliverables of the design phase during implementation, and this is the longest phase of the SDLC. For a developer, this is the main focus of the life cycle because this is where the code is produced.

3.2.1.4. Testing Phase:

During testing, the implementation is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements phase.

3.2.1.5. Maintenance Phase:

Software will definitely undergo changes once it is delivered to the customer. There are many reasons for the change. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period [28,29].

3.2.2 Waterfall Model:

This is the most common and classic of all life cycle models. It is very simple to understand and use. The relationship of each stage to the others can be roughly described as a waterfall, where the outputs from a specific stage serve as the initial inputs for the following stage. In a waterfall model, each phase must be completed in its entirety before the next phase can begin. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. Unlike the general model, phases do not overlap in a waterfall model [28, 29].

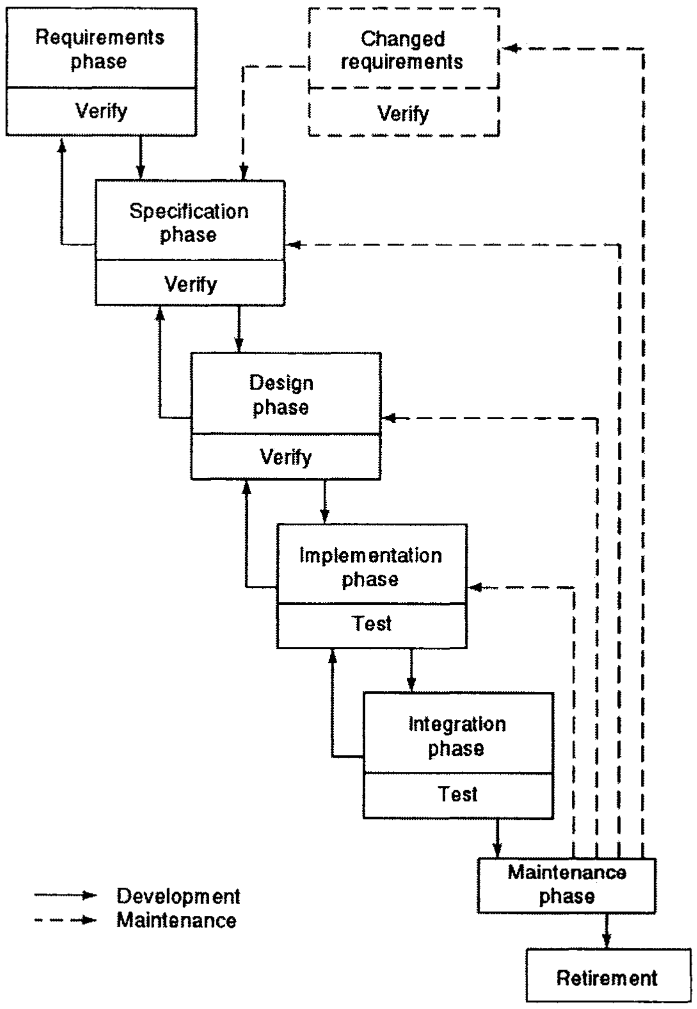


Figure 3.2: Waterfall Lifecycle Model

3.2.2.1 Advantages:

- Simple and easy to use.
- Maintenance easier
- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

3.2.2.2 Disadvantages:

- Adjusting scope during the life cycle can kill a project
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

3.2.3 Rapid Prototyping Model

A rapid prototyping model is a working model that is functionally equivalent to a subset of the product. The first step in the rapid prototype lifecycle model depicted in figure 3.3 is to build a rapid prototype and let the client and future users interact and experiment with the rapid prototype. Once the client is satisfied that the rapid prototype indeed does most of what is required, the developers can draw up the specifications document with some assurance that the product meets the client's real needs. The Focus here is on development speed, not on correctness or design. Rapid prototype model aims to prevent customer discovering new desires after seeing the original system at the end of a different process.

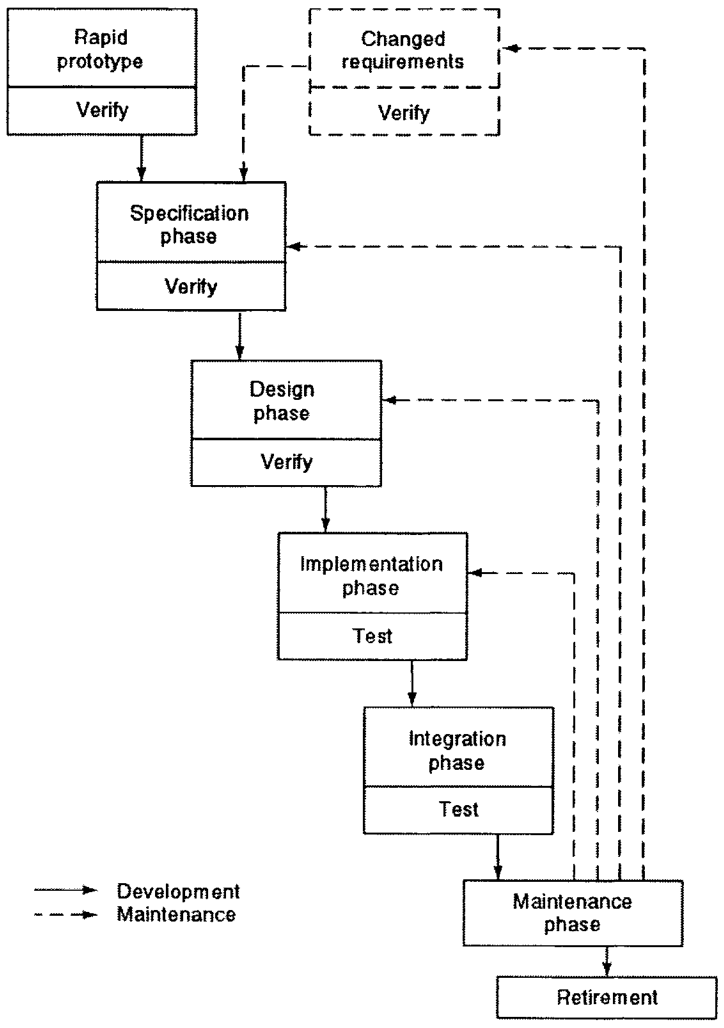


Figure 3.3: Rapid Prototype Lifecycle Model

3.2.3.1 Advantages:

- Reduces development time.
- Reduces development costs.
- Developers receive quantifiable user feedback.
- Results in higher user satisfaction.
- Exposes developers to potential future system enhancements.

3.2.3.2 Disadvantages:

- Can lead to insufficient analysis.
- Users expect the performance of the ultimate system to be the same as the prototype.
- Developers can become too attached to their prototypes
- Can cause systems to be left unfinished or implemented before they are ready.
- If sophisticated software prototypes are employed, the time saving benefit of prototyping can be lost.

3.2.4 Incremental Model

The incremental model is an intuitive approach to the waterfall model. Cycles are divided up into smaller, more easily managed iterations. Each iteration passes through the requirements, design, implementation and testing phases.

A working version of software is produced during the first iteration, so we have working software early during the SDLC. Subsequent iterations build on the initial software produced during the first iteration [28,29].

3.2.4.1 Advantages:

- Generates working software quickly and early during the software life cycle.
- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

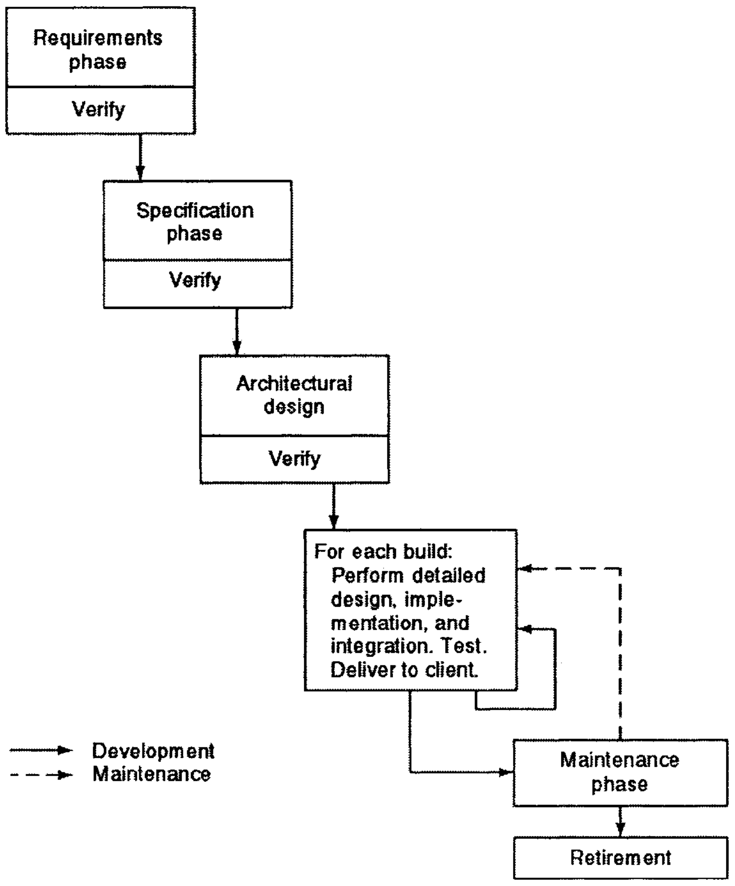


Figure 3.4: Incremental Lifecycle Model

3.2.4.2 Disadvantages:

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

3.2.5 Spiral Model

Spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations. Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase [28,29].

Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.

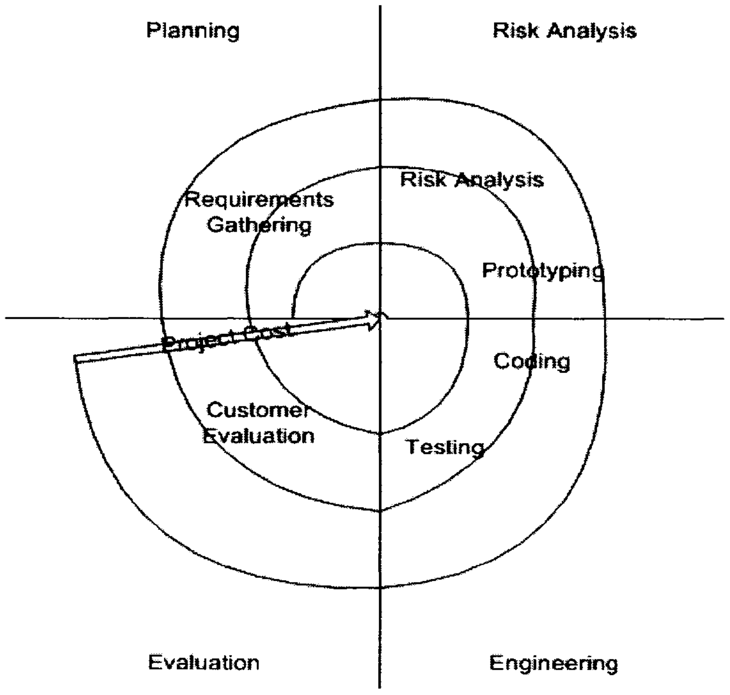


Figure 3.5: Spiral Lifecycle Model

3.2.5.1 Advantages:

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

3.2.5.2 Disadvantages:

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

3.3 Existing State-of-the-Art Technologies:

This section gives a brief description of those techniques, which are being used during SDLC to help building secure software. Making sure that security is fairly addressed during each phase of the SDLC is a main job for the software practitioners. In what follows, we present how much security is considered during each development phase by presenting some techniques used for this purpose.

3.3.1 Requirement Phase: Abuse Cases

Although use case diagrams have proven quite helpful in requirements engineering, both for eliciting requirements and getting a better overview of requirements already started, not all kinds of requirements are equally well supported by use cases. They are good for functional requirements, but poorer for non-functional requirements like security requirements which often concentrate on what should not happen in the system. Therefore, since use cases, by their nature, concentrate on what the system should do, they obviously have less to offer when describing the opposite.

This means that use cases will be good at reflecting the so-called functional requirements, but may not be so good for non-functional requirement.

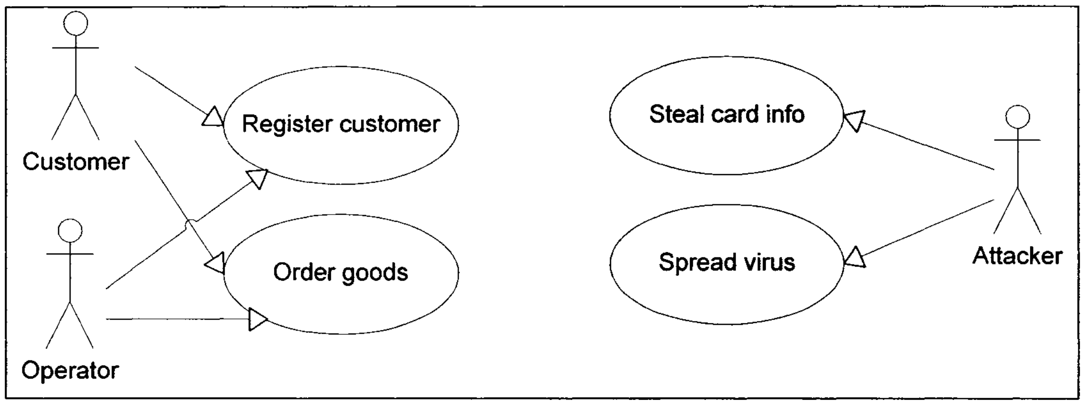


Figure 3.6: An example of wanted behavior and unwanted behavior

Figure 3.5 shows a simple example about the wanted behavior and unwanted behavior for a system that represents an online shopping process. The normal behavior expected from the system is that the customer can register him/her self and/or order goods. Also the operator can register and/or order goods for the customer. However, using abuse cases allows the developer to consider the unwanted behavior that might be done by the attacker as stealing card information and spreading viruses. As seen in the figure above, including the expected behavior of the attacker can basically help the developers to prevent such activities early in the development process.

3.3.1.1. What Are Abuse Cases?

Abuse cases (sometimes called misuse cases) are tools that can help you begin to think about your software the same way that attackers do. By thinking beyond the normative features and functions and also unexpected events, software security professionals come to better understand how to create secure and reliable software [11].

Understanding who might attack the system is really critical. Building abuse cases is a great way to get into the mind of the attackers. In addition to capturing and describing relevant attacks, abuse cases allow an analyst to think carefully through what happens when these functional security mechanisms fail or otherwise compromised.

The idea of abuse cases has a short history in the academic literature. McDermott and Fox published an early paper in abuse cases at ACSAC in 1999 [30]. Later, Sindre and Opdahl wrote a paper that explained how to extend use case diagrams with misuse cases [31].

Their basic idea is to represent the actions that systems should prevent in tandem with those that it should support so that security analysis of requirements becomes easier. Creating useful abuse cases can be simply done through a process of informed brainstorming. Approach that covers a lot of ground more quickly involves forming brainstorming teams that combine security and reliability experts with system designers. This approach relies heavily on experience and expertise.

To guide such brainstorming, software security experts ask many questions that help identify the places where the system is likely to have weaknesses. This activity mirrors the kind of thinking that an attacking adversary performs. Abuse is always possible at places where legitimate use is possible. Such brainstorming involves a careful look at all user interfaces as well as functional security requirements and considers what things most developers assume a person can't or won't do.

One of the goals of abuse cases is to decide and document a priori how the software should react to illegitimate users. The process of specifying abuse cases makes a designer differentiate appropriate use from inappropriate use very clearly. Approaching this problem involves asking the right questions. For example, how can the system distinguish between good and bad inputs? How can the system tell that a request is coming from legitimate Java Applet and not from rogue application replaying traffic? This puts the designer squarely ahead of the attacker by identifying and fixing a problem before it can even be created [11].

3.3.2 Design Phase: Threat Modeling

3.3.2.1 Threat Modeling-Defined:

Threat modeling is a security-analysis methodology that can be used to assess and document the security risks associated with an application during the design phase[25,32]. It is also known as risk analysis and risk assessment [11]. It has become a popular technique to help system designers think about the security threats that their systems will face. It enables them to develop mitigation strategies for the potential vulnerabilities [32].

3.3.2.2. Threat Modeling-the process:

The threat modeling process consists of the following steps: decompose the application, determine threats to the system, rank the threats, choose how to respond to the threats, and choose how to mitigate the threats.

- **Decompose the application:**

During the decomposition phase, the job is looking at the application through an adversary's eyes. When adversaries view the application they only see the exposed services. From these exposed services, the adversary formulates goals to attack the system. To do so, a systematic approach is required. One approach is to use DFDs (Data Flow Diagrams). By using this approach, the Threat-Modeling team is able to systematically follow the flow of data throughout the system to identify the key processes and the threats to those processes [25,32]. UML (Unified Modeling Language) could also be used during this phase to decompose an application into its key components since some parts of UML, such as activity diagrams, lend themselves well to the task as they capture process in a way that is very similar to DFDs [25].

- **Determining Threats To The System:**

The components identified during the decomposition process are used as the threat targets for the threat modeling. Each threat must then be categorized and ranked according to criteria that enable the team to prioritize them. Microsoft has developed a scheme for this categorization called STRIDE. STRIDE is a classification scheme standing for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of Privileges. [2,25]

- **Using STRIDE to categorize threats:**

The STRIDE threat model has been used in the design of secure software systems [25]. As we mentioned above, STRIDE classifies threats into six classes based on their effect:

- **Spoofing identity:** Spoofing occurs when an attacker successfully poses as an authorized user of a system.
- **Tampering with data:** Data tampering occurs when an attacker modifies, adds, deletes, or reorders data.
- **Repudiation:** Repudiation occurs when a user denies an action and no proof exists to prove that the action was performed.
- **Information disclosure.** Information disclosure occurs when information is exposed to an unauthorized user.
- **Denial of service.** Denial of service denies service to valid users. Denial of service attacks are easy to accomplish and difficult to guard against.

- **Elevation of privilege:** Elevation of privilege occurs when an unprivileged user or attacker gains higher privileges in the system than what they are authorized.

Classifying the threat makes it easier to understand what the threat allows an attacker to do and aids in assigning priority. [25,32]

- **Ranking the threats:**

In order to determine the most important threats from the threats that we have already identified in the previous phase, there is another method to determine risk called DREAD (which is an acronym for Damage potential, Reproducibility, Exploitability, Affected users and Discoverability) developed by Microsoft. DREAD Method is described in the following paragraph.

- **Using DREAD to rank the threats:**

DREAD is another step in analyzing the threats to determine the risk of the threat and the threat's conditions. When using the DREAD method, a threat-modeling team calculates security risks as an average of numeric values assigned to each of these five categories.

- **Damage potential:** Ranks the extent of damage that occurs if vulnerability is exploited.
- **Reproducibility:** Ranks how often an attempt at exploiting vulnerability really works.
- **Exploitability:** Assigns a number to the effort required to exploit the vulnerability. In addition, exploitability considers preconditions such as whether the user must be authenticated.

- **Affected users:** A value characterizing the number of installed instances of the system that would be affected if an exploit became widely available.
- **Discoverability:** Measures the likelihood that vulnerability will be found by external security researchers, hackers, and the like.

When using the DREAD method, a limited range of values should be applied in order to make the categorization of the vulnerabilities less ambiguous and more meaningful. In addition, simplifying the range by using fewer, more meaningful values makes it easier for a team to assign a DREAD rating to vulnerability [32].

- **Choosing how to respond to the threats:**

After the threats to the system are identified, we get to the point of determining how we will deal with the threats. We have four options when considering threats and how to mitigate them:

- **Do nothing:** Doing nothing is not a recommended solution because it might lead to putting the users at risk. However, sometimes the risk is so low and so costly to mitigate that it is worth accepting.
- **Inform the user of threat:** Warning the user about the problem is the second alternative that we can choose in order to respond to a threat. By informing the user about the problem, you allow the user to decide whether to use the feature or not.
- **Remove the problem:** This solution is simply accomplished by pulling the feature from the product. This choice is used in case when fixing the security problem is impossible or when there is no time to fix the problem.

- ➢ **Fix the problem:** Fixing the problem is the most obvious solution and also the most difficult as it involves more time for the development team.[25]

- **Choosing how to mitigate the threats:**

Determining how to allay threats is a two-step process. First, determine which techniques can help. Second, choose the appropriate technologies for these techniques.

Techniques are not the same as technologies. A Technique is derived from a high-level appreciation of what kinds of technologies can be applied to mitigate a threat. For example, authentication is a security technique, and on the other hand, Kerberos is a specific authentication technology [32].

3.3.3 Implementation Phase: Code Review

Software construction errors can lead to implementation flaws, some percentage of which will become security vulnerabilities. Thus, a major goal is to reduce the chance that developers introduce security vulnerabilities. To this end, we must have code reviews[25,33,34]. In terms of bugs and flaws, code review is about finding and fixing bugs. Code review processes -both manual and automated- attempt to identify security bugs prior the software's release. Current strategies are based on tools to find security vulnerabilities in the source code. For example, Splint [33] uses lightweight static analysis to detect common security vulnerabilities (including buffer overflows) and can be extended to detect new vulnerabilities. Instead, Flawfinder [35] is a program that examines source code and reports possible security weaknesses sorted by risk level. RATS [36] as its name states, it is a "Rough Auditing Tool for Security", because it performs only a rough analysis of source code. These techniques will be discussed in details later on.

Catching implementation bugs early is worth. Therefore, creating simple tool to help look for security problems in the source code is an obvious way forward. The promise of static analysis is to identify many common coding problems automatically before a program is released.

Static analysis tools examine the text of a program statically without attempting to execute it. On the other hand, manual auditing is very time consuming, and to do it effectively, human code auditors must first know security vulnerabilities before they can rigorously examine the code. Using a tool makes sense because code review is boring, difficult, and tedious.

Some tools used to scan and detect some security problems in C and C++ source code are presented in the following paragraphs [37].

3.3.3.1 ITS4:

ITS4 is a tool for statically scanning C and C++ source code for vulnerabilities. It's the first scanner built to look for security problems in code. ITS4 is actually an acronym for "It's the software Stupid Security Scanner" which was developed at Cigital in early 2000. ITS4 scans the C and C++ source code for known dangerous library calls. The tool does a small amount of checking on the argument of these calls and reports the severity of the threat. As an example, library calls that copy a fixed –length string into a buffer is rated as less sever than library calls copy the contents of an array into a buffer[38].

3.3.3.2 Splint:

Splint stands for Secure Programming Lint. It is a static analysis tool for checking C programs for security vulnerabilities and programming mistakes. The software checks that the source code is consistent with security properties stated in annotations. The annotations provide a way for the Splint software to use the preconditions to see if function implementation ensures the postconditions. It resolves preconditions using postconditions from previous statements and annotated preconditions for the function [33,39,40].

3.3.3.3 RATS:

RATS (stands for Rough Auditing Tools for Security) is a scanning tool that provides a security analyst with a list of potential trouble spots on which to focus, along with describing the problem, and potentially suggest solutions. It also provides a relative assessment of the potential severity of each problem, to better help an auditor prioritize. This tool also performs some basic analysis to try to rule out conditions that are obviously not problematic. It does only rough analysis, so it may not find errors, or find things that are not errors [36,40].

3.3.3.4 Flawfinder:

Flawfinder is a program that searches through source code looking for potential security flaws in source code by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks and providing a list of potential security flaws, sorted by risk, with the most potentially dangerous flaws shown first. Risk level depends on both the function and the values of the parameters of the function. It ignores text inside comments and strings [35,40].

3.3.3.5 Combined Dynamic and Static Testing for Detecting Buffer Overflow:

Te main idea behind this approach is to rewrite the vulnerable source code so that the modified code uses the new safe call version of old vulnerable C and C++ functions. When the rewriting is possible, the tool gives different types of warnings, depending on the complexity of the function syntax, format, and some other factors. This approach works as follows: the tool takes the C or C++ as input, and then it does parsing process. Every time it encounters a vulnerable function call, it classifies this function call. If this function call is belonging to category of their interest, then it will rewrite this vulnerable function by a safe version, which prevents the buffer overflow vulnerability. Otherwise, a warning is issued if rewriting process is not possible. This technique claims to bring down the false positive and false negative factors as low as possible [41].

3.3.4 Testing Phase: Penetration Testing

Security testing is about making sure the defensive mechanisms work correctly (e.g. you cannot spoof another user's identify), rather than that the functionality works correctly. However, the best defense is building security in, not testing it in [11,42].

A majority of security defects and vulnerabilities in software are not directly related to security functionality. Instead, security issues involve often unexpected but intentional misuses of an application discovered by an attack.

Penetration testing is about testing a system in its final production environment. For this reason, penetration testing is best suited to probing configuration problems and other environmental factors that deeply impact software security.

Though penetration testing alone is not the answer, it is extremely useful and it is an attractive late lifecycle activity. Once an application is finished, its owners subject it to penetration testing as a part of the final acceptance regimen. Penetration testing can be effective, as long as we base the testing activities on the security finding discovered and tracked from the beginning of the SDLC [43].

One reason for the prevalence of penetration testing is that it appears to be attractive as a late-lifecycle activity. Operation people not involved in the earlier parts of the development lifecycle can impose it on the software. One major limitation of this approach is that almost always represents a too-little-too-late attempt to tackle security at the end of the development cycle. Fixing things at this stage is prohibitively expensive. Post-penetration-test fixes tend to be practically reactive and defensive in mature.

The real value of penetration testing comes from probing a system in its final operating environment. Uncovering environment and configuration problems and concerns is the best result of any penetration testing. This is because mostly such problems can actually be fixed late in the lifecycle [11].

3.3.4.1 Penetrate and Patch Approach Is Bad:

Basically, rushing out a patch after the product has been broken by someone instead of coming to the realization that designing security in from the ground up is not a good way to address security issues in a software product. This scheme is known as Penetrate and Patch approach. Following are some of the limitations of this approach.

- Only known problems can be patched. Attackers might find problems that are not reported before.
- Due to the fact that patches are often fixed under pressure, they might introduce new vulnerabilities to a system.
- Sometimes patches do nothing to address the origin of the problem. Instead, they often fix the symptom of the problem.

However, we can avoid the penetrate-and-patch approach to security only by considering security as a crucial system property and not as a simple add-on feature. It is always better to design for security from scratch than to try to add security to an existing design [44].

3.4 Summary:

Building secure software is a challenge. After reviewing the existing state-of-the-art techniques developed for handling the issue of software security, we have come to the conclusion that security should be considered a critical part of the design from the very beginning and included in every subsequent phase in the development process [23] because integrating security into the SDLC may cope with the situation. Therefore, introducing security features in all the phases of the software development process extremely helps limiting costs of adding security features when it's too late and very expensive in terms of time and resources. The next chapter is devoted to propose our new metrics for building secure software.

Chapter 4: New Measurements for Building Secure Software

Among the different quality attributes of software artifacts, security has lately gained a lot of interest. However, both qualitative and quantitative methodologies to assess security are still missing. This is possibly due to the lack of knowledge about which properties must be considered when it comes to evaluate or measure security. As we said in the previous chapters, the above-mentioned gap is even larger when one considers software development phases [45].

In this chapter, we present our new metrics for building secure software. They are listed below in groups according to each phase in the SDLC. As mentioned in chapter 1, the security metrics form used in our thesis is based on the adapted version of the metrics development guidelines from NIST.

4.1 Requirement Phase:

4.1.1 Percentage of developers who have security awareness

Security goal	To check whether the developers have security knowledge or not
Security objective	Do all developers have experience in software security or at least have enough awareness about software security issues
Metric	Percentage of developers who have security awareness
Purpose	To gauge the level of expertise in security aspects among developers
Implementation evidence	1- Have security requirements been defined, with qualifications criteria, and documented? 2- How many developers involved in the application have security awareness or security knowledge?

Frequency	Once after knowing the development team
Formula	Number of developers with security awareness / Number of developers
Data source	developers training records or database; course completion certificates
Indicator	The target for this measure is making sure that we have qualified people involved in the development team. Having developers with high security awareness would definitely improve the security of the application.
Implementation	Manual calculation

4.1.2 Percentage of developers with significant security responsibilities who have received specialized training

Security goal	To determine whether the developers received adequate training to fulfill their security responsibilities?
Security objective	Did practitioners receive a special training or take any security courses to be qualified for their responsibilities.
Metric	Percentage of developers with significant security responsibilities who have received specialized training
Purpose	To gauge the level of expertise among designated security roles and security responsibilities for the system
Implementation evidence	1- Have significant security responsibilities been defined, with qualifications criteria, and documented?

	2- Are records kept of which developers have specialized security responsibilities? 3- How many developers have significant security responsibilities? 4- How many of those with significant security responsibilities have received the required training stated in their training plan?
Frequency	Once after knowing the development team
Formula	Number of developers with significant security responsibilities who have received required training / Number of developers with significant security responsibilities
Data source	developers training records or database; course completion certificates
Indicator	100 percent represents the goal of this measure, If security personnel are not given appropriate training, the system may not be equipped to combat the latest threats and vulnerabilities. Continued training enforces the availability of necessary security information.
Implementation	Manual calculation

Developers with significant security responsibilities are, for instance, developers whose job is implementing authentication or authorization mechanisms for the application being developed.

4.1.3 Percentage of security requirements that have been taken into account.

Security goal	To determine the total number of security requirements stated when collecting the requirements of an application.

Security objective	Are the security requirements that have been considered during the requirements phase enough for developing secure software?
Metric	Percentage of security requirements that have been taken into account.
Purpose	To quantify the security requirements of a system so that we can compare it to other products.
Implementation evidence	1- Is security considered as a requirement for the application? 2- How many requirements have been stated for the application? 3- How many requirements among them were security-related requirements?
Frequency	Once after getting the requirements.
Formula	Number of security requirements / Total number of requirements stated for the application
Data source	Requirement documentations
Indicator	The number of security requirements should represent all possible security requirements that could be considered in order to get a secure application because before we can build a secure software, we need to determine exactly what its security requirements are.
Implementation	Manual calculation

An example of security requirements is when an organization requests the following identification requirements to be among the security requirements for its application:

- Ensure that users are identified and that their identities are properly verified.

4.1.4 Percentage of security requirements that have not been properly interpreted

Security goal	To determine the number of the security requirements mentioned in the requirements but have been misunderstood
Security objective	Are all security requirements mentioned within the requirements phase and meaningful unambiguous?
Metric	Percentage of security requirements that have not been properly interpreted
Purpose	To avoid the misunderstanding or misinterpreting of the security requirements.
Implementation evidence	1- Is the security considered as a requirement for the application? 2- How many security requirements have been taken into account during the requirements phase? 3- How many security requirements haven't been well interpreted?
Frequency	Once after getting the requirements.
Formula	Number security requirements haven't been well interpreted / number security requirements have been taken into account
Data source	Specification documents
Indicator	The target of this measure is to be as small as possible because interpreting security requirements in a way different to the desired means that this requirement will not be achieved and might lead the system to be unsecured.
Implementation	Manual calculation

A good example for this metric is when an organization states the following Authorization Requirements:

- Ensure that users can access data and services for which they have been authorized.

This may be understood by the developers as to allow users to access data and services for which they have been authorized without preventing them to access data and services of other users. This could be solved by describing the requirements in a more meaningful way as follows:

- Ensure that users can <u>only</u> access data and services for which they have been <u>properly</u> authorized.

4.1.5 Number of security requirements that have not been considered.

Security goal	To determine the number of security requirements not mentioned during the requirements phase.
Security objective	Do the security requirements that have been taken into account within the requirements phase represent all security requirements that could be considered for an application?
Metric	Number of security requirements that have not been considered.
Purpose	To ensure that all possible security requirements are taken into account and documented during the requirement phase.
Implementation evidence	1- Is the security considered as a requirement for the application? 2- How many security requirements have not been taken into account during the requirements phase?
Frequency	Once after getting the requirements.

Formula	Number of security requirements that have not been considered
Data source	Requirement documents
Indicator	This measure should be very small or even zero if it is possible so that we can ensure that all security aspects are considered.
Implementation	Manual calculation

Identifying the missing security requirements can be accomplished by different techniques. It can be achieved by comparing the stated security requirements of the application to a standard such as the Common Criteria, an international standard for identifying and defining security requirements, or to security requirements of other products. For instance: if the security requirements of an application don't include ensuring that confidential communications and data are kept private, we can consider that Privacy Requirements of the application being developed are incomplete.

4.1.6 Number of techniques applied during the requirements phase to uncover exceptional cases.

Security goal	To determine the number of techniques used during the requirements analysis to uncover the exceptional cases.
Security objective	Are there any techniques used during the requirements analysis.
Metric	Number of techniques applied during the requirements phase to uncover exceptional cases.
Purpose	To use techniques for providing essential insight into a system's assumptions and how attackers will approach and undermine them.

Implementation evidence	1- Are exceptional cases considered to be handled from the very beginning? 2- How many techniques have been used to uncover them?
Frequency	Once after getting the requirements.
Formula	Number of techniques sued to uncover exceptional cases
Data source	Requirements' documents
Indicator	The number of techniques used during the requirements analysis should be enough to uncover exceptional cases existing in the system. However, sometimes using only one technique is good enough to get this job done.
Implementation	Manual calculation

Among the techniques used to uncover exceptional cases are the abuse cases. Abuse cases allow an analyst to think carefully through what happens when functional security mechanisms fail or are otherwise compromised.

4.1.7 Number of exceptional cases found during the requirements analysis

Security goal	To determine the number of the exceptional cases within the requirements phase.
Security objective	Is there any security requirement going to be compromised?
Metric	Number of exceptional cases found during the requirements analysis
Purpose	To make sure that the exceptional cases are discovered and handled.

Implementation evidence	1- How many techniques have been used to uncover them? 2- How many exceptional cases have been discovered and handled?
Frequency	Once after getting the requirements.
Formula	Number of exceptional cases discovered and handled
Data source	Requirement documents + Abuse cases results
Indicator	This measure should represent the actual number of the exceptional cases in the application. Exceptional cases that are not discovered, might lead the application to be easily attacked or compromised
Implementation	Manual calculation

When the security is our goal, we need to talk about and prepare for abnormal behavior. One possible way to do so is by asking many questions that help identify the places where the system is likely to have weaknesses. One question that could be asked here is: What can a bad guy do?

4.2 Design Phase:

4.2.1 Number of Threat Modeling (or security risk analysis) performed during the design phase

Security goal	To determine how many times Threat Modeling applied on the application within the design phase.
Security objective	Has Threat Modeling been used during the design phase?
Metric	Number of Threat Modeling (or security risk analysis) performed during the design phase

Purpose	To make sure that the system is free of design flaws or at least to minimize the number of design flaws.
Implementation evidence	1- Has Threat Modeling been applied on the application? 2- How many times has it been applied?
Frequency	At least once.
Formula	Number of times Threat Modeling applied
Data source	Threat Modeling documents and results.
Indicator	This metric computes the number of repeating Threat Modeling during the design phase. It should be applied enough times because systems that are not receiving regular risk assessments are likely to be exposed to threats.
Implementation	Manual calculation

Threat modeling (also known as risk analysis or risk assessment) is the most important activity during the design software process from a security viewpoint since it involves information assets, threats, vulnerabilities, risks, impacts, and mitigations.

4.2.2 Number of design flaws found in design.

Security goal	To determine the of design flaws in a system.
Security objective	Does the system contain design flaw or design flaws?
Metric	Number of design flaws found in design.
Purpose	To make sure that the design flaws are found and then handled early

	in the development life cycle so that the cost of fixing them later will be reduced.
Implementation evidence	1- Are design flaws considered to be checked and then handled during the design phase? 2- Are there any techniques used to find them? 3- How many flaws found in the design?
Frequency	At least once.
Formula	Number of architectural flaws found in design
Data source	Threat Modeling documents and results.
Indicator	This measure should represent the actual number of the flaws exist in the design of the application. By finding all possible design flaws in an application and then fixing them early we can reduce the cost of handling later on.
Implementation	Manual calculation

Design flaws occur when software is planned and specified without proper consideration of security requirements and principles. For instance, clear-text passwords are considered as design flaws.

4.2.3 Percentage of assets that have been well protected.

Security goal	To determine whether the asset of the system have been protected
Security objective	Have the assets of the system identified and protected?

Metric	Percentage of assets that have been well protected
Purpose	To ensure that the valuable components of the system have been well protected.
Implementation evidence	1- Have the system been decomposed in order to determine its assets? 2- How many assets have been found? 3- How many assets have been protected?
Frequency	At least once.
Formula	Number of assets protected / Number of assets found.
Data source	Results of Threat Modeling
Indicator	The target of this measure is 100%. This metric should cover as many assets as possible to make sure a malicious user can not reach any asset to compromise it or modify it in order to attack or/and modify it.
Implementation	Manual calculation

Assets of a system encompass abstract and physical assets. A firm's reputation is considered an important asset. On the other hand, database of the firm is an example of the physical assets.

4.3 Implementation Phase:

4.3.1 Number of implementation bugs found in the system

Security goal	To determine the coverage of implementation bugs in the application
Security objective	Are the implementation bugs of the application defined?
Metric	Number of implementation bugs found in the system
Purpose	To ensure the maximum coverage of implementation bugs existing in the application.
Implementation evidence	1- Are there any techniques or tools used during the implementation phase to find the implementation bugs? 2- How many implementation bugs have been found?
Frequency	At least once depending on the SDLC model.
Formula	Number of implementation bugs found in the application
Data source	Results of static analysis tools
Indicator	The target of this metric is to cover all implementation bugs existing in the code. They could be discovered by using some tools when performing code review.
Implementation	Automatic calculation

Implementation bugs occur when software developers make a mistake when coding software. They are independent of design. Vulnerabilities are considered implementation bugs. A good example about these bugs is Buffer overflows.

4.3.2 Percentage of buffer overflow among the total number of implementation bugs found.

Security goal	To determine how many implementation bugs that are common bugs
Security objective	Have implementation bugs been categorized?
Metric	Percentage of buffer overflow among the total number of implementation bugs found.
Purpose	To classify the vulnerabilities into groups so that mitigating them would be easier.
Implementation evidence	1- Have implementation bugs been defined or standardized? 2- How many implementation bugs found in the application? 3- How many buffer overflows found?
Frequency	At least once depending on the SDLC model.
Formula	Number of buffer overflows / number of implementation bugs found in the application
Data source	Results of static analysis tools
Indicator	This metric calculates the percentage of buffer overflows in the system. By identifying the buffer overflows we can easily mitigate them because they have been already defined and ranked as well as the ways to mitigate them have been already described.
Implementation	Automatic calculation

The reason why we focus on buffer overflows in this stage is that because buffer overflow vulnerabilities are one of the most common security flaws.

4.3.3 Number of code reviews performed within the Implementation phase

Security goal	To determine haw many times the developers applied code reviews on the code of the program
Security objective	Are the code reviews considered to find the bugs?
Metric	Number of code reviews performed within the Implementation phase
Purpose	To make sure that the code has been reviewed by certain tools in order to uncover the bugs and vulnerabilities
Implementation evidence	1- Are there any tools or techniques used to find the implementation bugs? 2- How many times has the code been reviewed?
Frequency	At least once depending on the SDLC model.
Formula	Number of code reviews performed.
Data source	Results of static analysis tools
Indicator	One of the best ways to ensure that code is resistant to attack is to have it reviewed by other trusted individuals. Code review can be performed by using static analysis tools. However, using the code reviews alone doesn't guarantee coed bug free because the tools used in the code review look for a fixed set of patterns or rules in the code.
Implementation	Manual calculation

For more information about some existing tools used to perform code review, you can refer to section 3.3.3

4.3.4 Average of implementation bugs over lines of code.

Security goal	To determine the ratio between the number of implementation bugs and the number of the lines of code
Security objective	Are implementation bugs identified?
Metric	Average of implementation bugs over lines of code.
Purpose	To minimize the number of implementation bugs over lines of code
Implementation evidence	1- How many lines of code does the program have? 2- How many implementation bugs have been found in the application?
Frequency	At least once depending on the SDLC model.
Formula	Number of implementation bugs / Number of the lines of the code
Data source	Results of static analysis tools
Indicator	This measure should be as low as possible so that we can guarantee that bugs in the system are low.
Implementation	Automatic calculation

Lines of code play an important role for system security. More complex systems are likely to be more insecure due to the greater number of lines of code needed to develop them. However, the number of errors per line of code varies greatly according to the language used, the type of quality assurance processes, and level of testing.

4.3.5 Number of exceptions that have been implemented to handle execution failures

Security goal	To determine the number of exceptions implemented to address execution failures.
Security objective	Are abnormal executions or execution failures considered to be addressed?
Metric	Number of exceptions that have been implemented to handle execution failures
Purpose	To ensure that the abnormal executions have been well handled
Implementation evidence	1- How many exceptions have been implemented?
Frequency	At least once depending on the SDLC model.
Formula	Number of exceptions have been implemented
Data source	Results of static analysis tools
Indicator	The target of this metric is to cover all the failures that might occur when executing the application. Using exceptions is appropriate for abnormal execution. A failure is a situation that has been able to occur due to a programming defect.
Implementation	Automatic calculation

The best solution is usually to print an error message. Specify the specific error messages in response of particular failure.

4.3.6 Percentage of components with incident handling and response capability

Security goal	To determine the components with incident response capability
Security objective	Are incident handling and response capability integrated into system components?
Metric	Percentage of components with incident handling and response capability
Purpose	To develop capabilities to detect problems, determine their cause, minimize the resulting damage, resolve the problem, take appropriate disciplinary or legal action, and documenting each step of the response for future reference.
Implementation evidence	1- How many components does the system have? 2- How many components among them have incident handling and response capability?
Frequency	At least once depending on the SDLC model.
Formula	Number of components with incident handling and response capability added to it / Number of components in a system
Data source	Implementation documents
Indicator	This result of this metric should be 100% because if Incident Response Capability is a part of a computer security program, incidents can be contained and ultimately prevented in a timely and cost-effective manner.
Implementation	Manual calculation

Integrating incident response capabilities and handling incidents into the product's components includes handling specific types of incidents, such as:

- Denial of Service: an attack that prevents or impairs the authorized use of networks, systems, or applications by exhausting resources
- Malicious Code: a virus, worm, or other malicious entity that infects a host.
- Unauthorized Access: a person gains logical or physical access without permission to a network, system, application, data, or other resource.

4.4 Testing Phase:

4.4.1 Percentage of the security test cases out of all test cases applied on the system

Security goal	To determine the percentage of coverage of a security functional tests
Security objective	Is the application's security function explicitly defined
Metric	Percentage of the security test cases out of all test cases applied on the system
Purpose	To ensure that maximum coverage of security test cases is attained.
Implementation evidence	1- Are security functional tests clearly defined?
Frequency	During testing , During retesting
Formula	Number of security test cases/total number of test cases
Data source	Results of testing techniques applied
Indicator	This metric can be used to measure the quality of testing process and

	in case the application is upgraded or patched, the metric can be used to make comparison between the coverage of previous results and the new retesting results.
Implementation	Manual calculation

Some of the security features should be tested explicitly such as entering wrong password. The system has to reject the attempt in such a case. Another security test case is entering wrong identity with correct password. The attempt has to be rejected too by the system.

4.4.2 Percentage of unsuccessful logins to the system

Security goal	To determine the percentage of unsuccessful logins to successful logins to the system.
Security objective	Is the login program used to audit all attempts to log into the system?
Metric	Percentage of unsuccessful logins to the system
Purpose	To make sure that unsuccessful logins to the system are recorded and hence audited so that the system can be more protected
Implementation evidence	1- How many attempts have been made to login to the system? 2- How many attempts among them were successfully logged in?
Frequency	During testing , During retesting
Formula	Number of unsuccessful logins to the system / Number of total attempts to login into to the system
Data source	Results of testing techniques applied

Indicator	Upon the result of this metric, developers can mix and match some security features to provide optimal protection of a system. Login controls and password management features are commercially attractive security features that are relatively easy to implement and most systems tend to have a lot of them. Such features include, but not limited to, Limited attempts, Last login message, and System-generated passwords.
Implementation	Automatic calculation

Usually all attempts to log into a system are audited by the login program. This creates an important trail of user accesses and attempted accesses to the system. By using the audit records for login or logoff, it is easy to determine who actually used the system. However, there are a lot of possibilities of unsuccessful logins, among them:

- Wrong user name or password.

4.5 Maintenance Phase:

4.5.1 Number of formal risk assessments performed and documented in response to changes in the application

Security goal	To determine whether risk periodically assessed?
Security objective	Are risk assessments performed and documented on a regular basis or whenever the system, facilities or other conditions change?
Metric	Number of formal risk assessments performed and documented in response to changes in the application

Purpose	To quantify the number of risk assessments completed in relation to the application's requirements.
Implementation evidence	1- Has risk analysis been applied on the application whenever the system, facilities or other conditions change? 2- How many times has it been applied?
Frequency	At least once after changing the software.
Formula	Number of times security risk analysis applied due to changes in the application
Data source	Maintenance documentation
Indicator	This metric represents the number of repeating the security risk analysis performed due to changes in the system. Appling risk analysis whenever changes done help avoid system weaknesses that might occur as a result of those changes.
Implementation	Manual calculation

Changes done in the application are classified into two groups: changes done upon changes in the requirements such as adding a new functionality or new feature to the application and changes done in response to the environments in which the application works as changes in the regulations of the country.

4.5.2 Percentage of software changes have been done due to security considerations

Security goal	To determine the percentage of software changes done due to security reasons.

Security objective	Is software considered to might be changed due to security considerations?
Metric	Percentage of software changes that have been done due to security considerations
Purpose	To determine the level of software configuration changes done upon security considerations
Implementation evidence	1- Has security been taken into account as a factor within the SDLC? 2- Are changes in software done with respect to security affects? 3- How many software changes done on the software? 4- How many security holes and vulnerabilities have been patched?
Frequency	At least once after changing the software.
Formula	Number of security holes and vulnerabilities have been patched / Total number of software changes
Data source	Maintenance documentation
Indicator	This metric helps identifying the amount of work done in order to keep the application secure. Comparing the changes done with respect to security considerations to the entire number of changes done on the application gives us an idea about whether security was taken into account or not and how much it is considered.
Implementation	Manual calculation

Software changes due to security considerations include patches released after software is delivered or any other security updates. However, it is very important that venders keep providing security patches or updates even after the deployment of their products.

4.5.3 Percentage of security holes or vulnerabilities that have been patched

Security goal	To determine the percentage of security patches done to address security holes or vulnerabilities.
Security objective	Has security holes and vulnerabilities been defined and identified?
Metric	Percentage of security holes or vulnerabilities that have been patched
Purpose	To ensure that security holes and vulnerabilities are patched early in the development life cycle.
Implementation evidence	1- How many security holes or vulnerabilities have been discovered in the system? 2- How many security holes among them have been patched?
Frequency	Whenever security holes patched.
Formula	Number of security holes or vulnerabilities found in the application / Number of security holes or vulnerabilities that have been patched
Data source	Maintenance documentation
Indicator	The target of this metric should be 100%. This measure should include patches that cover as many as possible security holes existing in the application before releasing it.
Implementation	Manual calculation

Not all changes done on software after its release are due to security consideration. However, security should to be taken as an important issue that could have the software changed even after releasing it.

4.5.4 Number of incidents reported

Security goal	To determine the number of incidents reported by the system
Security objective	Are there any technologies used to handle the incidents or any unwanted actions that might affect badly on the system?
Metric	Number of incidents reported
Purpose	To ensure that the incidents are reported so that the system can select the actions to be implemented in order to respond to these incidents.
Implementation evidence	1- Are incidents considered to be handled? 2- What are the methods that are used to count the incidents in the system? 3- How many incidents reported by the system?
Frequency	Regularly to ensure that all incidents are reported.
Formula	Number of incidents reported
Data source	Maintenance documentation
Indicator	This metric counts the incidents reported by the system. Reporting the incidents helps outline the actions to be implemented in response to these incidents such as receiving technical assistance.
Implementation	Automatic calculation

Sometimes reported incidents represent some intrusion detection data. However, an incident is the act of violating an explicit or implied security policy. A majority of security incidents results from defects in software requirements, design, or code. These activities include but are not limited to:

- Attempts (either failed or successful) to gain unauthorized access to a system.
- Unwanted disruption or denial of service.

4.5.5 The average amount of time it takes to respond to and mitigate known vulnerabilities or weaknesses.

Security goal	To ensure that corrective actions are effectively implemented
Security objective	Is there an effective process for reporting significant weakness and ensuring effective remedial action?
Metric	The average amount of time it takes to respond to and mitigate known vulnerabilities or weaknesses.
Purpose	To measure the efficiency of closing significant system weaknesses to evaluate the existence, the timeliness and effectiveness, of a process for implementing corrective actions
Implementation evidence	1- Is there any tracking system for weakness discovery and remediation implementation? 2- How many system weaknesses were discovered within the reporting period?
Frequency	Whenever a security hole or vulnerability discovered and then closed
Formula	(number of weaknesses * period of time during which they are discovered and closed) / Total number of weaknesses closed
Data source	Maintenance documentations
Indicator	A target time must be set for corrective action implementation. Results should be compared to this target. The trend for corrective

	action implementation/weakness closure should be toward shorter time frames.
Implementation	Manual calculation

4.6 Summary:

This chapter was devoted to introduce our new metrics for building secure software. As seen during the chapter, the proposed set of metrics consists of sub groups of metrics representing each phase of the SDLC. We believe that if developers can get feedback about the security problems that might occur as early as possible in the SDLC, then they can gain the time needed to fix these problems in advanced phases such as in the maintenance phase. Therefore, theses metrics are a step forward towards building secure software because taking these metrics into account during the development process can basically ensure that security is fairly represented during each phase of the development lifecycle from the requirements phase till the deployment phase.

In the following chapter, we are going to explain the implementation of the tool we built to calculate the metrics as well as to discuss the results obtained from the tool.

Chapter 5: Implementation and Results

In this research, we have developed a new set of metrics for building secure software. This is a complete suite of metrics which is a collection of several subgroups of metrics. Each of these subgroups is specifically designed for one particular phase of the SDLC. Moreover, it is clearly mentioned whether a particular metrics can be automatically calculated or not. Then, for those that can be automatically calculated, we have used JavaCC to calculate them. While for those that can't be the related tables are filled with entries manually.

This chapter is devoted to explain the structure of our tool as well as to test it by studying the results obtained when using this tool. As the major task of our tool is to find out security loop holes in software while calculating the proposed metrics, we have compared the results obtained by our tool to the results of other existing tools used to catch security problems in the source code such as ITS4. This comparison is dictated later in this chapter.

5.1. Approach Methodology:

As the metrics we have proposed need to be calculated, therefore we have implemented a tool for this purpose. During the process of Design and implementation of our tool we had in mind the target of calculating the automated metrics. The structure of the tool is described in the following figure.

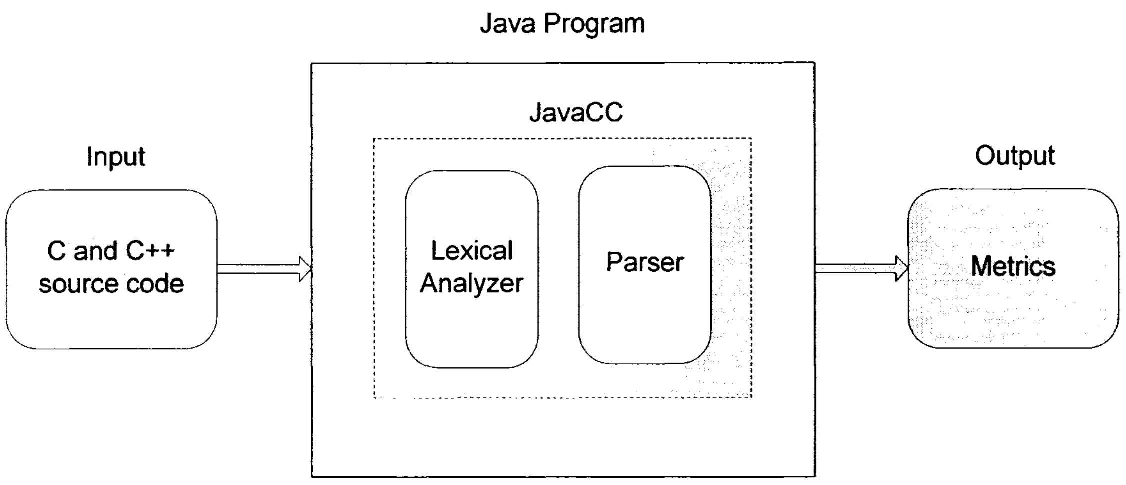


Figure 5.1: Tool Structure

As shown in the above figure, the structure of the tool consists of three main components; input characters (C++ source code), parsing stage, and finally the output (the proposed metrics).

The input is basically C and C++ source code files. The reason why we have chosen the source code of C and C++ to be input to our tool is due to the popularity of these languages. Both C and C++ are widely used for building applications. Moreover, the vast majority of vulnerabilities have been surfaced in programs written in one of these two languages. The parsing stage is a java program that consists of Java code besides JavaCC expressions. Finally, the output obtained from the tool is the suite of security metrics which can be calculated over the entire SDLC. The metrics are listed in the same order as they appear in chapter 4.

5.2 Why JavaCC?

Using a compiler is the best tool when there is a need to analyze an input file against certain format or a grammar. JavaCC (Java Compiler Compiler) is a parser generator and a lexical analyzer generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it into a Java program that can recognize matches to the grammar [46,47]. Figure 5.2 depicts the structure of a parser generated by JavaCC.

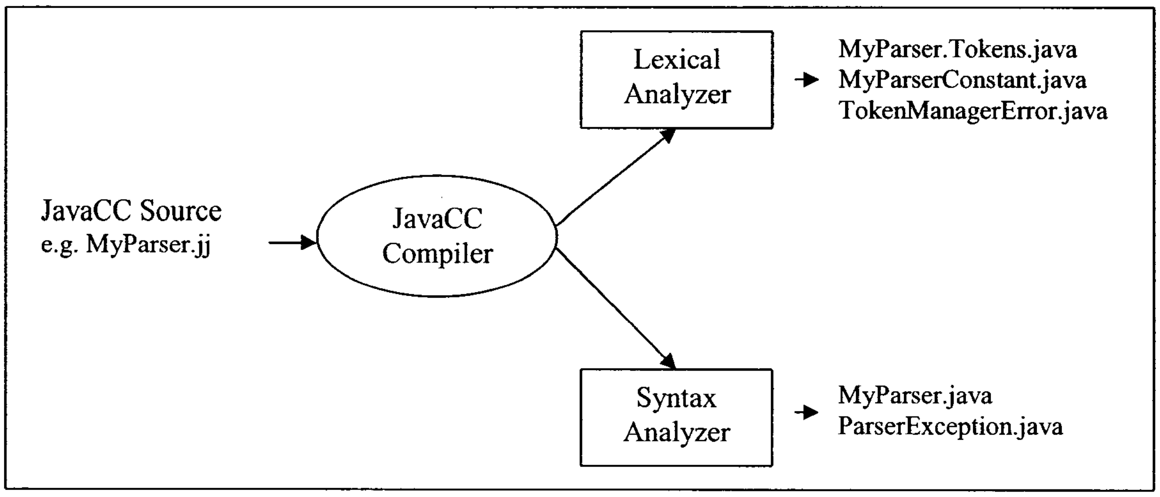


Figure 5.2: Generation of JavaCC Parser

To illustrate how JavaCC works, Figure 5.3 shows the relationship between a JavaCC generated lexical analyzer (token manager) and a JavaCC generated parser. The token manager reads in a sequence of characters and produces a sequence of objects called tokens [46]. The rules used to break the sequence of characters into a sequence of tokens obviously depend on the language; they are supplied by the programmer as a collection of regular expressions.

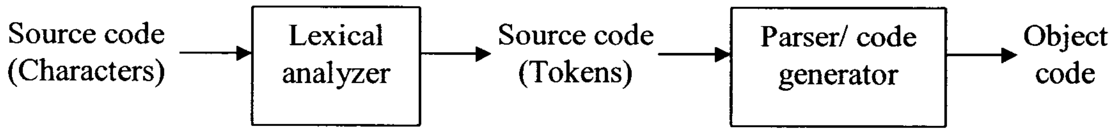


Figure 5.3: The relationship between JavaCC lexical analyzer and JavaCC parser

The parser consumes the sequence of tokens, analyses its structure, and produces whatever the programmer wants as long as it can be expressed in Java. The programmer supplies a collection of Extended BNF production rules. JavaCC uses these productions to generate the parser as a Java class.

5.3. Automatically Calculated Metrics:

Only 6 metrics among the entire suite of metrics could be automatically calculated. They are listed in table 5.1. The tool was implemented to calculate only the metrics of Implementation phase.

Metric	**SDLS phase**
Number of implementation bugs found in the system	Implementation
Percentage of buffer overflow among the total number of implementation bugs found.	Implementation
Average implementation bugs over lines of code.	Implementation
Number of exceptions that have been implemented to handle execution failures	Implementation
Number of incidents reported	Maintenance
Percentage of unsuccessful logins to the system	Maintenance

Table 5.1: Metrics declared to be automatically calculated

5.4 Security Problems:

Even though software security problems vary from buffer overflows, race conditions, Format string, etc, our focus in the implementation of our tool was on buffer overflow security problems. This is due to the fact that the buffer overflow security problems are more popular than others in C and C++ [48,49,50,51]. Also, implementing a tool to deal with all these different issues simultaneously is not only difficult but also impractical. By doing so, we believe that it would be easy for developers to reduce the overall cost of the application being developed in terms of time and money by addressing these well-known problems earlier in the development process.

We are concentrating on the collection of information about functions and data constructs that pose a risk to the security of a computer system. These insecure items can be considered as a backdoor through which malicious code can enter into a system.

We consider the following standard functions the existing of which pose a real security threat to a software system. However, not all uses of them are bad. Exploiting one of them requires an arbitrary input to be passed to the function [4].

Function	Name	Description
strcpy()	copy a string	char *strcpy(char **s1*, const char **s2*); The *strcpy()* function copies the string pointed to by *s2* (including the terminating null byte) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.
strcat()	concatenate two strings	char *strcat(char **s1*, const char **s2*); The *strcat()* function appends a copy of the string pointed to by *s2* (including the terminating null byte) to the end of the string pointed to by *s1*. The initial byte of

		s2 overwrites the null byte at the end of *s1*. If copying takes place between objects that overlap, the behavior is undefined.
fscanf(), scanf(), sscanf()	convert formatted input	The fscanf() function reads from the named input stream. The scanf() function reads from the standard input stream stdin. The sscanf() function reads from the string s. Each function reads bytes, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string format and a set of pointer arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.
memset()	set bytes in memory	void *memset(void **s*, int *c*, size_t *n*); The *memset()* function copies *c* (converted to an unsigned char) into each of the first *n* bytes of the object pointed to by *s* and returns *s*.
memmove()	copy bytes in memory with overlapping areas	void *memmove(void **s1*, const void **s2*, size_t *n*); The *memmove()* function copies *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first copied into a temporary array of *n* bytes that

		does not overlap the objects pointed to by *s1* and *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1* and returns *s1*
memccpy()	copy bytes in memory	void *memccpy(void **s1*, const void **s2*, int *c*, size_t *n*); The *memccpy()* function copies bytes from memory area *s2* into *s1*, stopping after the first occurrence of byte *c* (converted to an unsigned char) is copied, or after *n* bytes are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.
memcpy()	copy bytes in memory	void *memcpy(void **s1*, const void **s2*, size_t *n*); The *memcpy()* function copies *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.
bcopy()	memory operations	void bcopy(const void **s1*, void **s2*, size_t *n*); The *bcopy()* function copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.
fprintf(), printf(), snprintf(), sprintf()	print formatted output	The *fprintf()* function places output on the named output *stream*. The *printf()* function places output on the standard output stream *stdout*. The *sprintf()* function places output followed by the null byte, '\0', in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough space is available.

		snprintf() is identical to *sprintf()* with the addition of the *n* argument, which states the size of the buffer referred to by *s*. Each of these functions converts, formats and prints its arguments under control of the *format*.
gets()	get a string from a *stdin* stream	char *gets(char **s*); The *gets()* function reads bytes from the standard input stream, *stdin*, into the array pointed to by *s*, until a new line is read or an end-of-file condition is encountered.

Table 5.2: Vulnerable functions

One thing should be mentioned here is that arrays declared as of type constant will be considered to be allocated in the stack memory. We are interested in this kind of buffer overflows rather than those allocated in the heap memory (Heap buffer overflow) because they are less reported than the stack based buffer overflow as well as more difficult to catch [38,48,52].

To illustrate this point, we will use the following piece of code:

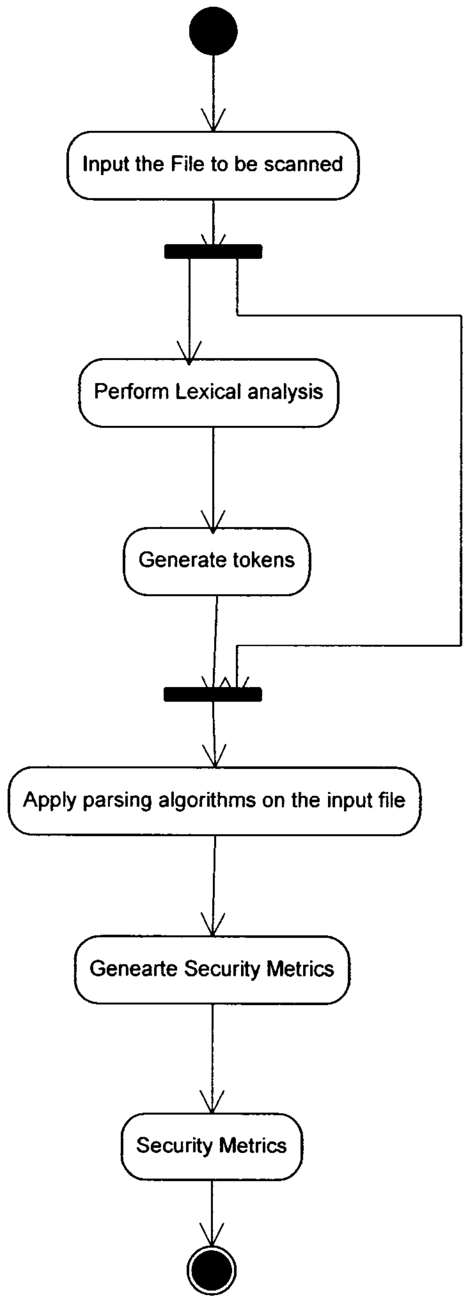
```
char x[]="Happy New Year to you";
char y[15];
strcpy(y,x);
```

Figure 5.4: C++ code poses a buffer overflow security problem.

Obviously, this code has buffer overflow security problem. This is because it doesn't perform a boundary checking. The source length (21 characters) is much longer than the destination length so the result of executing strcpy(y,x) is that the data at the end of array "x" that didn't fit into array "y" overwrites the memory locations that follow array "y". This means that memory allocations just next to array "y" will be definitely corrupted. This kind of security problems makes it easy for attackers to modify and/or corrupt data by entering an arbitrary input.

5.5 Activity Diagram:

The following activity diagram is used to display the sequence of activities in the implementation of our tool. It describes the state of activities by showing the sequence of activities performed on the input file to get the metrics.



Input the File to be scanned
Perform Lexical analysis
Generate tokens
Apply parsing algorithms on the input file
Genearte Security Metrics
Security Metrics

Figure 5.5: Activity diagram

5.6 Sequence Diagram:

In this section we depict the sequence diagram of our tool which is used to model the flow of logic within the tool in a visual manner.

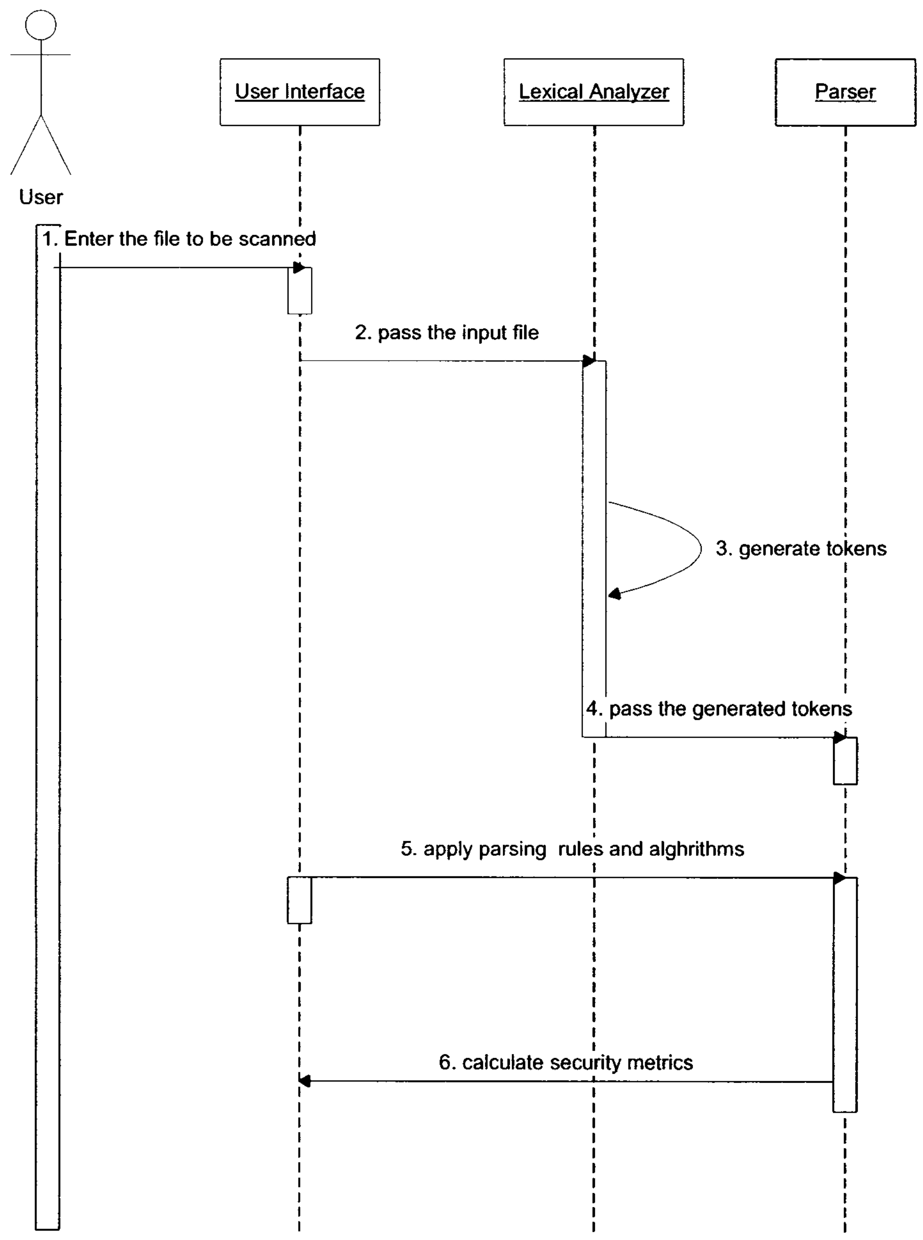

User
User Interface
Lexical Analyzer
Parser
1. Enter the file to be scanned
2. pass the input file
3. generate tokens
4. pass the generated tokens
5. apply parsing rules and alghrithms
6. calculate security metrics

Figure 5.6: Sequence diagram

5.7 Running the Tool:

In order to run the tool, the user needs to make the following three steps respectively:

1- Typing the following command in the MS-DOS prompt windows where the tool is installed:

```
\javacc  MetricsTool.jj
```

2- Typing the following command:

```
\javac MetricsTool.java
```

3- Running the tool by executing the file MetricsTool as follows:

```
\java MetricsTool
```

Having done the third step, the following window will appear:

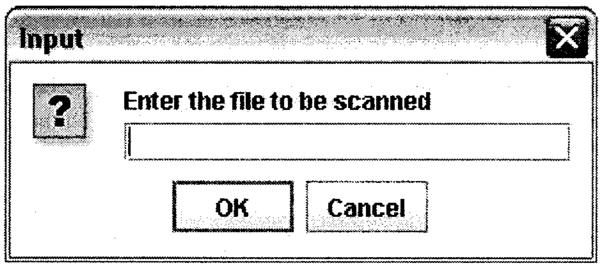
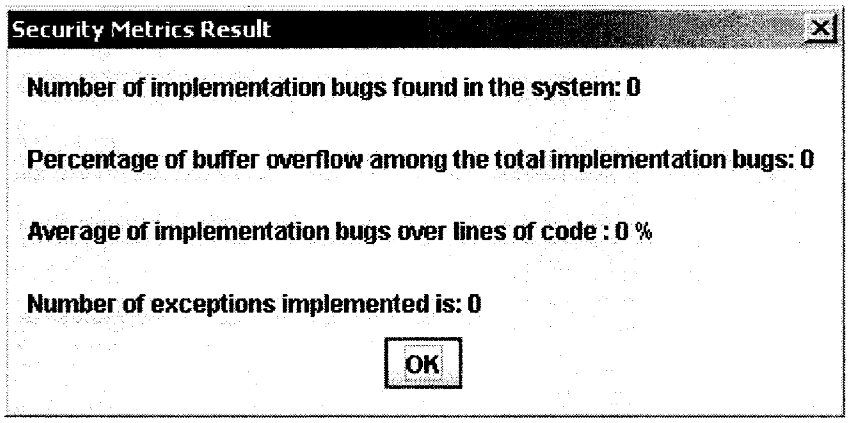


Figure 5.7: GUI for the input file

Then, by entering the proper file, the following window which lists the calculated metrics will be shown up:



Number of implementation bugs found in the system: 0

Percentage of buffer overflow among the total implementation bugs: 0

Average of implementation bugs over lines of code : 0 %

Number of exceptions implemented is: 0

Figure 5.8: GUI for displaying the proposed metric for the C/C++ input file.

As seen in the last two figures, our tool maintains a very simple interface while accomplishing what it has been designed for. The automatically calculated metrics which are computed by our tool appear in the same order as they are listed in chapter 4.

5.8 Results Analysis:

5.8.1 Simple Example:

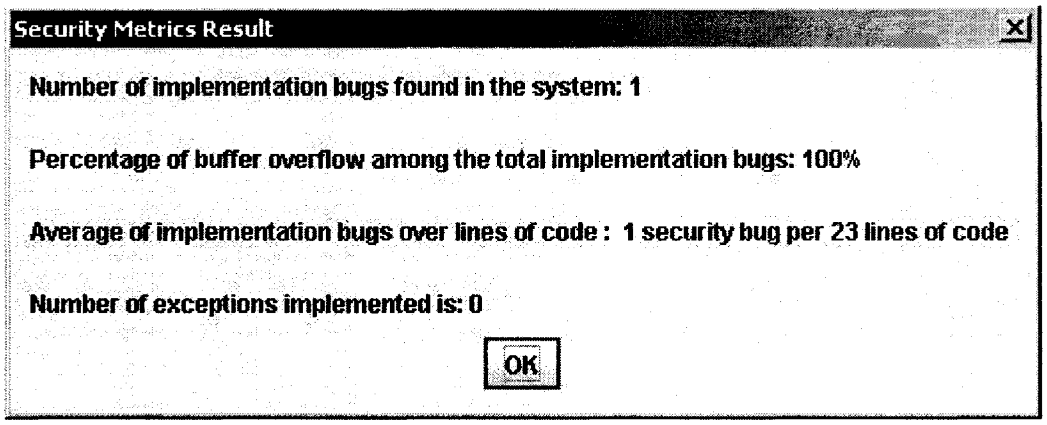
Let's consider this example to see how our tool will extract the security metrics for the program.

```
// Using strcpy
#include <iostream>
# include <cstring>
using namespace std;
int main()
{
        char x[]="Happy New Year to you ";
        char y[15];
        strcpy(y,x); // copy contents of into y
        cout<< "The string in array x is: "<<x
                <<"\nThe string in array y is: "<<y<<"\n";

        return 0; // indicate successful termination
} // end main
```

Figure 5.9: Simple example applied to our tool.

After entering the file to our tool, the security metrics will be computed and the result will be, as shown, in the following figure.



Number of implementation bugs found in the system: 1

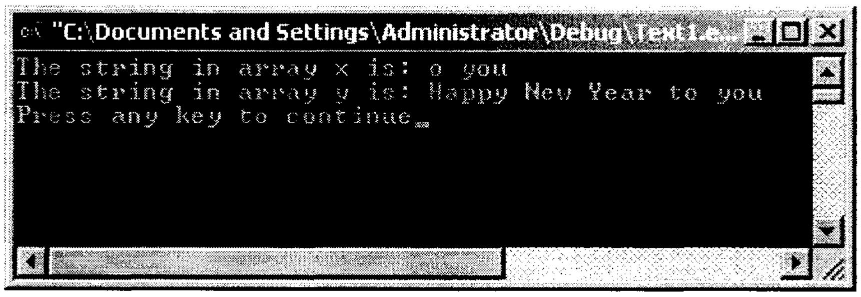
Percentage of buffer overflow among the total implementation bugs: 100%

Average of implementation bugs over lines of code : 1 security bug per 23 lines of code

Number of exceptions implemented is: 0

Figure 5.10: The result obtained from entering the simple example to our tool.

As seen from this simple example, our tool detected the bugs found in this code and calculated the metrics. The reason why the percentage of buffer overflow appears as 100%, is because we were interested in buffer overflow only while testing the input file for the security problems as we have mentioned earlier. Therefore, all implementation bugs found are buffer overflow problems. The following figure shows the result of executing this file. The impact of the buffer overflow is obvious i.e. it has corrupted the contents of array "x" as shown below.



```
The string in array x is: o you
The string in array y is: Happy New Year to you
Press any key to continue_
```

Figure 5.11: The result of executing the previous example

5.8.2 Results Discussion:

The best way of evaluating the proposed tool is to compare its results with the results obtained by other existing tools. However, as discussed in chapter 1, there in no existing tool for calculating security metrics for SDLC. Therefore, we can't evaluate our tool using the mentioned criterion. However, we can compare some results obtained by our tool like the number of buffer overflow vulnerabilities found in the source code to the results of other tools developed to do so. To perform such comparison, we are going to compare the number of buffer overflow achieved by our tool to the result obtained by ITS4 tool on the so-called wu-ftpd-2.6.2 server.

Wu-ftpd-2.6.2 is a replacement ftp daemon for Unix systems developed at Washington University. Wu-ftpd-2.6.2 is a popular ftp daemon used on the Internet, and on many anonymous ftp sites around the world. FTP is a method of transferring files between machines on a network and/or over the Internet. [53]. Many of the security issues in wu-ftpd version 2.6.2 and earlier versions over the years have been buffer overflows of one sort or another.

The wu-ftpd-2.6.2 package consists of 47 files. These files are listed in table 6.2. We applied our tool on all of the package files and the same procedure was applied on ITS4 as well.

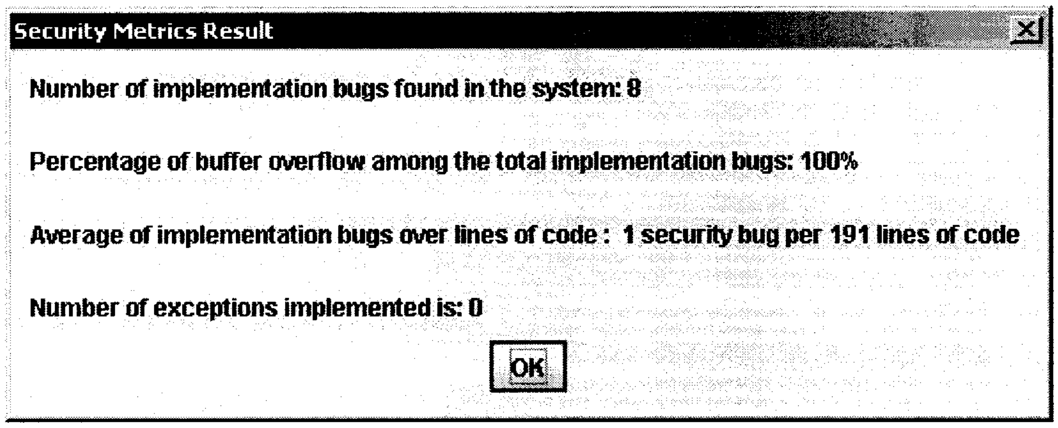
Number	**File name**	**Number**	**File name**
1.	access.c	2.	acl.c
3.	auth.c	4.	authenticate.c
5.	authuser.c	6.	ckconfig.c
7.	conversions.c	8.	copyright.c
9.	domain.c	10.	extensions.c

11.	ftpcount.c	12.	ftpd.c
13.	ftprestart.c	14.	ftpshut.c
15.	ftruncate.c	16.	ftw.c
17.	getcwd.c	18.	getpwnam.c
19.	getusershell.c	20.	glob.c
21.	glob.c	22.	hard.loop.c
23.	hostacc.c	24.	loadavg.c
25.	logwtmp.c	26.	paths.c
27.	popen.c	28.	private.c
29.	privatepw.c	30.	rdservers.c
31.	realpath.c	32.	recompress.c
33.	restrict.c	34.	routevector.c
35.	sco.c	36.	sigfix.c
37.	snprintf.c	38.	strcasestr.c
39.	strdup.c	40.	strerror.c
41.	strsep.c	42.	strstr.c
43.	syslog.c	44.	test.loadavg.c
45.	timeout.c	46.	vsnprintf.c
47.	wu_fnmatch.c		

Table 5.3: The files exist in wu-ftpd-2.6.2package

To do a logical comparison between buffer overflow security problems obtained by our tool and those of ITS4, we will focus in the following functions: strcpy(), strcat(), sprintf(), snprintf(), memcpy(), gets(), and bcopy(). After applying our tool and ITS4 tool on each file in the Wu-ftpd-2.6.2 package, we collected together all of the detected security problems over the entire package.

Figure 5.12 shows the security metrics obtained when our tool was applied on file access.c, which is included in wu-ftpd package.



Security Metrics Result

Number of implementation bugs found in the system: 8

Percentage of buffer overflow among the total implementation bugs: 100%

Average of implementation bugs over lines of code : 1 security bug per 191 lines of code

Number of exceptions implemented is: 0

Figure 5.12: The metrics obtained when our tool applied on access.c file

From the results shown in the figure above, we see that the security problems found in this file are 8 all of which indicate buffer overflow security problems. One can also notice that there is an average of one security problem over 191 lines of code. Finally, as noticed from the results there is no exception handling provision in this file.

Function	Number of buffer overflows detected
bcopy	0
gets	0
memcpy	2
snprintf	2
sprintf	0
strcat	0
strcpy	4

Table 5.4: Results obtained from our tool on access.c file

When ITS4 applied on the same file, the results were as follows:

Function	Number of buffer overflows detected
bcopy	0
gets	0
memcpy	2
snprintf	3
sprintf	0
strcat	0
strcpy	5

Table 5.5: ITS4 results for access.c file

As mentioned earlier in the chapter, we applied our tool on the entire package, file by file. As a result of applying our tool on the entire wu-ftpd-2.6.2 package, we obtained a total of 189 buffer overflow security problems.

Function	Number of buffer overflows detected
bcopy	3
gets	15
memcpy	5
snprintf	31
sprintf	52
strcat	14
strcpy	69

Table 5.6: Results obtained from our tool on wu-ftpd 2.6.6

Table 5.7 shows the results of testing with ITS4. It was run with a command line parameter that set the sensitivity cutoff to 1. At this cutoff, all vulnerabilities in the ITS4 database are reported, except ones at the level of NO_RISK. This cutoff was chosen because it was the highest that includes all of the interesting functions checked by our tool.

Function	Number of buffer overflows detected
bcopy	3
gets	17
memcpy	5
snprintf	36
sprintf	57
strcat	15
strcpy	82

Table 5.7: ITS4 results for wu-ftpd 2.6.6

When we applied ITS4 on the same application, the result was 215 problems. This observation approves the validity of the results obtained by our newly designed tool. Further, both the tools were implemented by different parsers with different ways of thinking and different rules as well. We implemented our tool using Javacc parser generator. In contrast, ITS4 was implemented using Yacc and Lex compilers.

5.9 Summary:

In this chapter, we have presented the new metrics that we have proposed for building secure software. These metrics can be applied in a systematic way to the SDLC that can be used in the real world. The tool accomplished in this research was able to calculate the Implementation Phase's metrics as claimed.

Though, there is no such metrics for SDLC as we have mentioned before to compare them to ours we compared the security problems detected by our tool to those achieved by ITS4 as a step to ensure that our tool works well. We saw that the security problems achieved by our tool are significant compared to the results obtained by ITS4.

The next chapter summarizes our contributions and some suggested future research prospects in this connection that if continued could enhance our work.

Chapter 6: Conclusion and Future Work

Over the past decade, the need to build secure software has become a dominant goal in software development [40]. However, we believe, as many others do, that security must be treated as part of the entire system's engineering process. Although, security experts have done a lot of research to provide secure software, yet there exists room for improvement.

6.1 Achievements:

In this research, we have put forward a set of new metrics for building secure software. This set will update developers throughout the entire development process about the security level of their software while being developed. This will enable them to get feedback earlier in the development process and before releasing their software. The reason of developing these metrics is due to the fact that the idea of introducing such metrics in each phase of the SDLC has never been materialized in the past. Therefore, we believe that our contribution will be a step forward in achieving the goal of producing secure software.

We implemented a tool using JavaCC for calculating the metrics of the Implementation Phase which are declared to be computed automatically in their tables in chapter 4. The results obtained by our tool were considerable compared to the results of ITS4 when both applied on the same application. It is worth mentioning that this comparison was mainly focused on the buffer overflow security problems because ITS4 is dealing with the identification of only those security problems.

Although we have used the metrics of the implementation phase to prove the concept and show that the metrics we developed are implementable, it is possible in future to do the same with those metrics which are meant for design phase for example. In design phase we can extend the developed metrics to cover the class diagram, sequence diagram, and other UML diagrams. Then we can choose a case study (an application while being developed) to calculate the metrics which specified for the design phase to demonstrate that our metrics are implementable for each phase in the SDLC.

6.2 Future Work:

Though we have collected metrics for all the phases of SDLC, we have calculated only those metrics which are meant for implementation phase using our new tool. The logic behind choosing this single phase of the SDLC for our present research is the amount of efforts required. Therefore, we are planning to extend our work in the following two directions:

- First, we plan to have the rest of the metrics calculated by our tool.
- Second, we intend to investigate the ways of incorporating our proposed metrics into other state-of-the-art security techniques including security design patterns.

References:

[1] CERT/CC Advisories. Website: http://www.cert.org/advisories.

[2] G. McGraw, "Software Security", IEEE security & privacy, April 2004, pp. 80-83.

[3] D. P. Gilliam, T. L.Wolfe, J. S. Sherif, "Software Security Checklist for the Software Life Cycle", Processing of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise, IEEE Computer Society, June 2003, Austria, pp. 243-248.

[4] J. Viega & G. McGraw, "Building Secure Software: How to Avoid Security Problems the Right Way", Addison-Wesley, Boston, 1st edition, 2002.

[5] N. Davis, "Developing Secure Software", Software Tech news: secure software engineering, Vol. 8, No. 2, July 2005.

[6] M. Bishop, "Computer Security: Art and Science", Addison-Wesley, Boston, 1st edition, 2002.

[7] J. A. Chaula, L. Yngström, and S. Kowalski, "Security Metrics and Evaluation of Information Systems Security", in the 4th Annual Conference on Information Security for South Africa, Midrand, South Africa, 2004.

[8] M. Swanson, N. Bartol, J. Sabato, J. Hash, and L. Graffo, "Security Metrics Guide for Information Technology Systems". NIST Special Publication 800-55, National Institute of Standards and Technology, Gaithersburg, Maryland. July 2003. Available at http://csrc.nist.gov/publications/nistpubs/800-55/sp800-55.pdf

[9] J. C. Munson, "Software Engineering Measurement", Auerbach Publications, CRC Press LLC, USA, 2003.

[10] G. Jelen, "SSE-CMM Security Metrics." NIST and CSSPAB Workshop, Washington D.C., 2000.

[11] N. Fenton, and S. L. Pfleeger, "Software Metrics: A Rigorous and Practical Approach", International Thomson Computer Press, 1997.

[12] P. Goodman, "Software metrics: best practices for successful IT management", Rothstein Associates Inc., 2004.

[13] S. A. Whitmine, "Object Oriented Design Measurement", Wily Computer Publications, 1997.

[14] R. E. Park, W. B. Goethert, W. A. Florac, "Goal-Driven Software Measurement: A Guidebook". 1996.

[15] G. McGraw, "Software Security: Building Security In", Addison-Wesley, Boston, 1st edition, 2006.

[16] S.C. Payne, "A Guide to Security Metrics." SANS Security Essentials GSEC Practical Assignment Version 1.2e, SysAdmin, Audit, Network, Security Institute, Bethesda, Maryland. 2001. www.sans.org/rr/whitepapers/auditing/55.php

[17] R. B. Vaughn, R. Henning, and A. Siraj, "Information Assurance Measures and Metrics - State of Practice and Proposed Taxonomy", Proceedings of the 36th Hawaii International Conference on System Sciences, IEEE Computer Society, 2002.

[18] B. Henderson-Sellers, "Object-Oriented Metrics: Measures of Complexity", Prentice Hall PTR, 1 edition, 1995

[19] G. McGraw, "Building Secure Software: Better than Protecting Bad Software", IEEE Software, December 2002, pp. 57-59.

[20] R. C. Seacord, "Secure coding in C and C++ ", Addison-Wesley, Boston, 1st edition, 2006.

[21] K. C. Wallnau, S. Hissam, and R. C. Seacord "Building Systems from Commercial Components", Addison-Wesley, Boston, 2002.

[22] I. Flechais, M. A. Sasse, and S. M. V. Hailes, "Bringing Security Home: A Process For Developing Secure And Usable Systems", In ACM/SIGSAC New Security Paradigms Workshop, *2003.*

[23] P. Devanbu & S. Stubblebine, "Software Engineering for security; a roadmap", in processing of the conference on the future of the software engineering. Pp. 227-239, ACM Press 2000.

[24] P. Falcarin, M. Morisio, "Developing Secure Software and Systems", in IEC Network Security: Technology Advances, Strategies, and Change Drivers, July 2004.

[25] M. Howard & D. LeBlanc, "Writing Secure Code", Microsoft press, 2nd edition, 2003.

[26] J. A. Whittaker & M. Howard, "Building More Secure Software With Improved Development Process", IEEE Security & Privacy, pp. 63-65, 2004.

[27] A. K. Ghosh, C. Howell, and J. A. Whittaker, "Building Software Securely from the Ground Up", IEEE Software, February 2002, pp. 14-16.

[28] S. R. Schach, "Object-Oriented and Classical Software Engineering", Addison-Wesley, Boston, 7th Edition, 2007.

[29] R. S. Pressman, "Software Engineering, A Practitioner's Approach", McGraw Hill, 6th edition, 2005.

[30] J. McDermott & C. Fox, "Using Abuse Case Model for Security Requirements Analysis", Processing of the 15th Annual Computer Security Application conference", Scottsdale, IEEE Computer Society Press, 1999, pp.13-15.

[31] G. Sindre & A. L. Opdahl, "Eliciting Security Requirement by Misuse Cases", Requirements Engineering, Springer London, Volume 10, Issue 1, January 2005, pp. 34 – 44.

[32] F. Swiderski & W. Snyder, "Threat Modeling", Microsoft press, 2004.

[33] D. Evan, "Secure Programming Lint (Splint)", Department of Computer Science, University of Virginia. 2003. www.splint.org

[34] M. G. Graff and K. R. Van Wyk, "Secure Coding: Principles and Practices", O'Reilly and Associates, California, 1st edition, 2003.

[35] Flawfinder home page: http://www.dwheeler.com/flawfinder

[36] Secure Software Inc. October, 2003, http://www.secure software.com

[37] G. McGraw, "Static Analysis for Security", IEEE Security & Privacy, December 2004, pp. 76-79.

[38] J. Viega, J. T. Bloch, Y. Kohno, G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code", Reliable Software Technologies, Dulles, Virginia, 16th Annual Computer Security Applications conference. December, 2000.

[39] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis", IEEE Software, February 2002, pp 42-51.

[40] J. Tevis and J. Hamilton, "Methods for the Prevention, Detection and Removal of Software Security Vulnerabilities", ACM Southeast Conference 04. 2004, USA.

[41] S. Alouneh and A. En-Nouaary, "A New Method for Testing Buffer Overflow Vulnerabilities in C and C++ Programs., in Fifth International Network Conference, Samos Island, Greece, July 2005.

[42] M. R. Stytz and J. A. Whittaker, "Why Security Testing is Hard", IEEE Security and Privacy, August 2003, pp 83-86.

[43] G. McGraw, "Software Penetration Testing", IEEE Security and Privacy, 2005.

[44] G. McGraw, "Testing for Security Development: Why We Should Scrap Penetrate-And Patch", IEEE Aerospace and Electronic Systems, vol. 13, no. 4 April 1998, pp. 13-15.

[45] R. Scandariato, B. D. Win, and W. Joosen, "Towards a Measuring Framework for Security Properties of Software", ACM, Virginian, USA, 2006, pp 27-29.

[46] G. Succi and R. W. Wong, "The Application of JavaCC to Develop a C/C++ Preprocessor", Department of Electrical and Computer Engineering, University of Calgary, Calgary, Canada.

[47] T. S. Norvell, "Introduction to JavaCC", University of British Colombia, Vancouver, 2003.

[48] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities", Proceedings of the 2003 Network and Distributed System Security Symposium, February, 2003, pp. 123–130 .

[49] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". In Proceedings of the DARPA Information Survivability Conference and Expo, 1999.

[50] T. Plum and D. M. Keaton, "Eliminating Buffer Overflows Using the Compile or a Standalone Tool", NIST Workshop on Software Assurance Tools, Technologies, and Metrics at ASE '05, 2006.

[51] M. Howard and J. A. Whittaker, "Secure Coding in C and C++", IEEE Security and Privacy, 2006, pp. 74-76.

[52] D. A. Wheeler, "Secure Programming for Linux and Unix HOWTO", v3.010, 3 March 2003.

[53] WU-FTPD Development Group, website: http://www.wu-ftpd.org