Combining Static Source Code Analysis and Threat Assessment Modeling For Testing Open Source Software Security

By
Abraham Ghebrehiwet Ghebremedhin

Supervisor
Vladimir A Oleshchuk

Thesis submitted in Partial Fulfillment of the
Requirements for the Master of Science Degree in
Information and Communication Technology

University of Agder, 2012
Faculty of Engineering and Science
Department of Information and Communication Technology

Summary

Nowadays, large number of open source software are being developed by the open source software communities and made available to the public domain. Open source software started with the promises for better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in. Risk free (secure) software is the key requirement for organizations planning to implement open source software as part of their software stack. In order to understand Open source software security better, this thesis offers two approaches, which can be used for testing and analyzing the security of open source software.

This project was performed at Agder University in the Department of Information and Communication Technology. The goals of the project were to describe the various common source code vulnerabilities. Validate their presence in open source software using static source code analysis technique. And finally, develop a threat assessment model to further explore and document potential threats so that to be able to build a solid security strategy to guard against the threats.

A brief introduction and background theory of open source software security is followed by the two approaches introduced for testing the security of open source software. These approaches are Static Source Code Analysis and Threat Assessment Modeling techniques.

The practical part of the experiment consists of two steps. In the first step static source code analysis is performed on a test case application, using two tools namely, Flawfinder and RATS. This approach was chosen mainly because it is normally similar with code auditing to the extent of concentrating on the actual source code, but instead of auditing the code manually, it uses automated tools. The second step threat assessment modeling (Threat Risk Modeling) was introduced to discover the design and architectural vulnerabilities on the test case application that would otherwise be impossible to detect them using static source code analysis. For this purpose, the tool Microsoft Threat Analysis & Modeling tool v2.1 was used.

A case study was performed on an open source application rdpdesk-3.2 (remote desktop connection manager). In this case study the application was scanned to check for the presence of source code related vulnerabilities using the two static source code analysis tools. After the scanning were performed, a threat risk model was developed for the application that was used in the detection of design and architectural vulnerabilities. The results of the case study led to a clear conclusion that the application contains large number of vulnerabilities. Moreover, the results also made clear that the combined use of both approaches can be used to improve the security of open source software by verifying the source code as well as design and architectural vulnerabilities.

Abstract

Organizations that implement open source software in their system before they verify the software for security vulnerabilities are more vulnerable to attacks. Therefore, it is important to discover and fix vulnerabilities in open source software before their implementation. Nowadays different techniques exist that help in the vulnerability discovery. The goal of this project is to improve the security of open source software by discovering various source code vulnerabilities using static source code analysis technique, and design and architectural vulnerabilities by developing a threat risk model. I conducted a case study on a remote desktop connection manager application using two static analysis tools and one threat risk modeling tool. In the case study performed, I found that the static analysis tools discovered large number of different types of vulnerabilities on the application. I also discovered some design and architectural vulnerabilities using the threat risk modeling tool. The results obtained from the case study suggest that it is unsafe to deploy open source software in a system without first verifying it for vulnerabilities.

Preface

This thesis is submitted to Agder University as a final work of the Master's Program in Information and Communication Technology to fulfill requirements for the Master of Science degree.

This thesis was performed at the Department of Information and Communication Technology Grimstad, under the supervision of Prof. Vladimir A Oleshchuk.

I would like to thank my supervisor for his valuable help, suggestions, and feedback, which made this work possible. Besides, I would like to thank my wife for her full support.

Grimstad, May 2012

Abraham Ghebrehiwet

Table of Contents

Appendices

List of Figures

List of Tables

Chapter 1

Introduction

The security of a system that makes use of open source software mainly depends on the software used as well as the security practices followed. Nowadays, large numbers of open source software are being developed by the open source software communities and made available to the public domain. Open source software started with the promises for better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in [5]. Even though the mentioned promises are very attractive, the issue of security must be considered seriously. This is due to the fact that open source software also offers equal opportunity for attackers to detect and misuse software vulnerabilities, affect the code by adding insecure code and distributing tainted version of the software.

Open source software security is a measure that assures open source software system is free from danger or risk of attacks. Risk free software is the key requirement for organizations planning to implement open source software as part of their software stack, particularly if the software will play a major role.

Nowadays securing organizations' perimeter has largely been successful. As a result hackers and other malicious individuals have turned their attention in to attacking organizations applications. This issue forced organizations to focus highly on securing the application layer. However, by using embedded code or exploiting flaws in software, hackers gain control of company computers and get access to confidential information and customer records. Therefore, proper source code review must be performed to evaluate potential vulnerabilities of the code.

1.1 Problem Statement

Various software security professionals [7, 8] claim that transparency makes systems more secure. This is true in a sense that by allowing the source code to be open everyone can access the code, therefore, bugs and vulnerabilities are found more quickly and thus are fixed more quickly, closing up security holes faster and this increases software security. Anyone who is interested in improving the software is also free to create a better, more secure version of the software.

Trusting the above mentioned arguments of open source software advocates however, organizations might be tempted to deploy open source software before they evaluate the software for security. And this may pose several security challenges to the organizations unless the code is treated for security

vulnerabilities before it can be used. Therefore there is a demand for reviewing the software for security vulnerabilities, reveal and fix them before the software is used.

1.2 Thesis Objective

Software security is an understanding of software-induced security dangers and how to deal with them. The presence of vulnerabilities on software makes it so easy for attackers to target software, which can then exploit to violate the security. According to CERT most successful attacks result from targeting and exploiting known, software vulnerabilities and insecure software configurations, many of which are introduced during design and coding [1].
Therefore the main objectives of this thesis are as follows:

1. Describe the various common source code security vulnerabilities.
2. Validate their presence in open source software using static source code analysis technique.
3. Develop threat assessment model to further explore and document potential threats so that to be able to build a solid security strategy to guard against the threats.

1.3 Thesis Domain

The investigation for the test case has been carried out on rdpdesk-3.2 (remote desktop connection manager) an open source software written in C/C++ language. Certain assumptions and facts are taken from the test case software; the same approaches themselves however can be applied to any kind of open source software.

1.4 Importance of Topic

These days businesses and other organizations store huge amounts of sensitive and critical data on their computer systems and this makes systems security vital. Also, they have moved their transactions online and they rely more on applications that are part of a network. If such businesses or organizations implement open source in to their software stack, attackers might compromise sensitive and critical information by exploiting some of the security vulnerabilities in the software.
Therefore in order to highly benefit from open source software, the source code must be properly reviewed to uncover any vulnerability, reveal and fix them before the software is deployed. In this way, businesses and other users of open source software can keep their sensitive and critical information secure and thereby reducing the impact of losses and damages.

1.5 Related Works

.

There have been several significant efforts on open source code security. Azzam Mourad et al. [82] defined the concept of software security hardening, which allow developers and maintainers to deploy and harden security features and remedy present vulnerabilities and threats into existing open source software. They also propose a classification of the different levels at which the hardening can be applied and a methodology for hardening of high level security into applications based on a well-defined security ontology. In addition to this contribution, they elaborate the methods for hardening security vulnerabilities found in C according to the classification they propose. Anne Immonen et al. [83] contributed a method which provides clear guidelines to assist integrators to perform an evaluation in their own software development environment. Their method assists in the technical trustworthiness evaluation, containing tools for reliability analysis and testing of open source components (OSC). Prasanth Anbalagan et al. [84] presented an analysis and classification of 43,710 vulnerabilities from the Open Source National Vulnerability Database and vulnerabilities for two specific products - Bugzilla and FEDORA. With the focus on the disclosure and exploits of security problems with respect to calendar time and in-service time, they investigated a unifying approach to understand security as a component of reliability. Guido Schryen et al. [85] presented the first comprehensive empirical investigation of published vulnerabilities and patches of 17 widely deployed open source and closed source software packages, including operating systems, database systems, web browsers, email clients, and office systems. Their results suggested that it is not the particular software development style that determines the severity of vulnerabilities and vendors' patching behavior, but rather the specific application type and the policy of the particular development community, respectively. Carlos Ballester Lafuente [44] developed a software security guideline that can be used for evaluating methods and measuring security in open source projects with a high security implication such as healthcare applications for example, where the privacy and security are crucial factors. After applying the guideline, he found several vulnerabilities, like session hijacking or capturing login information on real time.

Ashish Aggarwal et al. [86] described a methodology which integrates static and dynamic analysis approaches in a complimentary manner. The methodology they used adopts the strengths of the two and eliminates their weaknesses. They dealt with buffer overflow vulnerability with pointer aliasing. Nathaniel Ayewah et al. [54] they proposed the need to develop procedures and best practices that make use of static-analysis tools more effective than alternative uses of developer time, such as spending additional time performing manual code review or writing test cases. Through user surveys they concluded that findbugs answered their needs. Lucas Torri et al. [87] in their paper have surveyed ten different free/open source tools that perform static software analysis and evaluated their use in embedded software. Furthermore, they discussed possible directions to improve the use of static analysis tools in the embedded domain. Peng Li et al. [88] their paper focused on software vulnerability static analysis techniques and tools. First they discussed the commonly-used static analysis

techniques and tools, and compared these tools in a technical perspective, and then they analyzed the characteristics of the tools through the experiment, finally, combining dynamic analysis, they proposed an efficient software vulnerability detection method. George Chatzieleftheriou et al. and Magnus Ågren respectively [89, 90] they compared four open source and two commercial static analysis tools in terms of their effectiveness and efficiency of their detection capability. For their experiments they used C code. They used the obtained results for identifying the appropriate tool, in terms of cost-effectiveness. Alexander Ivanov [17] examined a sample set of vulnerabilities and developed a vulnerability classification based on common source code patterns.

1.6 Research Methodology

Analyzing open source software for the existence of various vulnerabilities is a curtail process that need to be done before deploying the software in your system. Code auditing, dynamic source code analysis, penetration test, static source code analysis, threat risk modeling and so forth are some of the techniques used for such analysis. In this thesis, static source code analysis and threat risk modeling techniques are used to reveal the different source code, design and architectural security flaws that exist in open source code software. The static source code analysis technique is mainly used for the identification of vulnerabilities on a test case application source code, while the threat risk modeling technique is used in the identification of the design and architectural vulnerabilities of the test case application.

The practical experiments were done on the test case application (rdpdesk-3.2) using two static source code analysis tools (Flawfinder and RATS) and one threat risk modeling tool (Microsoft Threat Analysis & Modeling tool v2.1).

1.7 Contribution of Thesis

The key contribution of this paper is to develop open source software security best practices by combining static source code analysis and threat assessment modeling techniques. The result obtained from this combination will play significant role in the process of exploring and identifying source code, design and architectural potential threats and vulnerabilities to open source software. Finally, based on the identified threats and vulnerabilities, it will be easy to build a solid security strategy to guard against the identified threats to the software.

1.8 Thesis Structure

Chapter 2 reviews some background theory that helps understand the remaining chapters. Chapter 3 describes the two research methodologies used for this project, static source code analysis and threat assessment modeling techniques. In Chapter 4, an experiment is performed on a test case application using the methodologies mentioned in chapter 3. Chapter 5 presents a detailed description and analysis of the research results. Chapter 6 provides a discussion based on the results found. Finally chapter 7 presents conclusions of the research and recommendations for future research. Figure 1.1 shows the thesis structure.

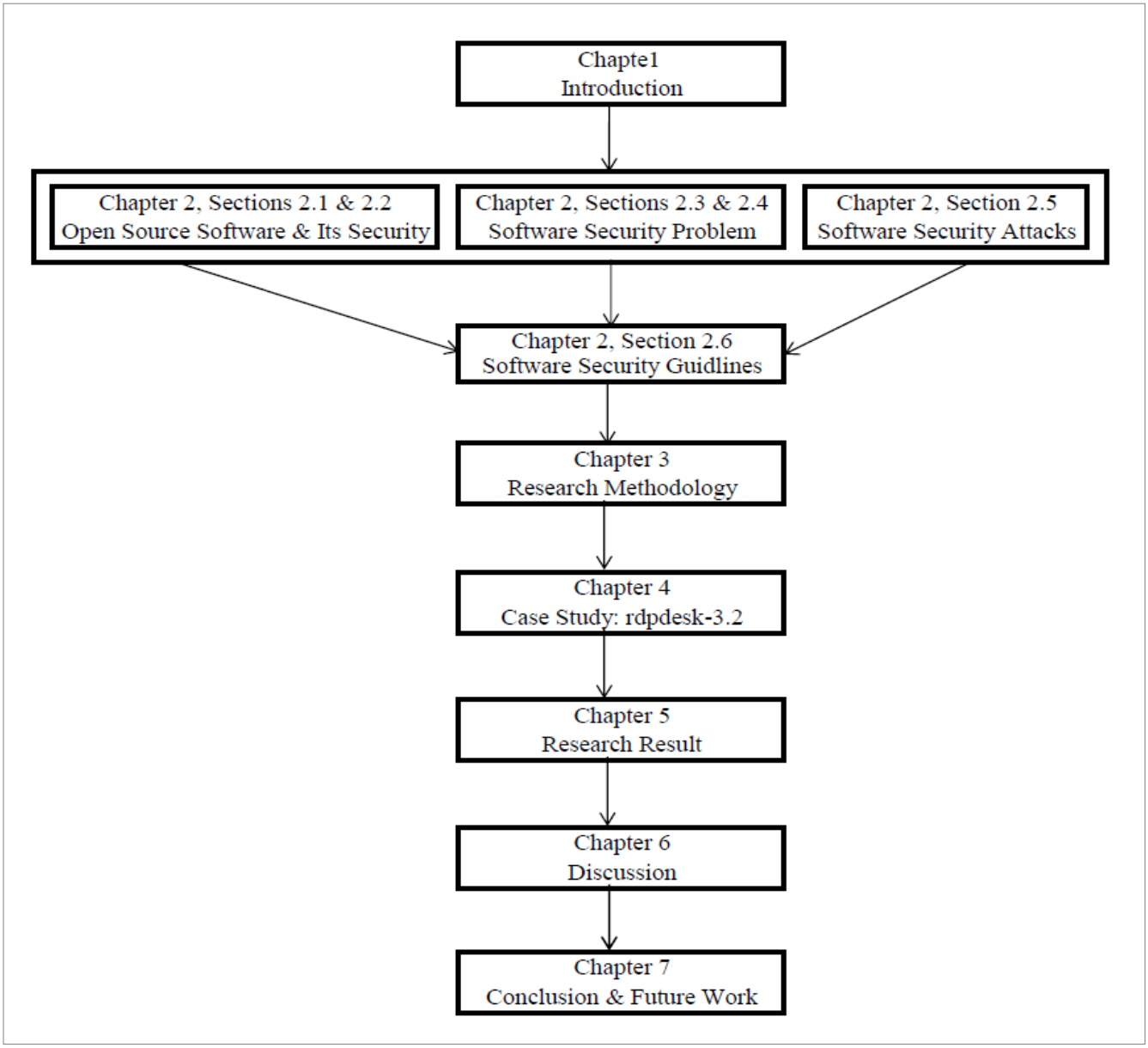


Figure 1.1 Thesis structure

Chapter 2

Background Theory

In this chapter I will present some background theory required for understanding the remaining chapters.
In sections 2.1 and 2.2, I provide a short description of open source software and their security issues. In the security issues of open source software I give an overview of security through obscurity and security through peer review.

In sections 2.3 through 2.6, I provide a short overview of software security problems followed by some common software attacks. And finally I summarize why software security is growing and provide some software security guidelines that need to be followed to have a more secure software.

2.1 Brief Introduction to Open Source Software

Software which is developed with the intention for which its source code is made available for use, study, re-use, modification, enhancement, and re-distribution by the users of that software is classified as open source software [2]. The development of the software relies on public collaboration and it is made available free. The philosophy on which Open Source is based is very similar to that of Free Software Foundation [3].
The licensing of the software permits users to use the software in a way they like. The software can be customized to meet specific needs of a user. Furthermore, a user can design a commercial solution from the software. This derived solution must be distributed to others with the source code. It is also required that the rights associated with the derived version should be extended to any recipient of the software.

One unique characteristics of open source software is that restrictions are not placed on the use, modification and distribution of programs. The distribution terms must comply with the following criteria [4]. The most important issue to consider in connection with open source is licensing. The software has to be licensed as open source. There are different types of licenses [5] one can choose from. The condition is that the license must be approved by Open Source Initiative (OSI) [5].

Generally, the licensing can be categorized into two; namely the GNU GPL and the GNU LGPL. The major difference between the two is that the LGPL places more restrictions on the usage and redistribution of open source software [6].

2.2 Security in Open Source Software

Security is an important aspect and integral part of the phases [91] of any software development. Either in open source or closed source software, trustworthiness solely depends on certain aspects [9] of the software design and development.

Open source software security is the measure of assurance or guarantee that open source software system is free from vulnerabilities that can be exploited. The quality and reliability of open source software can be assessed by reviewing the software's source code after it is made available to the public domain. This reviewing of the source code by several experts from anywhere around the world allow bugs to be discovered and fixed early.

Open source advocates are of the view that when the source code of a piece of software is in the public domain it would be given the same peer review as it is done to theories published in a scientific journals. In other words, they believe that other programmers will review the code for security vulnerabilities, reveal and fix them. By doing so, the number of new vulnerabilities that is introduced and discovered in the software will decrease with time. This theory can be considered nice in the ideal world of open source. The reality is that, there are a variety of factors that affect security of open source software [10]. It is true that the source code is made available to the public. But the questions one will like to ask is whether anyone is reading the code, what level of competence is needed to reveal security vulnerabilities, etc.

However, open source could benefits from reviewing the source code if the people reviewing it are really reviewing the source code with the intention of discovering vulnerabilities for the good of society. In that case the number of new vulnerabilities introduced and discovered in the software will decrease over time [10].

The reality is that, most of the users do not read the source code. They only run and test the software and once it is working they are satisfied (partly because of time constraints, etc.). The assumption is that someone else will do the auditing of the code and so it tends out that only the bad guys have real motivation to analyze it. Another issue is that developers do not have knowledge and qualification to analyze code for security vulnerabilities.

The fact is that open source has made it easier for the bad guys to find vulnerabilities in software. Both good guys and bad guys have equal advantages when it comes to open source software [10].

It is true that the more people review a piece of code, the less likely it is that the code will have a security flaw, a person who is an expert in reviewing code will be more effective than a group of people who are beginners.

Another point of interest that makes the whole issue complex is that it is easy to hide vulnerabilities in software that is complex, little understood and there is no (or little) documentation covering the source code.

2.2.1 Security through Obscurity

Open source advocates and that of closed source have different views when it comes to security. Whiles closed source software supporters believe that keeping the source code secret makes their products more secure, open source software supporters hold on to a contrary view [10].
Open source advocates argue that keeping the source code secret implies to "security through obscurity" [92]. This concept is generally criticized as ineffective in the IT community. Their point of view is that, keeping quiet about once possession just makes it less likely that thieves will target you. However it will not be sensible to leave your doors open at night just because you think that you do not make noise about your properties. In the same way, not making the source code open might deter some hackers; however the large number of successful attacks against Windows and other proprietary software is ample evidence that keeping the source code closed does not provide better level of security.

The interesting point to note in this case is that closed source software does not have any advantage [92] when it comes to using source code to find vulnerabilities. This is because a talented attacker can use tools such as disassembler/decompiler[1] to generate the assembly language of the product. From the above discussions one can realized that having the source code of programs hidden does not protect the program very much. Closed source supporters believe that a vulnerability that exists in software but is not discovered cannot be exploited, thus making the system secure. However, this situation can be likening to a time bomb[2] [8].

2.2.2 Security through View of Various Experts

There have been a lot of discussions in connection with open source software security. Various security professionals have contributed to this crucial topic. Transparency, they say makes systems more secure. The idea is that by allowing people to see the source code, find bugs and then make peer-reviewed changes in the code, the security of the software will be improved.
According to Bruce Schneier, a computer security and cryptography expert, it is important that security engineers have access to the code of open source software. He said, "Demand open source code for anything related to security" [11]. However, not all the security professionals are in agreement with him [7].

1 Disassembler is a computer program that translates machine language into assembly language.

2 Time bomb refers to a computer program that has been written so that it will stop functioning after a predetermined date or time is reached.

One of the experts who have a contrary view to that of open source software supporters is John Viega. He is co-author of Secure Programming Cookbook for C and C++. He is of view that the whole idea is only a type of "cultural elitism" that is coming from the open source community.

According to him, members of open source community belief that they are better at writing code than anyone else. Another expert named Fred Schneider of the Department of Computer Science at Cornell University in New York has this to say: "There is no reason to believe that the many eyes inspecting [open] source code would be successful in identifying bugs that allow system security to be compromised." [7]

2.3 Software Security Problems

Software security is a central aspect of the computer security. Software based security complication such as buffer overflow, integer overflow and the like continues to exist with us for decades. Too often attackers look for software vulnerabilities to hack into computer systems. Furthermore, with software ever increasing complexity and extensibility, internet-enabled software applications are easily targeted to attack [12, 13].

Software security is an understanding of the impact of software-induced security risks and the act of tackling these risks. Good software security practices [14] involves taking into consideration of security in the early stages of the software development lifecycle. This includes understanding of common security problems, identifying and analyzing risks and designing for security.

There are many kinds of software vulnerabilities [15] some examples are, Lack of input validation on user input, Fail-open error handling, buffer overflow and so forth. Description of each problem is out of the scope of this thesis. However, in the following subsections I provide some common software vulnerabilities description.

2.3.1 Buffer Overflow

Buffer overflow takes place as a result of programming error that happens when a program attempts to put data in a memory slot beyond the bound or outside of the memory allocated for the data. The main consequences of writing data outside of the memory allocated includes program crash, data corruption and malicious code execution.

Despite the fact that developers know what buffer overflow vulnerability is, buffer overflow attacks tends to be still common in newly-developed programs [16]. Partly the problem is caused due to the fact that there exist a wide variety of buffer overflow [17] and partly due to erroneous techniques used to prevent them.

In a classic buffer overflow [17] exploit, an attacker inputs data to a program with an undersized stack buffer. The result is that data on the stack is overwritten, including the return address of the function. The data from the attacker sets the return address so that when the function returns, control is transferred to malicious code of the attacker's data.

In C and C++ languages, many functions that manipulate memory do not perform bounds checking. As a result they can easily overwrite the allocated size of the buffers in which they operate. When used incorrectly, even bounded functions can cause vulnerabilities for example strncpy().

There exists many real-life examples of buffer overflows, among them are popular "industrial" applications, such as e-mail servers (Sendmail) [18] and web servers (Microsoft IIS Server) [19].

Here is a piece of code that clarifies buffer overflow:

```
void test(char *s) {
char buf[1024];
strcpy(buf, s);
}
int main(int argc, char **argv) {
test(argv[1]);
}
```

This code is vulnerable to buffer overflow. This is true because the code has a local buffer and the unsafe function [20] *strcpy()* that writes to memory with no bound checking. In this example argv[1] can be of any length, more than 1024 characters and it can be copied into the variable buf. As it is seen it seems simple to find a code vulnerable to buffer overflow. Using some automated tools will facilitate the searching for the vulnerability otherwise it can be searched by focusing on the string based functions like strcpy(), strcat() or gets() and then see if these functions operate on local buffer. If searching succeed in finding these functions there is high possibility of redirecting execution control to flow to anywhere you want. Another method would be trial and error that is by feeding a large input data to the space allocated during execution.

Poor input validation is the main reason for Buffer overflow vulnerabilities. An attacker exploits this vulnerability to run his code in the victim and control the victim's machine. So care must be taken to validate inputs to shut the doors that enable attackers to do their job. Here are some of the techniques used to prevent buffer overflow: Non-executable stack, Static Analysis, Dynamic runtime protection and using safer versions of functions.

2.3.2 Integer Overflow

An integer overflow is another programming error that is commonly seen in programs. This condition happens when a not properly sanity checked integer is used in the determination of an offset or size for memory allocation, copying, concatenation, or similarly. If the integer in question is incremented beyond the maximum bound, it may fold to become a very small, or negative number, and thereby providing wrong value.

I will illustrate this problem by taking a simple example. Let's consider an unsigned

char of 8-bit integer types.

Unsigned char a = 237;

Unsigned char b = 60;

Unsigned char c = a+b;

When executed the code gives the following result

(a) 11101101 +

(b) 00111100

(c) 100101001

The sum of the two variables *a* and *b* gives a 9-bit integer result. However, since *c* is an 8-bit variable, the left most bit is cut off from the result variable *c* and the result is 237 + 60 = 41 instead of 297.

Some of the main consequences of integer overflow include; Availability, Integrity and Access control. For details see [21].

The main problem with Integer overflow is the difficulty of detection when it happened. So an application gets real difficult to tell if a result calculated is correct or not. This can becomes very serious if the calculation has to do something with the size of a buffer or array index. Generally speaking Integer overflow is not directly exploited. And this is because it can not directly overwrite the memory, but sometimes this can open the door for another bug[3] "buffer overflow" which can be exploited by an attacker directly. Integer overflow can be prevented by using SafeInt library. For details on how to use SafeInt for the purpose of integer overflow prevention see [22].

2.3.3 Format String Problem

Format string problem is another programming error that allows an attacker to control a function's format string. This problem occurs due to; data entries from untrusted sources to an application or

3 Bug is a fault or defect in a computer program, system, or machine.

data passing as arguments to the function's format string. Format string problem was identified as software vulnerability in the year 1999. This problem mainly occurs in the C/C++ language [23]. The exploits of this problem ranges from crashing a program, reading and writing from/to the stack and executing intentional codes as well. The main consequences of these exploits are confidentiality[4] (information disclosure) and access control[5] (execution of arbitrary code).

To understand this problem let us see a couple of examples [24]. In the first example, an attacker will be able to read data from the stack and in the second, he will write data to the stack.

Example 1: Reading data from stack

```
/* read_stack.exe */
#include <stdio.h>
int main(int argc, char* argv[])
{
if(argc > 1)
printf(argv[1]);
return 0;
}
```

Here the intention is to display the content that is entered by the user at the command prompt. Let's see a run of the code with the input "Hello World".

C:\read_stack.exe "Hello World"

The output is a string 'Hello World'. That is what we expected from the execution of the above line.

What if we try to run this one?

C:\test.exe "%X %X"

The output should be a string like this '%X %X'. However, this time we get '12ffc0 4011e5' printed into the screen instead of '%X %X'. So what is wrong? In fact the program did what was asked to do. The %X specifier is meant to read the stack four bytes at a time. So the first string of the output 0x12ffc0 is the address of the stack location while the second output string 4011e5 is the return address of the *main()* function. From this example it can be easy to understand that if a confidential data is stored in the stack then it is easy for an attacker to read it.

Example 2: Writing to the stack.

```
/* write_stack.exe*/
#include <stdio.h>
int  main(int argc, char **argv)
```

4 Confidentiality: Discretion in keeping secret information.

5 Access control: refers to security features that control who can access resources in a system.

```
{
char buf[100];
int x;
if(argc != 2)
exit(1);
x = 1;
snprintf(buf, sizeof buf, argv[1]);
buf[sizeof buf - 1] = 0;
printf("buffer (%d): %s\n", strlen(buf), buf);
printf("x is %d/%#x (@ %p)\n", x, x, &x);
return 0;
}
```

This code formats any value passed to it from the command line prompt into fixed length buffer. After formatting, the buffer is printed into the screen. So let's give it a try.

C:\write_stack.exe "Hello World"

The output of the code is as follows:

buffer (11): Hello World

x is 1/0x1 (@0x804745c)

The program formatted the string into fixed-length buffer and prints the length of the buffer and the content in it "Hello World". Moreover the value of x is displayed both in decimal and hexadecimal in the second line. 0x804745c is the memory address where x is stored in the memory.

Now let's do the bad stuff and act like an attacker. We will see how to write anything into the memory. For this purpose, we modify the code a bit by adding the following line just before the return point.

printf("123456%n", &0x804745c);

Now let's run the code one more time with this input:

C:\write_stack.exe "Hello World"

The output will be something like this:

buffer (11): Hello World

x is 1/0x1 (@0x804745c)

The result is the same as before, but this time additional thing has happened. Since we knew the memory address of x, the piece of the printf code we added overwrote the value of x from its initial value 1 to 123456. Imagine what more attacks one can do. It is possible to overwrite an arbitrary

memory location and corrupt some useful data stored. Even it is possible to overwrite the return addresses and function pointers to point to some chosen address.

To prevent format string problem, First User input passed directly to a formatting function must be avoided. And secondly the format strings used by an application should only be read from trusted sources. This ensures that attackers cannot control the path to the strings.

2.4 Software Security Attacks

Computers are main targets of external attacks aiming at taking control over software behavior. In general, such attacks make their way to programs through a communication channel as data. Once the attacks arrive into the program memory, they activate existing vulnerabilities. Finally, these attacks overturn program execution and gain control by exploiting the program flaws [25].
The overall effects of these attacks are the reasons for the challenges in computer security. As a result, many mechanisms [27] have been proposed for defending against these attacks. Such mechanisms include static code analysis, dynamic code analysis and the likes.

Gray McGraw classified attacks into four basic categories [26]; configuration attacks, attacks on implementation defects in systems (aka bugs), attacks on design and architecture defects in systems (aka flaws), and attacks on confusion surrounding trust. For details on these categories see [26].

2.4.1 Buffer Overflow Attacks

Buffer overflows are the most-loved exploit that hackers look for. This is mainly due to the fact that an attacker takes control of a program expecting user inputs by exploiting some of the program flaws. Generally, there exist two types of buffer overflow attacks. These are stack[6]-based attacks and heap[7]-based attacks. This classification is according to Brien M. Posey [28]. For details on heap-based buffer overflow and stack-based buffer overflow see [29, 30].
Heap-based attacks are attacks that overflow the memory (space) reserved for a program, however, due to the difficulty involved in performing such an attack, this type of attacks happen rarely. On the other hand, stack-based buffer overflow attacks are the most common ones.

6 A stack in computing architectures is a region of memory where data is added or removed in a last-in-first-out manner.
7 A heap is a general term used for any memory that is allocated dynamically and randomly; i.e. out of order.

2.4.2 Cross Site Scripting (XSS) Attacks

Cross-Site Scripting attacks are attacking techniques which are most common in the application layer [32]. In these attacks, malicious scripts are injected into trusted web sites. For these types of attacks to occur, an attacker uses a web application to send malicious script to a different an end user, usually in the form of a browser side script. When a web application uses an input from a user for the output it generates without validating it, this condition opens a door for allowing flaws that lead to XXS attacks. According to Acunetix web application security, in general, cross-site scripting attack is defined as "*A hacking technique that leverages vulnerabilities in the code of a web application to allow an attacker to send malicious content from an end-user and collect some type of data from the victim.*" [8]

Putting it differently, when an attacker sends a malicious script to a non suspicious user, the user executes the script. As a result, the malicious script can access any cookies, session tokens, or other sensitive information (including credit card numbers, social security numbers and even medical records) maintained by the browser and used with that site. These malicious scripts are even capable of rewriting the content of the HTML page.

In general, there are two categories of XXS attacks [31]. These are stored XXS attacks and reflected XXS attacks. For details on stored and reflected XXS attacks go through [31].

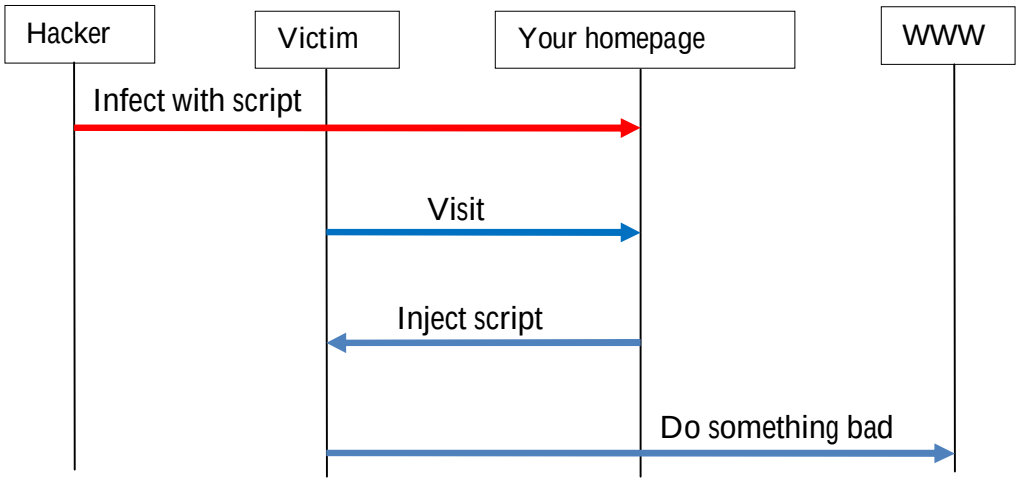


Figure 2.1 High level view of XXS attack

2.4.3 SQL-Injection Attacks

SQL Injection attacks are other types of attacks that are most common in application layer. These types of attacks exploit the improper coding of web applications vulnerabilities thereby allowing

8 "Cross-site scripting attacks" Acunetix web application security.

hackers to inject SQL commands into say a login form to allow them to gain access to the data held within a database. Once an SQL injection exploit is successful, an attacker can read sensitive data from the database, modify database by (Insert/Update/Delete) data, even he/she can execute administration operations on the database [33, 34].
Website features which are susceptible to SQL Injection attacks include; login pages, support and product request forms, feedback forms, search pages, shopping carts and the likes. This is mainly due to the fact that the fields available for user input in these features allow SQL commands to query the database directly.

Generally speaking, main consequences of SQL-injection attacks are; Confidentiality, Authentication[9], Authorization[10] and Integrity[11] problems. Prevention mechanism for these types of attacks includes; input validation, binding variables, securing functions and Error messages [35].

2.4.4 Denial-of-Service (DoS) Attacks

A denial-of-service (DoS) attack is an attack in which an attacker attempts to make a computer or network resource inaccessible and prevent users from accessing information or services. For example, using this attack mechanism, an attacker can prevent you from checking e-mail, websites, online accounts (banking, etc.), or other services that run or depend on the affected computer.
The most common and obvious type of DoS attack occurs when an attacker "floods" a network with external communications requests. When you type a URL for a particular website into your browser for example ‘www.yahoo.com’, you are requesting the site's computer server to view the page. The server can only process a certain number of requests at once, so if an attacker is able to overload the server with huge number of requests, it can't process your request. This is what "denial of service" is. The server rejects/denies your request because it is flooded with huge requests from the attacker.

(US-CERT) defined some symptoms of denial-of-service attacks to include [36]:

- Unusual slow network performance
- Unavailability of a particular website
- Inability to access of a website
- Dramatic increase in the number of spam e-mails received

An attacker can perform an attack on your email account by using spam e-mail messages. When you create an e-mail account, you are assigned a specific storage on the e-mail server of the account provider, this limits the amount of data you can have in your account at any given time. An attacker can

9 Authentication is the process of determining whether someone or something is, in fact, who or what it is declared to be.
10 Authorization is the process of giving someone permission to do or have something.
11 Integrity is a concept of consistency of actions, values, methods, measures, principles, expectations, and outcomes.

consume your quota by sending many, or large, e-mail messages to the account, preventing you from receiving legitimate e-mail messages.

2.5 Why is Software Security Problem Growing?

Nowadays, as businesses and societies have increased their dependencies on computing devices for their daily usage, software are getting developed almost in huge numbers. As there is no such an issue as perfect software however, most of the computing systems are susceptible to software security problems. From the above statements we can see the number of software (packages) developed have direct influence to the growth of software security problems. But is this the only reason why software security problem is growing? In the following subsection we will see three trends that have large influence on the growth of the problem [12, 37].

2.5.1 Connectivity

Thanks to the internet today it is easy to connect computers and establish communication among them. Due to the ease of connectivity and communication that is because access through a network does not require human intervention, the number of attacks and ease of these attacks has also increased. This issue places software at greater danger. Lots of businesses, government sectors and mass people have increased their dependencies on networked communication to interact with each other. Sadly, as these systems are connected to the Internet, they become more susceptible to software-based attacks from various distant sources. No need for an attacker to physically assess the system in order to exploit vulnerable software.
The omnipresence of networking means that there are more software systems to attack, more attacks, and greater risks from poor software security practices than in the past. This will even become worse over time. Because poor coding and design problems will still continue to exist, the Internet is everywhere so the attackers are now roaring at your doorstep [37].

2.5.2 Extensibility

A second trend that affects software security problem growth is system extensibility. Software extensibility is a system design principle in which the system is designed to include plug-in architectures and mechanisms for expanding/enhancing the system with future capabilities and new functionalities (features) without having to make greater changes to the system infrastructure. For example, Today's operating systems support extensibility through dynamically loadable device drivers and modules.

Today's applications, such as word processors, e-mail clients, spreadsheets, and Web browsers, support extensibility through scripting, controls, components, and applets.
From an economic point of view, extensible systems are attractive because they provide flexible interfaces that can be adapted through new components. In today's businesses, it is essential that software be distributed as rapidly as possible in order to gain market share. However at the same time the businesses also require that applications provide new features with each release/patch. With an extensible architecture, these demands are easily satisfied by releasing the base application code early, and then releasing later feature extensions as needed.

Unfortunately, despite of the mentioned attractiveness of extensible systems, it is hard to prevent software vulnerabilities from been released as unwanted extensions [37].

2.5.3 Complexity

A third trend that influences software security growth is the growth in the size and complexity of software systems. For example a computer running Windows vista and associated applications consists of at least 50 million lines of code[12] [38], and end-user applications are becoming increasingly large and even more complex. When systems become this large and complex, it is unthinkable to avoid bugs.
The figure below shows the growth in complexity of Windows (as measured in lines of code) over the years. The point of the graph is to show the growth rate of the lines of code over time.

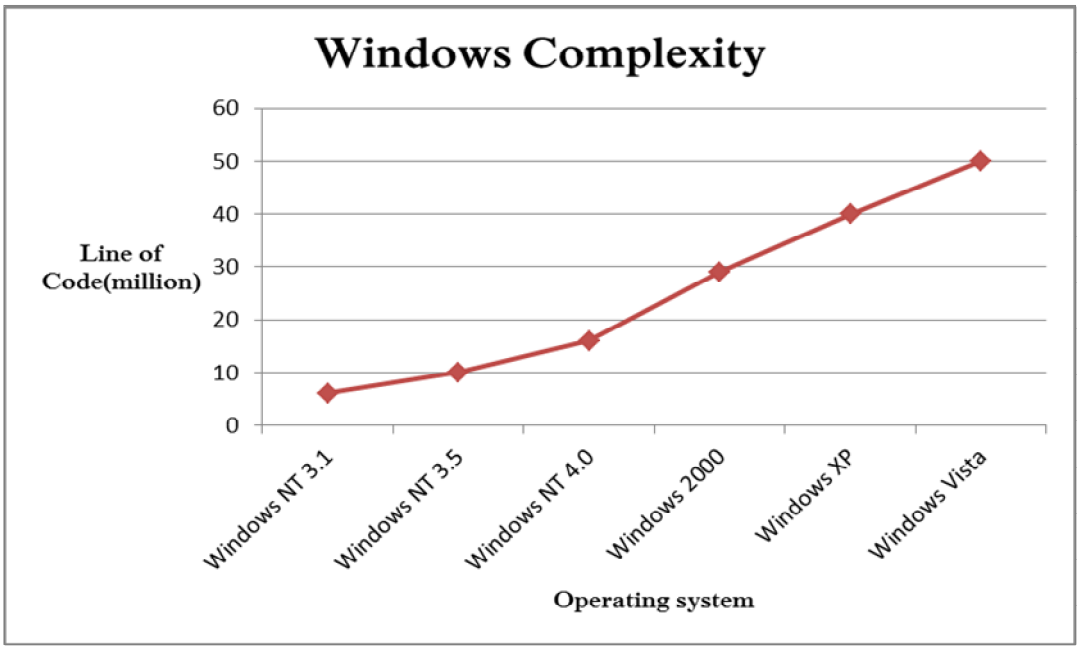


Figure 2.2 Windows Complexity (measured in Line of code)

12 Line of code is a popular definition of program size.

According to [39], it is estimated that in 1000 lines of code there exist an average of 5 bugs. This shows us how the number of bugs increased over the years with the increasing size and complexity of software. Besides the lines of code, other factors like, whether the code is tightly integrated, the overlay of patches and other post-deployment fixes, and critical architectural issues have significantly affected complexity [38].

The tendency for software systems to grow very large quickly is just evident in open source systems compared to Windows. As a result, the problem is that more code results in more defects and, in turn, more security risks.

2.6 Software Security Guidelines

From day to day, attacks are becoming more sneaky and sophisticated. As a result, they are creating a complicated and dynamic risk environment for IT-based operations. Due to these issues, users' concern about the software integrity, security and reliability have greatly increased.

To deal with these users' concerns and adhere to customer requirements, vendors and product providers must take substantial efforts focusing at reducing products' vulnerabilities, enhancing attack resistance and protecting integrity. The efforts used to guarantee the security of the above mentioned concerns are often denoted as "software assurance." Software assurance is particularly an essential factor for public safety and economies, critical information of organizations and national security. In other words these users require a high degree of software security confidence. This confidence is only achieved by using best practices for developing secure software [9, 14, and 41]. According to Todd Landry OWASP, SANS institute, MITRE, PCI Security Standards Council and SEI are the top five leading organizations when talking about software security guidelines. A concise overview of the top five leading organizations is found at [40].

In the following subsections I provide some guidelines used to ensure software security.

2.6.1 Secure the Weakest Link

Viega and McGraw [42] described secure the weakest link by saying *"Security practitioners often point out that security is a chain; and just as a chain is only as strong as the weakest link, a software security system is only as secure as its weakest component. Bad guys will attack the weakest parts of your system because they are the parts most likely to be easily broken. (Often, the weakest part of your system will be administrators, users or tech support people who fall prey to social engineering.)"*(p, 93-96).

Attackers have more tendencies to launch an attack by penetrating through weak sides in a software system than investing their time and effort to penetrate into a heavily strengthened system. For

example, some carefully designed cryptographic algorithms requires a great deal of time (usually years) to break. Attackers are not stupid, so probably they will not spent years trying to attack encrypted information communicated in a network. Instead, they prefer to target the termination points, could it be server/client or the likes, as the spot may be easier to attack.

The weakest spot of a system can be software or sometimes it can be surrounding infrastructures (components) that interact with the software. Therefore it should be a common practice for software venders to be sure as whether the weakest links (parts) of their products are secure enough before they release their products to end users. Doing so plays a great deal in closing some doorsteps to attackers [43, 44].

2.6.2 Defense in Depth

In an application, layering security defenses into multiple layers play an important role in reducing the chance of a successful attack. Defense in depth is an organized use of layered security defense used to protect the integrity of the information. In other words Defense in Depth is a principle which helps in maintaining a secure infrastructure by adopting a layered approach, i.e. defending against various threats at various levels. The scheme uses the principle from military defense that it is more difficult for an enemy to defeat a multi-layered military defense system than to penetrate a single barrier [9, 45]. The probability of an attack to succeed is minimized by building a well-structured Defense in depth. Building such a strategy also helps system administrators and security personnel to identify people who attempt to compromise their system. For example, if an attacker penetrates and gains access to a system, the system administrators will have enough time to deploy new countermeasures to prevent adverse impact from occurring if the system security is built based on the defense in depth strategy.

The Defense in Depth strategy should include all the possible countermeasures that need to be taken. Some of the components include Antivirus software, firewalls, anti-spyware programs, hierarchical passwords, intrusion detection, biometric verification and the likes. In addition to the above listed components, physical protection of business sites along with comprehensive and ongoing personnel training enhances the security of critical data against compromise, theft or destruction. All in all the main idea of Defense in Depth is to ensure the existence of a well-designed robust defense strategy in shelf. A detailed discussion of the topic is found at [46].

2.6.3 Fail Securely

Software or system crashes can happen any time by any means. For example attackers try to make it crash in order to expose several potential vulnerabilities in it during startup. So security professionals

and developers should design the system architecture in such a way that ensures failing securely when a component in a system fails. This concept is called “Fail Secure”.
The main idea here is, when a system fails, it should fail securely not giving any footprints which an attacker can start with. Normally this involves things like: secure defaults, restoration to a secure state, always checking return values for failure, existence of a default case that does the right thing and the likes. Assess all stuffs that could happen during system failure and be sure that it does not threaten the system [42, 47, and 48].

2.6.4 Use the Least Privilege

The principle of list privilege which sometimes called principle of minimal privilege is giving users or components the least access privilege required to accomplish its gob. In other word this is to say when a subject requires accessing a resource, only the lowest access rights should be assigned to it. And the access right shouldn’t stay long duration it should be in effect for a short period. A user with access right beyond the necessary right may change information in unwanted ways. Therefore, carefully delegated access rights play major role in limiting attackers from compromising a system.
For example, let's say you were to go on summer vacation, and ask one of your neighbors just to collect mail for you. No matter how close you are with your neighbor or how much you trust them you don’t put yourself at risk by giving them the key to your house. Only the key to your mail box will be enough. So this way you your mails will be collected without your house been set to risk [49].

2.6.5 Compartmentalize

Compartmentalization (multilateral) is the restricting of information flow across security levels. The idea is that if details of a system is kept or known to few people only, the risk probability of compromising that system is decreased. In information security, the concept of compartmentalization originated in the managing of classified information in military, intelligence agencies and the likes.
The principle of compartmentalization are used to impose the need to know principle, that is a person is only allowed access to the information if it is a must to know to do his task. If the person does not need to know the information in order to perform his task, then compartments can be enforced to limit the information that person can access [39, 51].

In compartmentalized system information-control boundaries need to be vertical as shown in figure 2.3 rather than horizontal as it is used in the Bell-LaPadula (multilevel) model [50] figure 2.4.

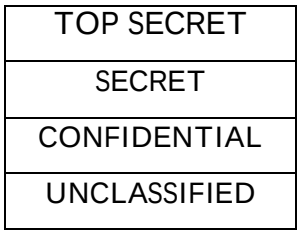


Figure 2.3 Bell-LaPadula model

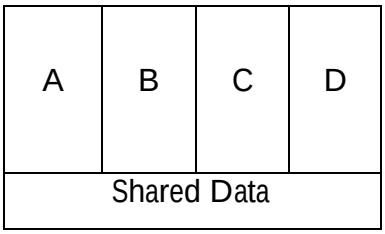


Figure 2.4 Multilateral (compartmentalize)

2.6.6 Keep it Simple

Generally speaking, complex systems tend to introduce more bugs. For instance if we consider a very simple Web browser and start adding more and more features to it the more we add new features, the more complicated it gets, and the harder it is to make it secure. Naturally, complex systems (systems that involve software) introduce multiple risks. One such risk is that due to the extensibility or patching of new features or fixes, malicious functionality can be easily added to the system. Sadly, complexity of the system lets malicious components to remain invisible to non-suspicious users until it is too late. Another risk is that complexity in a system makes it hard to understand, analyze and secure it. According to Gray McGraw et al., it is difficult to achieve security even when the system is simple and complexity makes security even more difficult. Security risks can remain uncovered in complex system until the areas have been exploited [52].

Chapter 3

Research Methodology

In this project two research approaches are used. The first method is static source code analysis technique and the second one is threat assessment modeling technique. The choice for the first technique is mainly because static source code analysis are normally similar with code auditing to the extent of concentrating on the actual source code, but instead of auditing the code manually, it uses automated tools. In this method, the auditing is mainly based on the use of gained knowledge and known issues more effectively and focuses the attention on areas of the code more likely to be vulnerable.

However, despite of the above advantage, the tools used in static source code analysis method might sometimes generate imprecise analysis report, that is either they generate some false negatives[13] or some false positives[14]. This means that the tools can hide existing vulnerabilities or sometimes they suggest non existing vulnerabilities on the code. Moreover, static analysis tools also have some limitations, especially in identifying design and architectural vulnerabilities. Therefore in order to have higher confidence on the security of an application, I introduced the second method, threat assessment modeling, to further explore and identify all possible potential threats including those that were not discovered using the static source code analysis tools. The developed threat model together with the result of the static analysis can then be used to build a solid security strategy to guard against the outlined threats to an application.

3.1 Static Source Code Analysis Technique

Software quality with respect to security is crucial. Different techniques are used to improve the quality of software. Static source code analysis is one of such techniques. It is the process of analyzing a program without executing it. Static analysis techniques are used primarily in the areas of software metrics, quality assurance, program understanding and refactoring [17]. Several static code analysis tools are developed in the recent years. With the help of these tools one can easily evaluate a program. Static source code analysis tools are used in the security arena for vulnerability detection. These tools operate by checking specific vulnerable code patterns in a program or it checks if a program control flow reaches some state of vulnerability. And up on finding either one of the two program flaws the

13 False negative refers to a result that tells you a condition is not present, when in reality, is there.
14 False positive refers to a result that tells you a condition is present, when in reality, is not there.

tools report for bugs. Alexander Ivanov mentioned in his paper [17] "*Most tools rely on a static rule set of predetermined code patterns that they should detect and report.*" This means that due to their nature, not all vulnerabilities can be detected by static analysis tools. So if a bug is not reported after an analysis by a tool it doesn't mean the code is free of bug. The recommended procedure is to use various tools and detect as much bugs as possible. Several approaches are used to static analysis some of the most popular approaches are pattern matching, lexical analysis, parsing and AST analysis. For details on the approaches see [17]. Listed below are both the advantages and limitations of static source code analysis.

Static source code analysis advantages:

- It can find weaknesses in the code at the exact location.
- It can be conducted by trained software assurance developers who fully understand the code.
- It allows a quicker turn around for fixes.
- It is relatively fast if automated tools are used.
- Automated tools can scan the entire code base.
- Automated tools can provide mitigation recommendations, reducing the research time.
- It permits weaknesses to be found earlier in the development life cycle, reducing the cost to fix [53].

Static source code analysis limitations:

- It is time consuming if conducted manually.
- Automated tools do not support all programming languages.
- Automated tools produce false positives and false negatives.
- There are not enough trained personnel to thoroughly conduct static code analysis.
- Automated tools can provide a false sense of security that everything is being addressed.
- Automated tools are only as good as the rules they are using to scan with.
- It does not find vulnerabilities introduced in the runtime environment.

For further reading on static source code analysis it is recommend reading [54].

3.1.1 Pattern Matching

Pattern matching is the process of examining some sequences for the presence of the same patterns. Pattern matching is the simplest technique used in static analysis. A common way to do code auditing is to use the grep utility to see the existence of the pattern 'strcpy'. Often the function strcpy() is misused and existence of this function is a good indication for a potential vulnerability. However, this doesn't mean that all calls to strcpy() is dangerous. Some calls could be safe. In this case, proper treatment

should be carried to the unsafe calls only. Pattern matching technique has some problems. These include:

- Not capable to detect complicated vulnerabilities
- May generate very large false positive results
- Lack of proper C parser
- Incapable of telling comments from real code

3.1.2 Lexical Analysis

Lexical analysis is another technique used in static source code analysis and it is the process of reading the characters of the source code from left to right discarding whitespaces and grouping them into tokens. Compared to pattern matching, lexical analysis provides slight improvement. In lexical analysis, the tokens developed by grouping the source code are matched against a database of known vulnerability patterns. This technique is used by tools like Flawfinder, RATS and ITS4 [55]. I will present Flawfinder and RATS in sections 4.2.1 and 4.2.2 respectively. Lexical analysis improves the result of pattern matching further by handling irregular whitespace and code formatting. However, the number of false positives is still very high.

3.1.3 Parsing and AST Analysis

To further improve the performance of static source code analysis, the next step to do is parsing the source code and building an Abstract Syntax Tree (AST). Normally this task is performed by the compiler and provides a possibility for code reusability. In order for a static source code analysis tool to parse and analyze various programs correctly, the tool's parser must be compatible at least with one of the significant compilers.

The abstract syntax tree (AST) captures only the important nodes in a tree form, while neglecting unnecessary syntactic details. The main distinguishing factor of ASTs from concrete syntax trees is they omit tree nodes that represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments. The abstract syntax tree (AST) allows analyzing the syntax and the program semantics. One of the drawbacks of lexical analysis tools is the inability to distinguish the difference of a variable and a vulnerable function that have the same name, but this is not a problem in AST. This tool can accurately distinguish the difference. AST reveal vulnerabilities hidden from lexical analysis tools by expanding complicated expressions. The pattern matching approach discussed in section 3.1.1 can be largely improved by using matching AST trees instead of sequences of tokens or character [17, 56].

3.1.4 Type Qualifiers

Type qualifiers are qualifiers that are used to qualify types of an identifier. They give an identifier one of two properties. The **const** type qualifier defines an immutable (non-modifiable) object while volatile type qualifier declares a mutable (modifiable) object.
Languages like Java, C/C++ and the likes have few number of type qualifiers [17]. Jeffrey Foster [57] developed a framework for type qualifiers that are used by some advanced vulnerability detection tools. Foster, in his work, proposed a general purpose system for adding light weight user-defined type qualifiers to improve the software. According to Foster this is done by annotating the source code and detecting type inconsistencies by type qualifier inference. Type qualifier systems possess several advantages [59].

Tools like cqual [58] require only a few type qualifier notations in order to be added to a program and the type qualifier inference is efficient even if the programs are large. The main disadvantage of this approach is that it is not applicable to large numbers of vulnerabilities that can be expressed in terms of type inconsistencies.

3.1.5 Data-flow Analysis

The analysis discussed so far are good at analyzing simple vulnerabilities like format string problem. However, vulnerabilities like buffer overflow, integer overflow and so forth, require some more complicated analysis. As seen in sections 2.3.1 and 2.3.2 respectively, such vulnerabilities arise when variables are assigned values larger than the buffer. Detecting such problems in a program without executing the program is quite hard. Data-flow analysis (technique for gathering information about the possible set of values calculated at various points) is another technique which is used for solving problems of these kinds. It solves the problem by determining the parts of the program to which a variable with a particular assigned value might propagate.

3.1.6 Taint Analysis

Taint analysis is a technique which tracks incoming data from untrusted source and marking them ‘taint’. Normally data originating from outside should be treated as untrusted, and therefore marked ‘tainted'. When these data are used in operations an eye is kept on, and taint flags are broadcasted to the result of such operations. When data marked as tainted is used in an operation, for example as target address in a jump, an alarm is raised and essential action is taken. An attempt made to use tainted data from outside sources without performing relevant action is an indication of vulnerability.

Results obtained from taint analysis can be very accurate; however this has an effect to the performance due to the fact that a lot of emulation is involved [17, 60, 61, 62].

3.2 Threat Risk Modeling Technique

Threat risk modeling is a technique used to develop a model by iteratively assessing the vulnerabilities in an application to find those that are most dangerous and thereby creating a prioritized set of countermeasures to measure and contain the risks. The model developed using this technique provides desirable recommendations that greatly maximize and insure the protection of confidentiality, integrity and availability without affecting the overall functionality of the application. Normally threat risk modeling is developed as a collaborative process between the various organizational levels. A model developed without the interaction of the organizational levels can result into an inefficient security measure and the cost can be high too [64, 65, 68, 69].

Threat risk modeling is used to assist management in allocating time and money that should be spent in order to contain the risks [63]. Threat risk analysis is divided into two types, the quantitative and the qualitative. The first analysis is a mathematical approach and the second is a high**/**medium**/**low approach. In this thesis I use the qualitative approach.

The central areas in threat risk modeling are:

- Identify Security Objectives
- Survey the Application
- Decompose Application
- Identify Threats
- Identify Vulnerabilities

Figure 3.1 shows the steps involved in performing threat risk modeling.

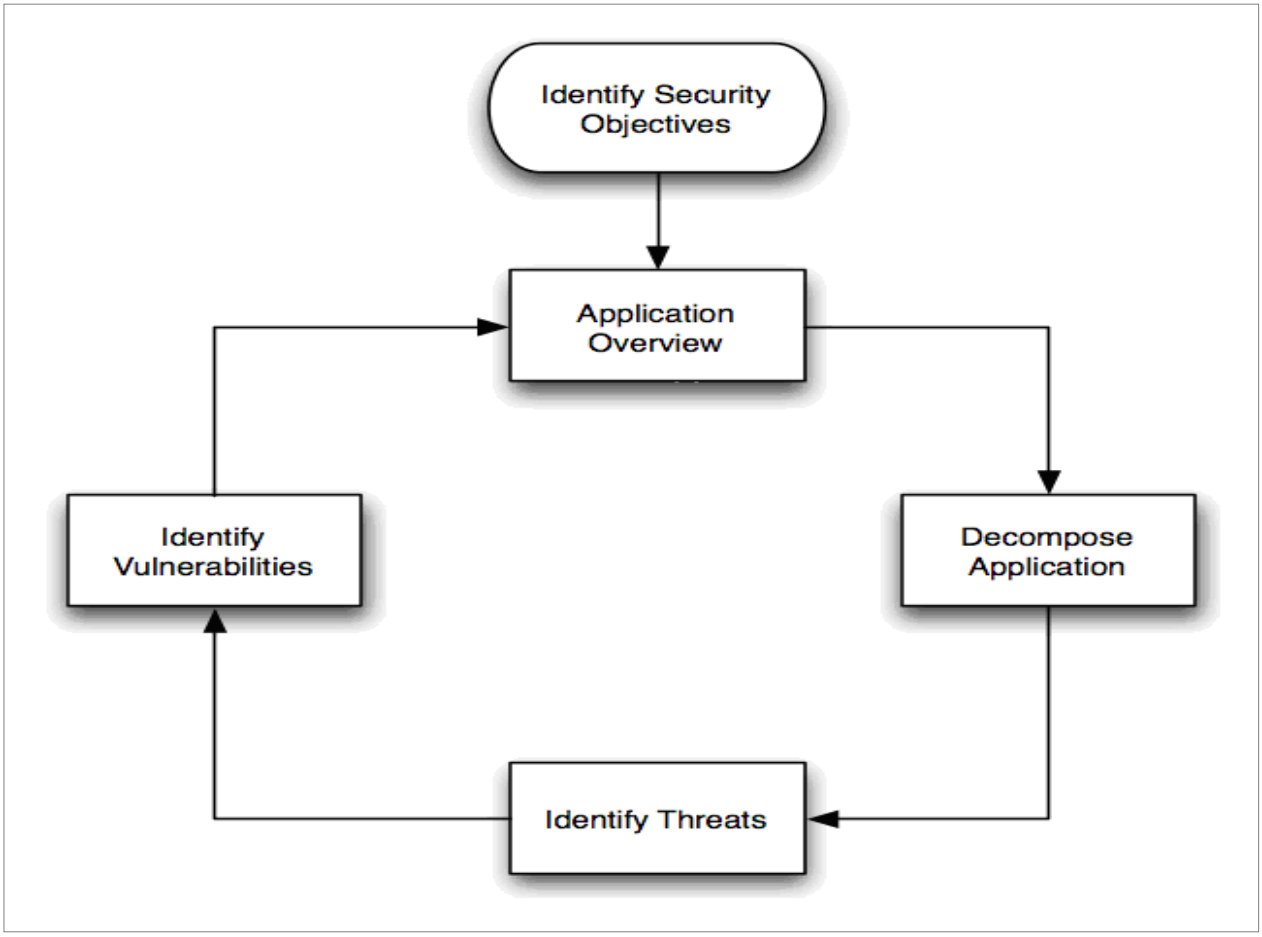


Figure 3.1 Threat risk modeling steps

3.2.1 Identify Security Objectives

When deploying open source application into your system, it is best to define security objectives and requirements early before you integrate the application in the system. These Security objectives are goals and constraints that have influence to the confidentiality, integrity, and availability of your data and other applications of the system. Defining security objectives helps to determine where to center your efforts and to understand the potential attackers on the application as well. Identifying security objectives is done iteratively by examining the application's requirements and usage premises.

In threat risk modeling, Identification of security objectives is the first step that one must do in order to protect the security of an application. Once the security objectives are defined, they can be used as the basis for the subsequent steps. However, this doesn't mean that these objectives should remain static; they can change with time and with new situations that may arise in the process. Any changes in the security objectives have an influence to other security activities. As a result, you should review your threat risk model now and then with every change happening to the objectives.

To alleviate this process, the following security objectives categories can be used as a starting point:

- **Identity:** check whether the application protect user identity.
- **Financial:** Assess the financial loss the organization can face in remediation.
- **Reputation:** Measure/estimate the loss of reputation that can happen when the application is abused.
- **Privacy and Regulatory:** The extent of protection of user data?
- **Availability Guarantees:** what is the availability requirement of the application per a ***Service Level Agreement (SLA)*** or similar guarantee?

Such a list is an exhaustive, but it provides some knowledge about the decisions that leads into selecting and building security controls [12, 65].

3.2.2 Survey the Application

After the security objectives have been identified, the next step is to have an overview of the application. Surveying the application is done by analyzing the application design in order to identify the components, data flows, and trust boundaries of the system. In other words this step helps in the identification of the application's key functionality, characteristics, and clients which in turn will help in identifying relevant threats that will be used in the steps that follow [65].
Following are some points used to create an application overview:

Draw end-to-end deployment scenario: sketch a diagram that depicts the components, structures, subsystems, deployment characteristics, authentication, authorization, and communication mechanisms of the application.

Identify roles: identifying the roles helps to determine both things that are supposed to happen and things that are not supposed to happen. For example, identify who can read data, who can update data, who can delete data?

Identify key usage scenarios: Identify the important features of the application. Avoid listing every use case. Instead, list the main ones.

Identify technologies: list the technologies and key features of the software. Some examples include operating system, development language and so forth. Identifying technologies helps in focusing on technology-specific threats.

Identify application security mechanisms: Identify any key points of the following; authentication, authorization, input-validation, cryptography and the likes.

3.2.3 Decompose the Application

In this step, break down the application to identify trust boundaries, data flows, entry points, and exit points with a security impact that need to be evaluated. Generally speaking, it is easier to reveal threats and discover vulnerabilities when the modules of the application become more known.

Identify trust boundaries: identifying an applications trust boundary helps focusing the analysis on specified areas of interest. Trust boundaries are those points where access control levels change in order to access resources or an entry points in applications where data passing those point are not fully trusted.

Identify data flows: trace data flows through the application starting from entry to exit. This makes clear how the application interacts with the external system and also clarifies how internal components interact.

Identify entry points: attackers use the entry point of the application to hack into the system. This point is intended for clients. Identify where these entry points are and the type of data they take.

Identify exit points: these are points where the application sends data to the client or to external systems.

3.2.4 Identify Threats

In this stage, identify the threats and attacks that might compromise the security objectives of the application. These threats are the bad consequences that could happen to the application. Generally it is difficult to identify unknown threats therefore concentrate on known threats. To perform this process, one can use the following basic approaches [65].

Start with common threats and attacks: start by preparing a list of common threats and attacks and then apply the threat list in your application. There is a possibility that some threats are simply eliminated because they are not applied in your application.

Use of question-driven approach: this helps in identifying relevant threats and attacks. For this approach a STRIDE model [66] can be used to ask questions that are related to the application.

Examine the application level by level, layer by layer and feature by feature when identifying threats. And while performing threat identification, concentrate on areas where security mistakes are most frequently made. Note that, threats identified during this stage do not necessarily indicate vulnerabilities.

3.2.5 Identify Vulnerabilities

In this step, review the application security clearly looking for vulnerabilities. Follow the same process as done while identifying threats in the previous step. However, this time sample questions presented must be designed in a way to help identify vulnerabilities instead of threats. A good approach in identifying vulnerabilities is to examine the application layer by layer looking at every vulnerability categories in each layer.
The following are some of the vulnerability categories that might help in the process of identifying vulnerabilities [67]:

- Identify authentication vulnerabilities
- Identify authorization vulnerabilities
- Identify input and data validation vulnerabilities
- Identify configuration management vulnerabilities
- Identify sensitive data vulnerabilities
- Identify session management vulnerabilities
- Identify cryptography vulnerabilities
- Identify parameter manipulation vulnerabilities
- Identify exception management vulnerabilities
- Identify auditing and logging vulnerabilities

Note that, vulnerabilities are present in an application when threats exist in the application and are not mitigated [70]. We will shortly see mitigation strategy in section 3.2.6.

3.2.6 Mitigation Strategies

Now that we have identified the threats and vulnerabilities to the application and set the risks and their priorities, the next stage is then to develop a well ordered and cost effective mitigation strategy. As Gary McGraw stated, "*Any suggested mitigation activities must take into account cost, time to implement, likelihood of success, completeness, and impact over the entire corpus of risks*"[12]. This is apparent that a nice and expensive mitigation strategy might not take the attention of the management's eyes. Moreover, implementation time is also another factor that needs to be considered. Most of all the impact of the strategy over the entire parts of the risks should be considered. That is the strategy must be applicable to all the risks of the application not only to some parts. From the point of view of a business context, the limiting factor for a mitigation strategy should be the level the organization can afford, integrate and understand. The strategy must also be able to demonstrate proper mitigation of the risks. Typical measurement system to consider in this strategy is cost [71].

Chapter 4

Case Study

Now that the methods or approaches for testing Open source software security have been defined, they need to be tested on real application. rdpdesk-3.2, Remote Desktop Connection Manager was chosen as a test case application, because it has a version distributed as Open Source software and among all the open source software explored, this application contains compiled files which are appropriate for the experiments. In addition, the application is developed in C/C++ language and tools like Flawfinder and RATS are capable of scanning software developed in C/C++.

4.1 rdpdesk-3.2 (Remote Desktop Connection Manager)

Remote Desktop Connection Manager is intended to coordinate RDP (Remote Desktop Protocol), VNC (Virtual Network Computing) and Citrix ICA (Independent Computing Architecture) connections to Computers which are a part of private or corporate, local or remote, virtual or real networks. It provides secure access over the Internet to remote networks via traffic redirection tools and corresponding client libraries. The application allows easy working with remote desktops and servers. It is distributed both as an OpenSource Edition (OSE) GPL v2 license and commercial proprietary licenses as the Professional Edition (Pro) [75]. The OpenSource Edition version of the application support both Windows 2000/XP/2003/Vista/7 and Linux while the Professional Edition supports Windows 2000/XP/2003/Vista/7 only. As stated earlier the OpenSource Edition is selected as a test case application for this task.

The application provides the following features:

- Manual connection/access configuration
- Simple user authentication
- Certificate-based authentication
- Login-password OS based authentication
- Automatic connection configuration based on downloading configuration files
- RDP-based connection
- ICA-based connection
- VNC-based connection
- Microsoft Internet Security and Acceleration server (MS ISA Server) Support
- Proxy / firewall traversal
- Secure settings storage

In section 4.2, the application will be treated with two static analysis tools namely, Flawfinder and RATS (Look Appendix B for a detailed explanation of the tools). The aim of this analysis is for identifying vulnerabilities in the test case application source code. Then finally in section 4.3, a threat risk model will be developed for the application. The threat risk model is used to further explore the design and architectural flaws in the application.

4.2 Static Source Code Analysis: rdpdesk-3.2

As discussed in section 3.1 static source code analysis is the process of analyzing a program without executing it. Therefore in this section I will take a test case application, rdpdesk-3.2, and then perform static source code analysis using two scanning tools. First we will see how to perform the analysis using Flawfinder and next we will repeat the process using RATS.

4.2.1 Analyzing rdpdesk-3.2 Using Flawfinder

In this analysis the application, rdpdesk-3.2, is scanned using Flawfinder tool. To perform the scanning, Flawfinder must be installed in the system. Note that, Flawfinder is Linux based tool so make sure to install Linux [72] in the system if it is not there. The easiest way to perform the scanning is to save the test case application in the same folder/location with the tool's executable file.
After putting the test case folder in the same location with the tool's executable file, the next step is to run the tool's executable file and pass the test case folder, in our case rdpdesk-3.2, as an argument. Here is how you execute it:

/home/flawfinder/flawfinder-1.27$ ./flawfinder rdpdesk-3.2

If everything went correct the tool starts scanning the test case application and a list of the analysis report is displayed in the screen. Figure 4.1 shows a snapshot of a sample run of the test case application using the tool Flawfinder.

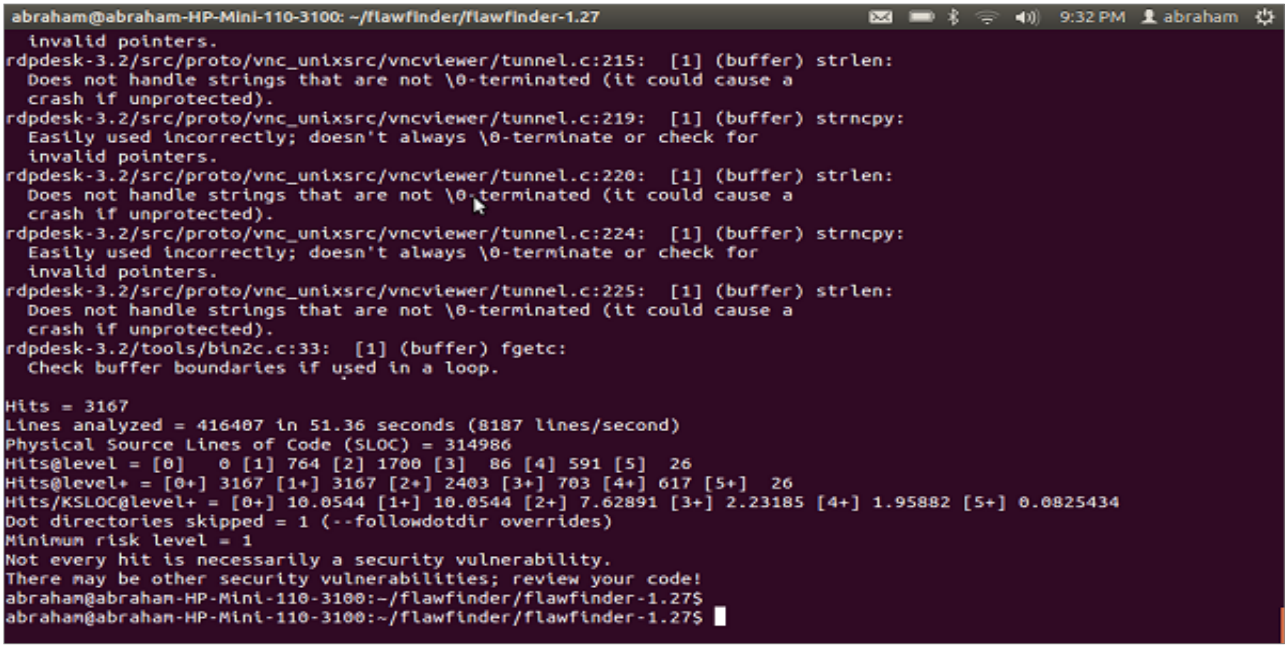


Figure 4.1 Flawfinder snapshot

4.2.2 Analyzing rdpdesk-3.2 Using RATS

Here, the process in section 4.1.1 is repeated but this time with a second tool RATS. To perform the scanning, RATS must be installed in the system. Note again, RATS is Linux based tool so make sure to install Linux in the system if it is not there. In order to install RATS in the system, first Expat XML parser [73] must be installed in the system otherwise RATS installation will not succeed. To analyze the test case application using RATS, I installed Expat-2.0.1. The easiest way to perform the scanning is to save the test case application in the same folder/location with the tool's executable file.

After putting the test case folder in the same location with the tool's executable file, the next step is to run the tool's executable file and pass the test case folder, in our case rdpdesk-3.2, as an argument. Here is how you execute it:

/home/rats/rats-2.3$ ./rats rdpdesk-3.2

If everything went correct the tool starts scanning the test case application and a list of the analysis report is displayed in the screen. Figure 4.2 shows a snapshot of a sample run of the test case application using the tool RATS.

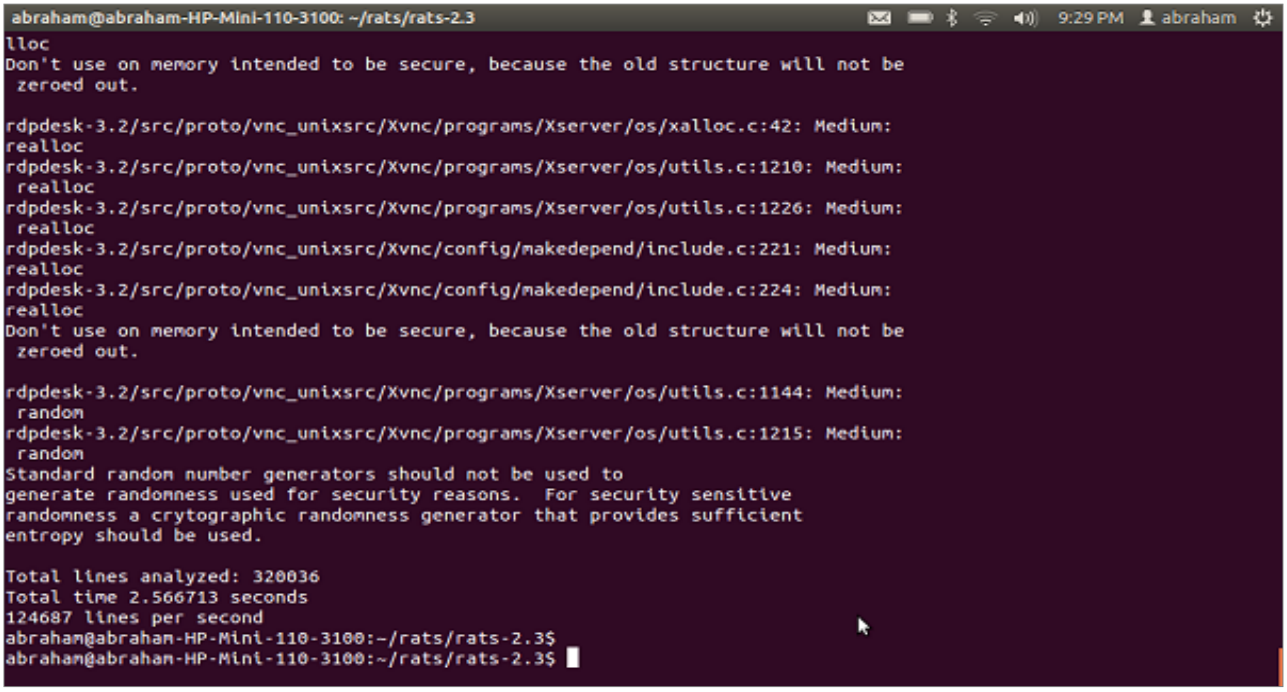


Figure 4.2 RATS snapshot

4.3 Threat Risk Model Development for rdpdesk-3.2

As mentioned in section 3.2, Threat risk modeling is a technique used to develop a model by iteratively assessing the vulnerabilities in an application to find those that are most dangerous and thereby creating a prioritized set of countermeasures to measure and contain the risks. In section 3.2 we have seen the steps required for developing a threat risk model. In this section we will apply these steps to our test case application in order to further secure our software. For this task I selected Microsoft Threat Analysis & Modeling v2.1 [74]. Figure 8 shows a snapshot of Microsoft Threat Analysis & Modeling tool. The main reasons for using this tool are, primarily, it is charge free and secondly, the tool allows anyone to produce a feature-rich threat model [74]. Along with automatically identifying threats, the tool can produce valuable security artifacts such as:

- Data access control matrix
- Component access control matrix
- Subject-object matrix
- Data Flow
- Call Flow
- Trust Flow
- Attack Surface

- Focused reports

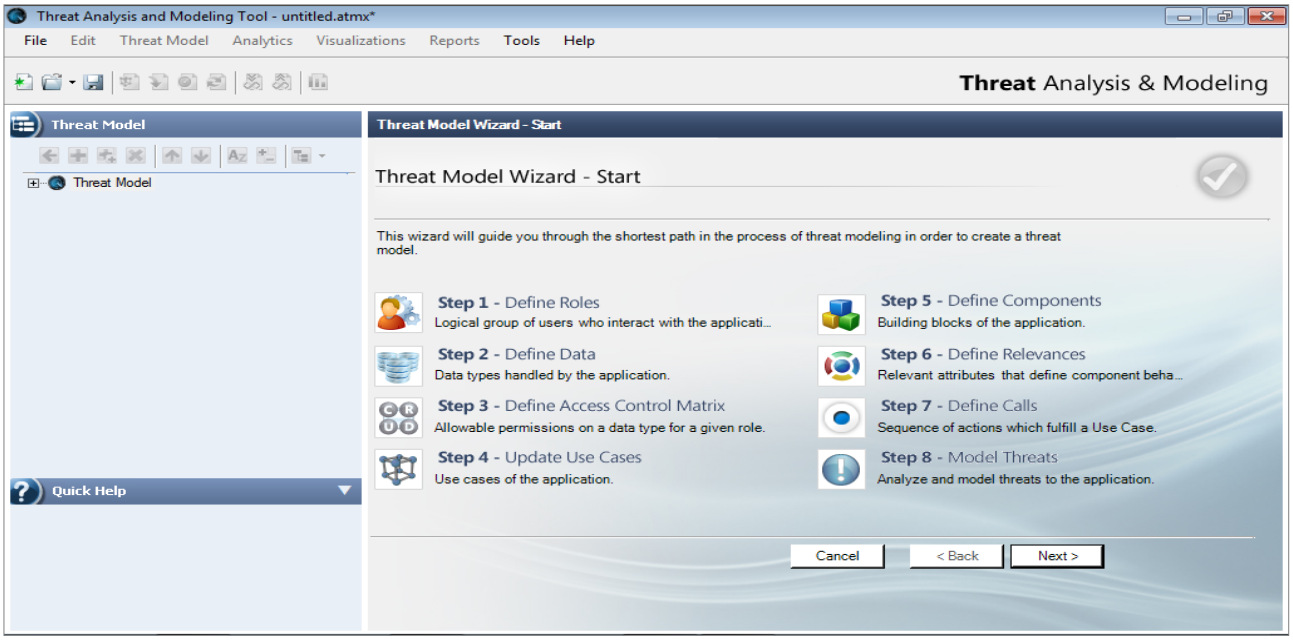


Figure 4.3 Microsoft Threat Analysis & Modeling snapshot

As proposed earlier, the main task of this section is to develop threat risk model for the test case application. The model is developed by iteratively assessing the threats and vulnerabilities in the application and finally finding those that are most dangerous to the application. The developed model helps to create a prioritized set of countermeasures to measure and contain the risks. Developing threat model using Microsoft Threat Analysis & Modeling tool is a three-phase process. First phase is defining application context. Second phase is modeling threats on top of the application context. And the third phase is measuring the risk associated with each threat. We start our task with the initial stage, identifying security objectives.

4.3.1 Identify Security Objectives

The first step in the development of threat risk model is identifying security objectives. Here we create a list of security objectives that provides an understanding about the decisions that leads into selecting and building security controls of our test case application rdpdesk-3.2.

In this task, the following configuration is used. We have a central server where several applications are installed there. rdpdesk-3.2 is installed on clients' machines and it allows them to establish connection with the central server via a network (internet). For example, instead of deploying databases or accounting software or new patches on all clients' machines, the applications can simply be installed on the server and clients can log on and use them via the network. This simplifies upgrading,

troubleshooting, and software management constraints. Figure 8 shows one such usage configuration of rdpdesk-3.2.

The main security objectives of the application include:

- Protection of user accounts and passwords
- Protection of all data transmission between clients and server
- Prevention of unauthorized users from connecting to the system
- Prevention of authorized users from changing privilege levels
- Ensure application availability

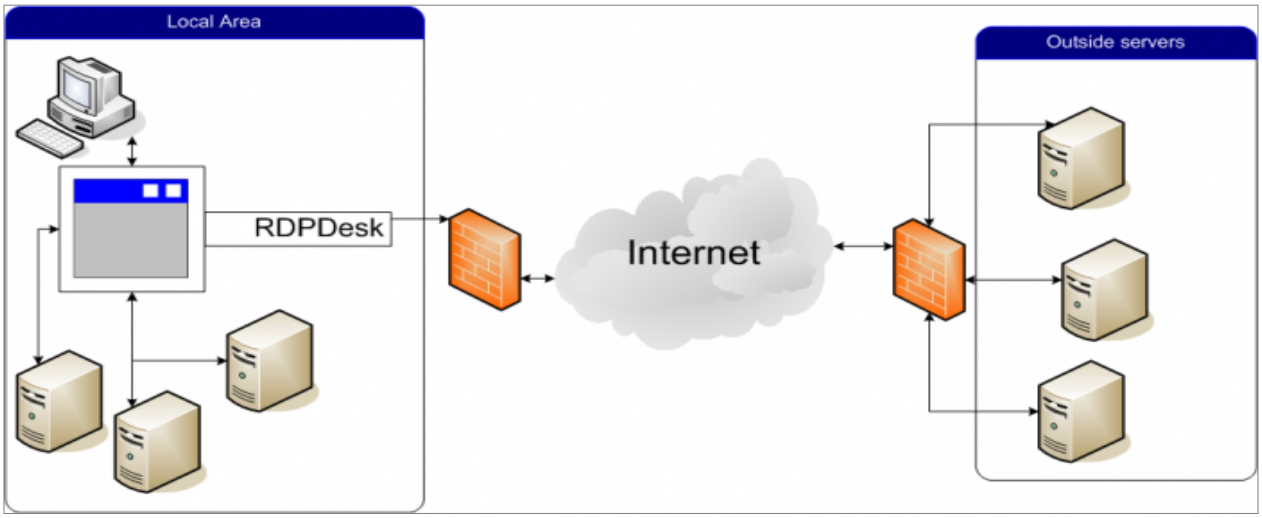


Figure 4.4 rdpdesk-3.2 usage configuration

4.3.2 Survey the Application

The application is a remote desktop connection manager which enables clients to connect to a central server. The server has several applications installed in it. The application enables the clients do everything on the server from their own computer provided that they have logged on and have appropriate privileges.

Application Roles

The test case application has the following user roles.

- Administrator
- Client

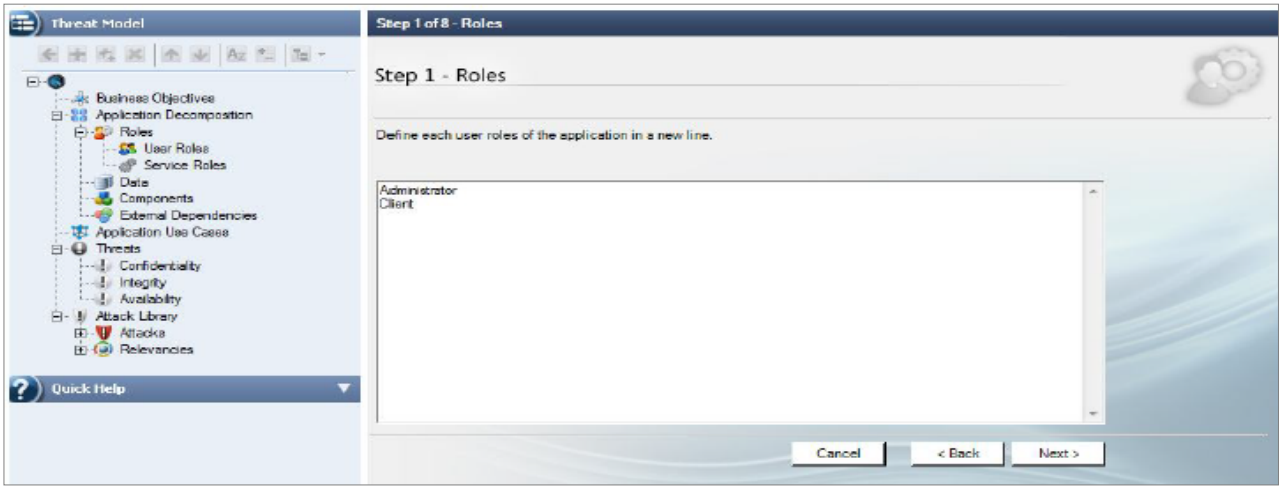


Figure 4.5 user roles

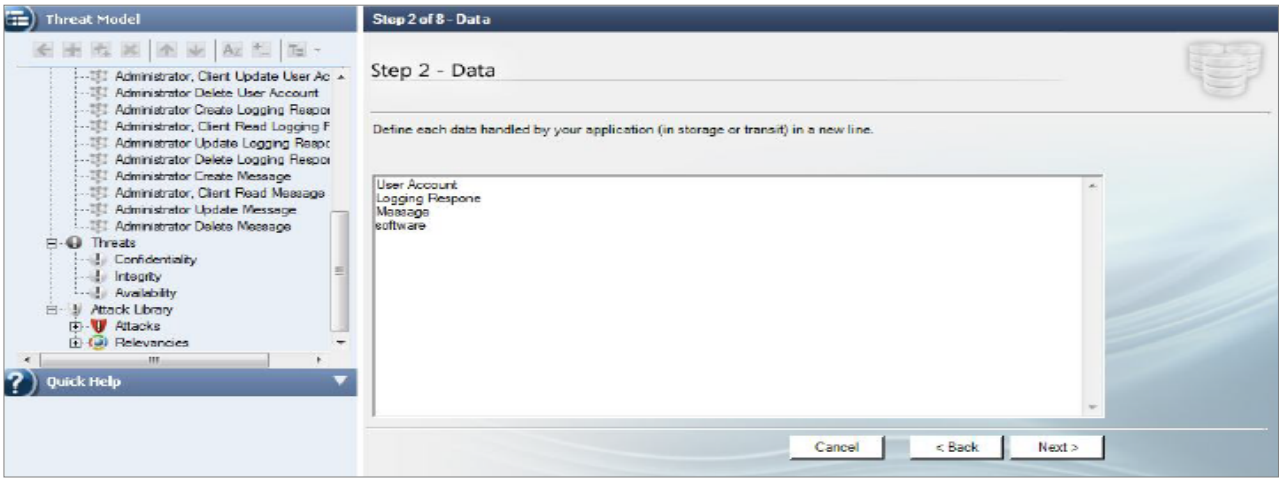


Figure 4.6 Data

Application Key Scenarios

The key scenarios of the test case application are listed below.

- Clients log on to connect to the server
- Authenticated clients browse through the server
- Authenticated clients download file from the server
- Authenticated clients can install software to the server if they have sufficient privilege
- Administrator logs on to the server/clients' machine
- Administrator creates user account
- Administrator deletes user account
- Administrator deletes file from client machine

- Administrator modify file in the clients' machine
- Administrator installs software in the server

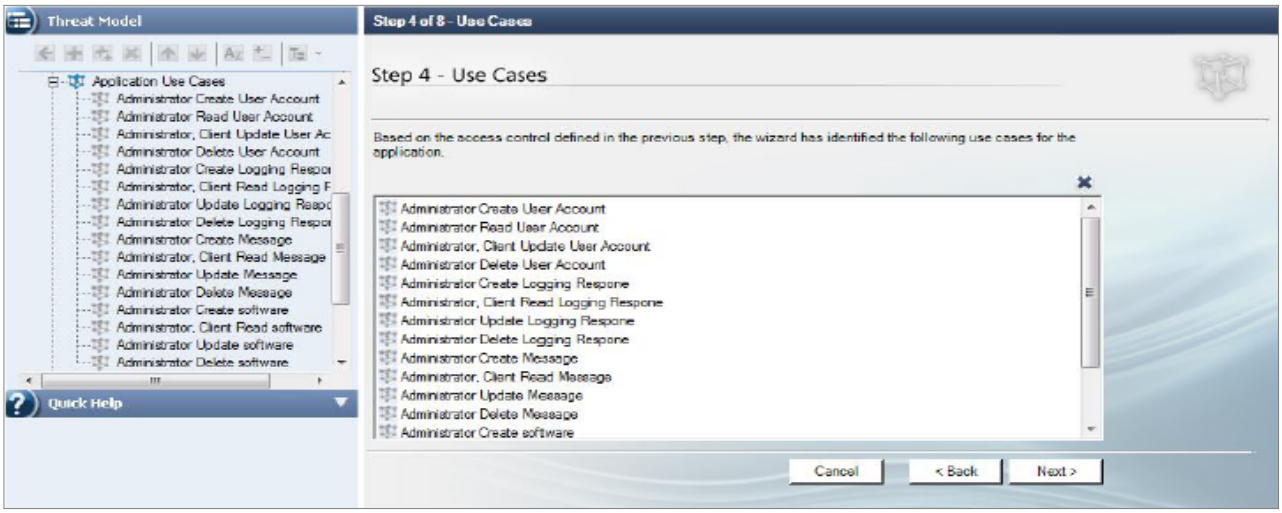


Figure 4.7 Application use cases

Application technologies

The application uses and supports the following technologies and features:

- Programming language C/C++
- Operating system: windows7
- Database server: Microsoft SQL server 2008
- Data encryption: RSA Security RC4 cipher
- Connections: Citrix ICA, Microsoft Windows terminal services (RDP), VNC.

Application security mechanisms

The most important known application security mechanisms of the test case software are:

- Clients can be authenticated using simple user authentication.
- Clients can be authenticated on the basis of certificate-based authentication.
- Administration can be done remotely.
- Clients can be authenticated based on windows based authentication.
- Roles are used to authorize access privileges.
- Application supports Microsoft Internet Security and Acceleration server (MS ISA server).

- Client application supports firewalls and proxies for Internet connection.
- Application uses RSA Security RC4 cipher to encrypt the data.

4.3.3 Application Decomposition

At this stage, the test case application is broken down to identify trust boundaries, data flows, entry points, and exit points with a security impact that need to be evaluated.

Trust boundaries

Listed below are the identified trust boundaries.

- The boundary line firewall
- The database server trusts calls from the application
- The data access part trusts the application to pass valid data

Data flows

The applications' data flows under RDP connection setting are:

1. A client submits group name, server name, user name, password, and domain name through the rdpdesk-3.2 connection setting window. The connection information is handled by the user database module. This module forwards the data to the access component, which verifies the credentials with the database to determine their validity.
2. A client generates an input message (keyboard or mouse), the information is captured by the RDP client and it is encoded as RDP data. The data is then sent to the server. When the Server device driver receives the input data, it decodes it. And finally the actual mouse and keyboard input is sent to the Win32 kernel in the user's session address space, where it is processed as normal input. Figure 4.8 shows RDP graphical data flow between the client and the server.

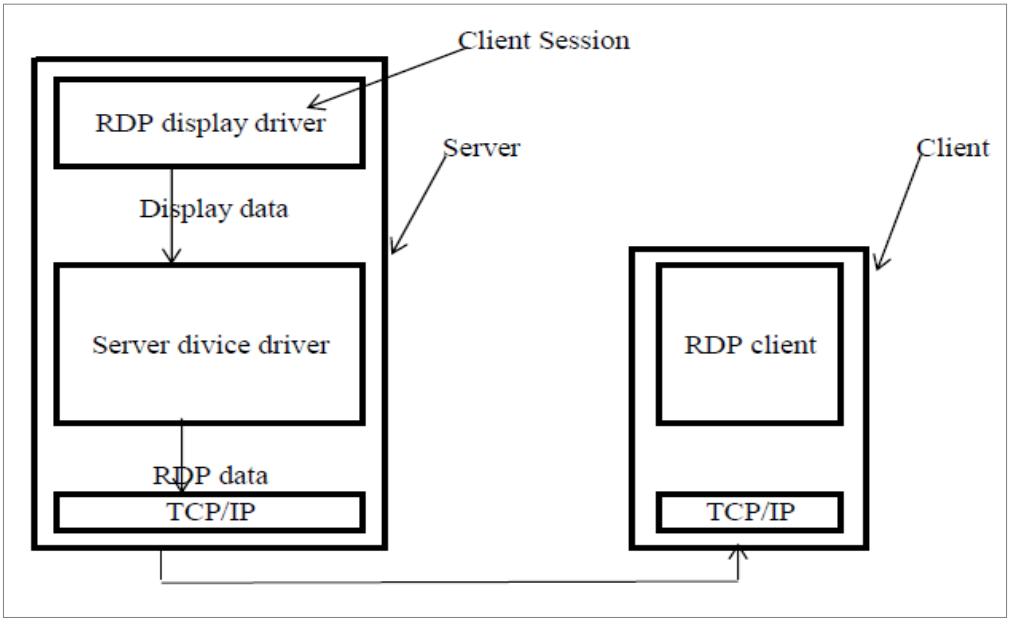


Figure 4.8 RDP graphical data flow between the client and the server

3. A client session uses a special RDP display driver responsible for receiving display commands from the GDI and passing the information to the kernel-mode Terminal Server device driver. This driver encodes the input as RDP data and sends it to the transport layer to be sent to the client. When the client receives the data, the RDP data is decoded and the display updated accordingly. Figure 10 shows RDP mouse or keyboard data flow between the client and the server.

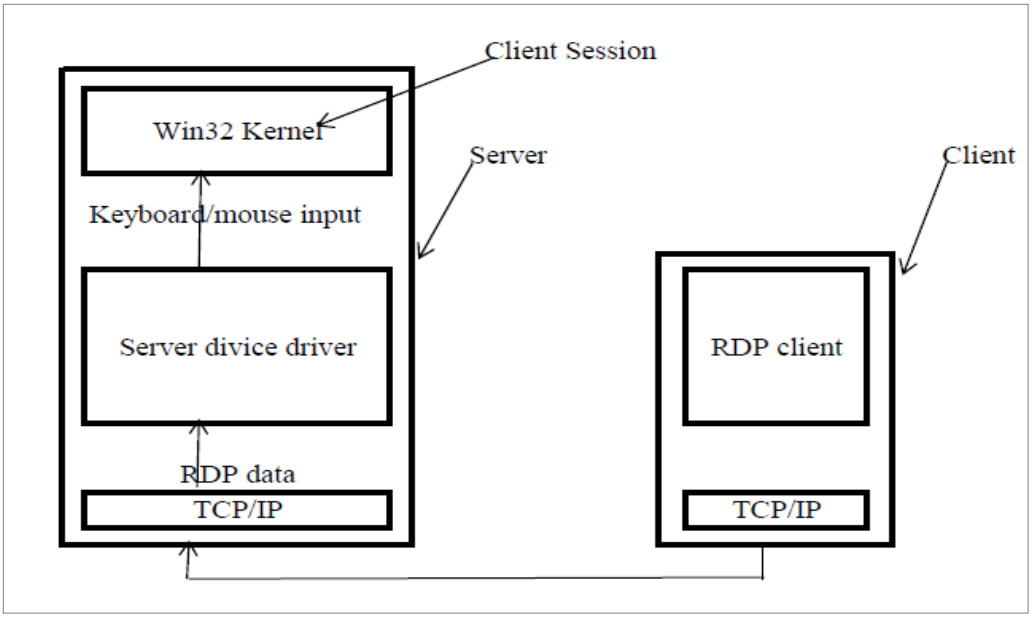


Figure 4.9 RDP mouse or keyboard data flow between the client and the server

4. An administrator logs on and accesses the accounts administration database. The account administration component checks the user role. If the user is authorized, the account administration component interacts with the data access component to view and modify user account.

Entry Points

These points are the main entrance points intended for clients to enter into the application. At this point I identify where these entry points are in the application and the type of data they take.

The following are some of the identified entry points.

- Port to be used for RDP session connection request is port 3389.
- Port to be used for ICA session connection request is port 1494.
- The connection setting windows which is available for the clients to create RDP connection.
- The connection setting windows which is available for the clients to create ICA connection.
- All other ports are restricted by the firewall unless use custom port option is selected by the clients.

Exit points

These points are where the application sends back data to the clients or external systems. For the test case application, I identify where these exit points are.

The following are some of the exit points identified in the application:

- The connection setting page, where the application notifies the client about the connection status (connected or not connected).
- The remote system panel that shows the remote connected server.

4.3.4 Identify Threats

Now that the application is decomposed, the next process in the threat risk modeling development is to identify the threats and attacks that might compromise the security objectives of the application. Since it is difficult to identify unknown threats, I concentrate on known threats. Listed below are some of the common know threats to the application.

- Information that is not intended to be public is disclosed
- A dictionary attack is used to guess a password by trying all of the password in a list or a dictionary
- A Brute force used to guess a password
- A Denial of Service is used to interrupt normal operation

- Man in the Middle (MiM) attack is used. That is an attacker impersonate a server and the clients try to create connection with the attacker unknowingly. As a result the attacker gets the client credentials and then he is able to use these credentials to connect to the real server.
- An attacker obtains the encryption keys used to encrypt sensitive data by acting as MiM.
- Cross-site scripting XSS occurs by injecting script codes.
- SQL injection occurs, an attacker exploits an input validation vulnerability to execute commands in the database and thereby access or modify data.
- An attacker or a client gets unauthorized access to server resources.

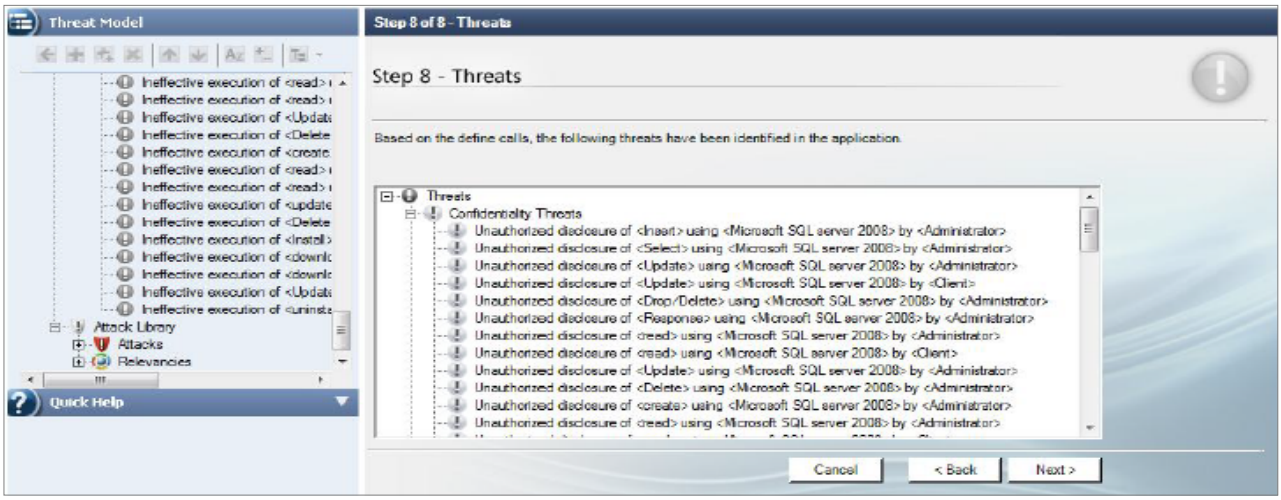


Figure 4.10 Application Threats

4.3.5 Identify Vulnerabilities

This step is the final step in the development of threat risk modeling. It is the stage where the application's vulnerabilities are identified. A good approach in identifying vulnerabilities is to examine the application looking at every vulnerability categories at each layer. The same procedure as the threat identification is used here except that now the target is identifying vulnerabilities rather than threats. Therefore the following vulnerabilities are some of the identified vulnerabilities.

- Users' account password storage.
- Deficiency of password complexity enforcement.
- Missing or weak input validation at the server.
- Lack of password retries.
- Improper encoding of output leading to potential cross-site scripting issues.
- Disclosing exception details to clients.
- Weak encryption keys used to encrypt sensitive data.

Note that a comprehensive threat risk model report for the application (generated using the tool Microsoft Threat Analysis & Modeling v2.1) is found in Appendix C.

Chapter 5

Research Result

Generally speaking, detecting vulnerabilities in an application is a vital process, but just as vital is being able to estimate the associated risk of the detected vulnerabilities to the organization. Estimating the impact of the risk is not an easy process unless the threats are ranked according to their risk ratings. Therefore once the vulnerabilities in the application are identified using any tools available, the next task is to rank them based on the value of their risk rating. With the use of the chosen static analysis tools, this is not a problem as the tools are capable of ranking the identified vulnerabilities in the application. However, the vulnerabilities identified using the threat risk model should be ranked by the analyst in order to help in the decisions of prioritizing risk mitigation.

This chapter starts with an explanation of the vulnerabilities generated using the static analysis tools. Finally, the vulnerabilities obtained from the threat risk modeling technique are further classified into several ranks (high, medium and the likes) based upon their risk ratings. As mentioned earlier, these ranks will be useful in identifying the risks that need to be prioritized during mitigation time. Remember that not all vulnerabilities are worth of mitigating. That is some vulnerabilities are difficult to be exploited by attackers. Therefore investing extra money to mitigate these flaws is of nothing importance.

In this task, for estimating the severity (risk rank) of the risks on the organization, the OWASP risk rating methodology is used. The ease of customization to the needs of the organizations is the primary reason for the choice of the aforementioned ranking method. A brief description of OWASP Risk Rating Methodology will be given later on this same chapter.

5.1 Flawfinder Report Analysis

The first static analysis experiment was carried out using flawfinder. The tool is designed for Unix and Linux platform However, with the proper installation of Cygwin it can be used in windows platform too. In my experiment I used Ubuntu 11.10. The experiment was done on test case application rdpdesk-3.2. The application contains 933 files and 314986 Source Lines Of Codes (SLOC).

The experiment generated 12,003 lines of report. By analyzing the report, it was found that the tool detected 3167 vulnerabilities on the application. All these vulnerabilities fall into five levels, 5 being the most dangerous and 1 the least dangerous. After carefully studying the report, 26 out of the 3167 vulnerabilities were found to be of level 5, 591 vulnerabilities of level 4, 86 vulnerabilities of level 3,

1700 vulnerabilities of level 2 and 764 vulnerabilities of level 1. As far as the type of vulnerabilities concerned, the tool detected 10 types of vulnerabilities.

As shown in Table 5.1, 2781(87.81%) of the detected vulnerabilities are of the type buffer overflow. These flaws happen when functions that do not perform bound checking are used. Therefore, it is advisable to use functions that limit length, or ensure that the size of array is larger than the maximum possible length.

From the table it is clear that race condition [78] should be giving high focus as it contains 22 (84.6%) out of the 26 critical vulnerabilities. And vulnerabilities of this type can occur when read and write commands of a large amount of data are received relatively at the same time. As a result, the machine tries to overwrite some or all of the old data while that old data is still being read. The main consequences of this are system crash, illegal operation, reading and writing errors, and so forth. Serialization of memory or storage access can prevent such problem. That is if read and write are received at the same time, the read command is by default executed and completed first and then execution continues on the write command.

In the application 1 port vulnerability is detected. It is of level 1 severity. Vulnerabilities of such types are not problem by themselves, however they open door to buffer overflow. This vulnerability happened due to incorrect implementation of the snprintf function. Proper checking should be done during installation to avoid the problem.

SN	Vulnerability type	Number of vulnerabilities detected	Severity of detected vulnerabilities				
			5	4	3	2	1
1	Buffer overflow	2781	4	461	29	1534	753
2	Format String overflow	95	0	95	0	0	0
3	Integer overflow	64	0	0	0	64	0
4	Race condition	38	22	14	1	1	0
5	Random number	25	0	0	25	0	0
6	Access	7	0	0	0	0	7
7	Port	1	0	0	0	0	1
8	Obsolete	3	0	0	0	0	3
9	Shell	18	0	16	2	0	0
10	Misc	135	0	5	29	101	0

Table 5.1 Flawfinder report analysis

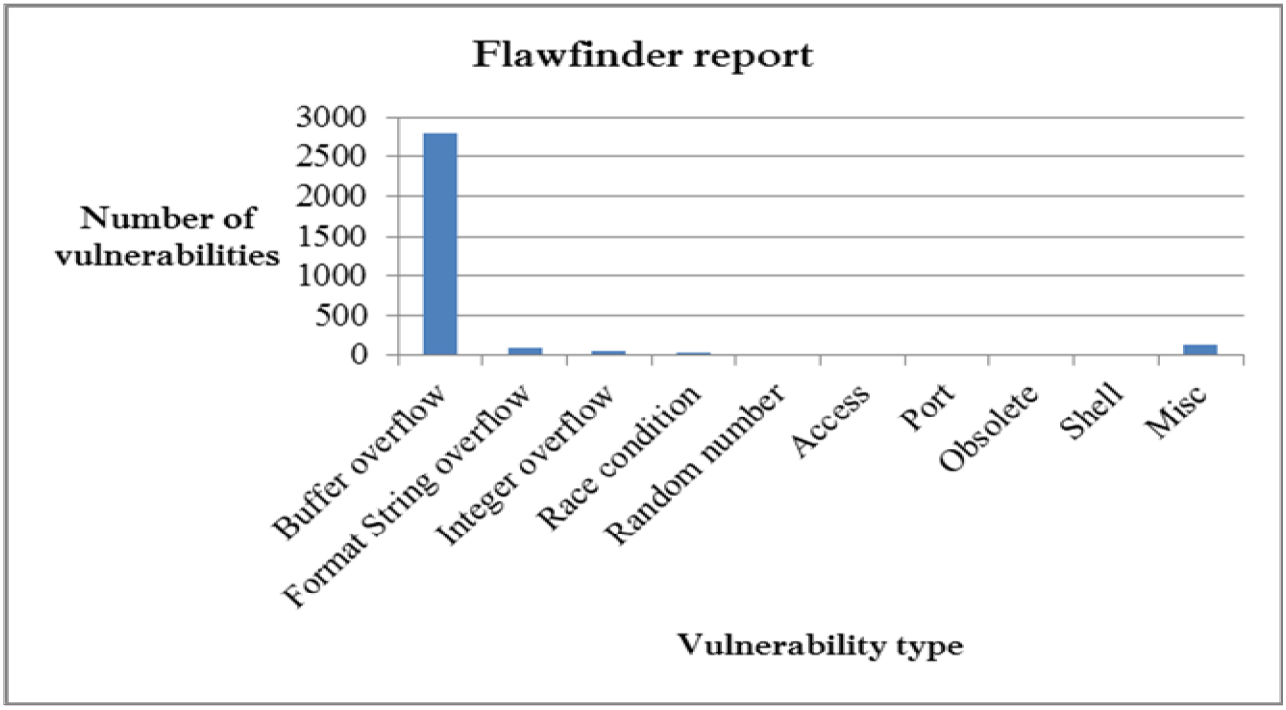


Figure 5.1 Flawfinder report

In the following paragraphs we will take a look on a snippet of the error causing codes for each of the vulnerability types identified in the test case application. And the proposed solution to these errors follows it. Table 5.2 shows description of the impact of the vulnerabilities detected using Flawfinder and their solution.

Buffer overflow: 87.8% of the total vulnerabilities of the application are of the type buffer overflow. This error is initiated when a function attempts to copy into the memory without performing bounds check. Vulnerabilities of such type can overwrite the memory or read sensitive data from the memory if exploited by an attacker. As an example we will see a snippet code of such vulnerability that was found in the test case application.

```
1  hostname = (char *) xalloc (
2  strlen (ciptr->transptr->TransName) + strlen (addr) + 2);
3  strcpy (hostname, ciptr->transptr->TransName);
4  strcat (hostname, "/");
5  if (addr)
6  strcat (hostname, addr);
7  return (hostname);
```

In line 2 the function call strlen is banned string length function. This function should be replaced with either String*Length or strnlen_s. this error is most problematic for critical applications, such as those accepting anonymous Internet connections. These problems can result to system crash.

In line 3 the function strcpy results in buffer overflow vulnerability this is because the functions copies the content of ciptr->transptr->TransName to hostname without performing bound check. This vulnerability can overwrite the memory. And if it is exploited by an attacker it gives the attacker a chance to run malicious code. Possible solution is to avoid using the function strcpy instead replace it with String*Copy or String*CopyEx or strcpy_s. These are the safe versions of strcpy.

In lines 4 and 6 we have a function call strcat. This is a string concatenating function and it can cause buffer overflow if it is misused. Therefore, using String*Cat or String*CatEx or strcat_s is the safe way.

Format string overflow: this type of vulnerability comprises ~3% of the total vulnerabilities on the application. These vulnerabilities are not dangerous by themselves however they can be the reason for buffer overflow. Format string overflow is caused from use of unchecked user input as a format string parameter. Problem of this type can cause crashing of a program or execution of harmful code. Let's see at a vulnerable code snippet from our application.

```
# else
   len = sprintf(buf, format, a1, a2, a3, a4, a5, a6, a7, a8,
            a9, a10, a11, a12, a13, a14, a15, a16, a17, a18, a19, a20);
# endif
```

The function call sprintf is a banned version. This function accepts formatting string parameters from a user without check it. At the worst case, an attacker may use the %s and %x format tokens to read data from the stack or possibly other locations in memory. One way to solve this problem is to use the safest versions of this function call, String*Printf or String*PrintfEx sprintf_s.

Integer overflow: these vulnerabilities accounts for ~2% of the total vulnerabilities in the application. The main problem with this type of vulnerability is that it is difficult to detect. These vulnerabilities are not exploited directly, however any result obtained from them which has something to do with the memory can cause buffer overflow. Here follows a snippet of the vulnerable code from the test case application.

```
1  if(++i < argc)
2  defaultPointerControl.num = atoi(argv[i]);
3  else
4  UseMsg();
```

In line 2, atoi is a function that converts string to integer, unless checked, the resulting number can exceed the expected range. If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).

Race condition: this accounts for ~1.2% of the total vulnerabilities in the application. Details are provided earlier some paragraphs above. A snippet code of this type looks like this;

```
if ((fp = fopen(fname,"w")) == NULL) return 1;
chmod(fname, S_IRUSR|S_IWUSR);
```

S_IRUSR read permission bit for the owner of the file. On many systems this bit is 0400 and S_IWUSR writes permission bit for the owner of the file usually this bit is 0200. [79]. if an attacker can move those files, a race condition results. Therefore, it is preferable to use fchmod() instead.

Random number: these vulnerabilities account for 0.7%. These vulnerabilities arise when a less secure random number is generated using a random value generating function. This code is taken from the test case application.

```
char random;
for (int i = 0; i < 256; i ++)
  {
    random = (char)(rand() % 256);
    random ^= (lt*(i+1) % 128);
    base_key.Append(random);
  }
```

The function rand() is not sufficiently random for security-related functions such as key and nonce creation. The proposed solution is to use a more secure technique for acquiring random values.

Access, Port, Obsolete: these types of vulnerabilities were identified to be of low numbers and low level of severity. Combined together they accounted for 0.34% of the total vulnerabilities in the application.

Shell: 0.56% of the total vulnerabilities on the application are of the type shell vulnerably. These vulnerabilities could allow execution of remote codes. If an attacker successfully exploited this vulnerability, he could gain the same user rights as the local user. The possible solution of this is to try using a library call that implements the same functionality if available. Here is a snippet of the vulnerable code from the tested application.

```
fp = popen (aout, "r");
if (fp == NULL)
abort ();
```

Vulnerability Type	Impact on Organization	Possible solutions
Buffer overflow	- overwrite memory - read sensitive data from the memory - system crash	Use of safe functions. That is avoid the use of banned functions.
Format string overflow	- overwrite memory - read sensitive data from the memory - system crash	Use of safe functions. That is avoid the use of banned functions.
Integer overflow	Same as buffer overflow	- Use unsigned integer instead of signed. - Use SafeInt technique
Race condition	- system crash - illegal operation - reading and writing errors	Serialization of memory or storage access.
Random number	Cryptographic key break	Use a more secure random number.
Shell	Gain access rights	Use of library call which implements same functionality.

Table 5.2 Impacts and solutions of the vulnerabilities detected using Flawfinder

5.2 RATS Report Analysis

The next experiment was carried out using another static analysis tool called RATS. This is because it is always better to use several tools in order to detect as much vulnerability as possible. RATS is designed for Unix, Linux and windows platforms. The experiment was done on test case application rdpdesk-3.2. The application contains 933 files and the total lines analyzed are 320036 lines.
The tool detected 1282 vulnerabilities on the application. 782 that is ~61% of the total vulnerabilities are of the type buffer overflow and 336 (26.21%) are of the type format string overflow. The rest of the vulnerabilities added together account for ~12%. 87.13% of the vulnerabilities are of high severity while 12.87% are of medium severity. The vulnerabilities detected fall into 10 vulnerability types.

As shown in table 5.3 RATS detected some vulnerabilities such as Exception throw, DNS-forgery, Time Of Check Time Of Use (TOCTOU) and Trojan horse insertion that were not detected using Flawfinder. However, the tool didn't detect any integer overflow vulnerabilities on the test case application.

S/N	Vulnerability type	Number of vulnerabilities detected	Severity of detected vulnerabilities		
			High	Medium	Low
1	Buffer overflow	782	709	73	0
2	Format String overflow	336	336	0	0
3	Integer overflow	0	0	0	0
4	Input validation	34	34	0	0
5	Exception throw	6	6	0	0
6	Race condition	40	0	40	0
7	Random Number	35	0	35	0
8	DNS-forgery	15	15	0	0
9	Trojan horse insertion	11	11	0	0
10	Time Of Check Time Of Use TOCTOU	23	0	23	0

Table 5.3 RATS report analysis

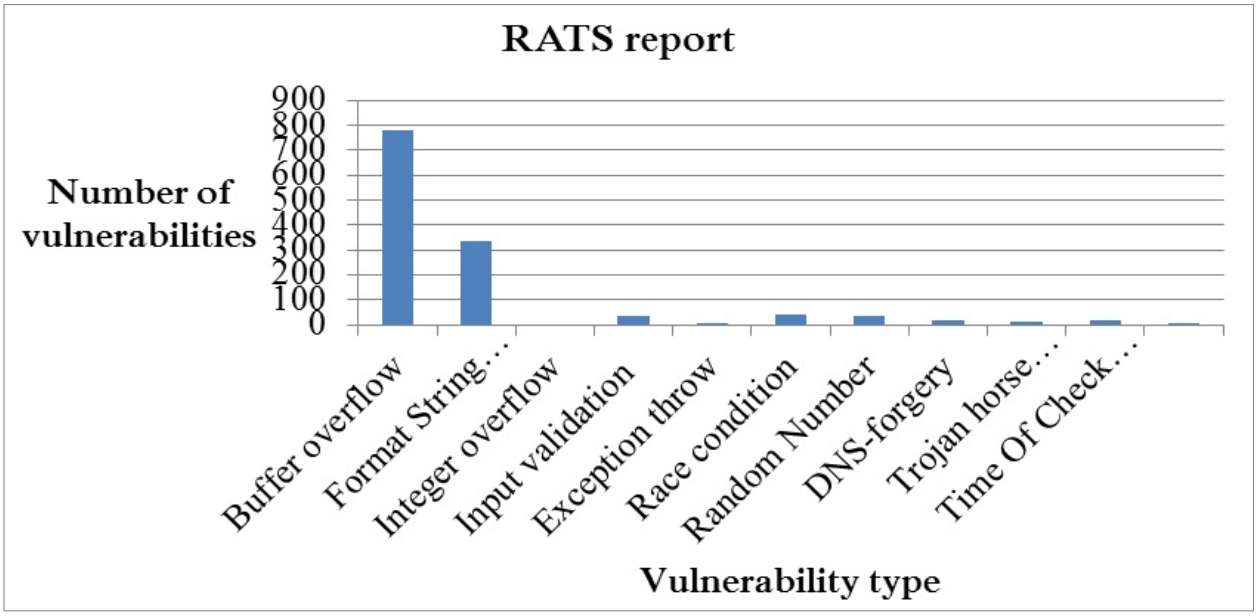


Figure 5.2 RATS report

Following are some explanation of the detected vulnerabilities on the application with a snippet of the vulnerable codes. Table 5.4 shows description of the impact of the vulnerabilities detected using RATS and their solution.

Buffer overflow:

```
FILE *fp;
   int i, ch;
char passwd[16];
```

Code snippet A,

```
}
    strcpy(passwdFile, argv[1]);
    read_from_stdin = 0;
    make_directory = 0;
    check_strictly = 0;
  }
```

Code snippet B,

In A the variable passwd is initialized to an array of 16 characters. This is vulnerable to buffer overflow. If unsafe functions are used to copy data to this variable, then there is high probability of buffer overflow. For example an attacker might send more than 16 characters to the passwd variable and overwrite the memory. In B the idea is to copy the second argument that passed from the command line to the variable passwdFile. However, due to the use of the unsafe function strcpy, buffer overflow can result if the value passed from the command line exceeded the length of the variable passwdFile. Successfully exploitation of such vulnerabilities might result in system crash, sensitive data reading from memory, writing to memory, and execution of malicious code. The main solutions to this type of vulnerability are use of safe functions as well as extra care should be taken to ensure that character arrays that are allocated on the stack are used safely.

Format String overflow:

```
if (m_pDSMPlugin->IsEnabled())
        {
                char szMess[255];
                memset(szMess, 0, 255);
                sprintf(szMess, "--- Ultr@VNC Viewer + %s-v%s",
                        m_pDSMPlugin->GetPluginName(),
                        m_pDSMPlugin->GetPluginVersion()
                        );
```

```
strcat(m_desktopName, szMess);
```

In this piece of code we see the unsafe function call strcat in the last line. The idea here is to concatenate two values m_desktopName and szMess. However, if argument2 copy more data than can be handled, it might open door for buffer overflow. Therefore the solution is to use a safer version strcat_s.

Input validation:

```
fp = popen (aout, "r");
 if (fp == NULL)
   abort ();
```

Solution Argument 1 to this function call should be checked to ensure that it does not come from an untrusted source without first verifying that it contains nothing dangerous.

DNS forgery:

```
gethostname(hname, 1024);
        host = gethostbyname(hname);
        if (host == NULL)
           hnameptr = hname;
        else
           hnameptr = host->h_name;
```

The gethostbyname function in the above piece of code returns information (ip) about the host named *hname*. If the lookup fails, it returns a null pointer. This function call is vulnerable because the DNS result obtained can't be trusted. This is because DNS results can easily be forged by an attacker (or arbitrarily set to large values, etc), and should not be trusted. If an attacker can send DNS responses to a vulnerable system, he can easily cause a denial of service, crashing the application that made calls to a vulnerable resolver library. It is hard to execute arbitrary code using this vulnerability. However, information disclosure is possible if a vulnerable system returns the contents of memory adjacent to a DNS response. In another way if exploited, this vulnerability could also be used to gain root access to the system. One solution is to use maximum buffer size.

Trojan horse insertion:

```
m_hZipDll = LoadLibrary(szDllFn);
                if ( m_hZipDll == NULL )
                {
```

```
        CHAR            szWinDir[ MAX_PATH ] = "";
        ::GetWindowsDirectory( szWinDir, sizeof( szWinDir ) );
        if ( ::strlen( szWinDir ) )
        {
            char        cLastChr = szWinDir[ ::strlen( szWinDir ) - 1 ];
            if ( cLastChr != '\\' && cLastChr != '/' )
                ::strcat( szWinDir, "\\" );
            ::strcat( szWinDir, ZIP_DLL_NAME );
            m_hZipDll = LoadLibrary( szWinDir );
        }
    }
```

The function LoadLibrary in the above piece of code loads the module named szDllFn into the address space of the calling process. However, if m_hZipDll is null that is szDllFn does not exist, the function loads szWinDir. The function LoadLibray() searches all places for a library, if no path is specified this might allow a Trojan to be inserted somewhere. The solution to this problem is to specify the full path of the library.

Time Of Check Time Of Use TOCTOU:

TOCTOU is another type of vulnerability that is caused by changes in a system between the *checking* of a condition and the *use* of the results of that check. As seen from the race condition vulnerability earlier, TOCTOU is a kind of race condition. The following code is a snippet of a vulnerable code that was taken from the test case application.

```
struct stat buf;
static time_t lastmod = 0;
int ret = stat(SecurityPolicyFile , &buf);
if ( (ret == 0) && (buf.st_mtime > lastmod) )
{
   ErrorF("reloading property rules\n");
   SecurityFreePropertyAccessList();
   SecurityLoadPropertyAccessList();
   lastmod = buf.st_mtime;
}
```

The *stat*() function obtain information about the file SecurityPolicyFile and write it to the area pointed to by the *buf* argument. If successfully completed, 0 is returned. Otherwise, -1 is returned.

The stat() call is a check call. If it is followed by a use call, it can be lead to TOCTOU vulnerability. One solution to this problem is to flag the call to the *stat()* if the first argument (dir name) is used later in use-category [80]. Another solution is to use safer functions such as, *lstat()*, *fstat()* and *open().*

Vulnerability Type	Impact on Organization	Possible solutions
Buffer overflow	- overwrite memory - read sensitive data from the memory - system crash	Use of safe functions. That is avoid use of banned functions.
Format string overflow	- overwrite memory - read sensitive data from the memory - system crash	Use of safe functions. That is avoid use of banned functions.
Input validation	- Malicious code execution - System crash	Check function call properly to insure that it does not come from an untrusted source.
DNS-forgery	- Denial-of-Service - Information disclosure - Gain root access	Use of maximum buffer size
Trojan horse insertion	- Steal information - System crash - Denial-of-Service - File modification - Keystroke logging and etc.	specify the full path of the library
Time Of Check Time Of Use TOCTOU	Same as race condition	- flag the call to the *stat()* if the first argument (dir name) is used later in use-category - Use safer functions such as, *lstat()*, *fstat()* and *open().*

Table 5.4 Impacts and solutions of the vulnerabilities detected using RATS

5.3 Risk ranking

In sections 5.1 and 5.2 we have seen the different types of vulnerabilities and their severity level which were detected on the test case application using the static analysis tools. The primary advantage of organizing vulnerabilities based on their severity level is that it helps us in making decisions on which class of vulnerabilities to focus during the mitigation process.

Up to this point, we have uncovered the vulnerabilities and their severities on the application's source code using the static analysis tools. Based on the analysis report from the tools, the impact of the detected vulnerabilities and their possible solutions were described.

In section 4.2, we have identified the Application design and architectural threats and vulnerabilities using the threat risk modeling approach. Therefore it is important to estimate the associated risk of the detected vulnerabilities to the organization in order to be able to differentiate those of high risk and those that have no impact or of little impact to the organization. As mentioned earlier, for estimating the severity (risk rank) of the risks on the organization I choose the OWASP Risk Rating Methodology. But first, as I promised earlier in the beginning of this chapter, I am going to provide a brief description of the OWASP Risk Rating Methodology.

5.3.1 OWASP Risk Ranking Methodology

The OWASP risk rating methodology is based on the standard risk model formula given by:

Risk = likelihood × impact

To determine the risk ranking, the OWASP risk rating approach undergoes a six step process. These steps are as follows:

Step 1: Identifying a Risk
Step 2: Factors for Estimating Likelihood
Step 3: Factors for Estimating Impact
Step 4: Determining Severity of the Risk
Step 5: Deciding What to Fix
Step 6: Customizing Your Risk Rating Model

However, in this task we will use only the first five steps.

Step 1: Identifying a Risk

This step involves collection of information about the threat agents involved, the attacks used by the threat agents, the vulnerabilities involved, and the impact of a successful exploit.

Step 2: Factors for Estimating Likelihood

This is an approximate measure of how likely a particular vulnerability can be discovered and exploited by an attacker. Threat agent and vulnerability are some of the factors that can help in determining the estimation of likelihood.

Note that each factor has a set of options which have a likelihood rating from 0 to 9 associated with them. [81]

Step 3: Factors for Estimating Impact

Basically, there exist two kinds of impacts. One is "technical impact" on the application, the data used, and its functionality. The second is the "business impact".

Note again each factor has a set of options which have an impact rating from 0 to 9 associated with them.

Step 4: Determining Severity of the Risk

This step deals with calculating the severity of a risk based on the likelihood estimate and the impact estimate from step 2 and 3. The major thing to do here is to figure out whether the likelihood and impact are LOW, MEDIUM, or HIGH based on table 5.5.

Likelihood and Impact Levels	
0 to <3	LOW
3 to <6	MEDIUM
6 to 9	HIGH

Table 5.5 Likelihood and Impact severities

Overall risk severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Table 5.6 Overall risk severity

Step 5: Deciding What to Fix

As now the risks to the application are classified, a prioritized list of what to fix is at hand. The most severe risks are fixed first. When performing the fixes, take into consideration cost as well as reputation damage.

5.3.2 Risk ranking of the Vulnerabilities Generated Using Threat Risk Modeling Technique

In section 4.2, several application design and architectural vulnerabilities were detected using threat risk modeling technique. Now the task is to determine the ranks of these vulnerabilities. The detected vulnerabilities are listed below:

- Users' account password storage.
- Deficiency of password complexity enforcement.
- Missing or weak input validation at the server.
- Lack of password retries limit.
- Improper encoding of output leading to potential cross-site scripting issues.
- Disclosing exception details to clients.
- Weak encryption keys used to encrypt sensitive data.

Let's start determining the ranks of the above vulnerabilities. We start with users' account password storage.

Users' account password storage

Threat agent factors					**Vulnerability factors**			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
9	4	4	2		9	5	6	3
4.75					5.75			
Overall likelihood = 5.25 (MEDIUM)								

Technical impact					Business impact			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability		Financial damage	Reputation damage	Non-compliance	Privacy violation
9	9	1	7		3	1	5	5
Overall technical impact = 6.5 (HIGH)					**Overall business impact = 3.5 (MEDIUM)**			

From the above calculation the overall likelihood is (MEDIUM), the overall technical impact is (HIGH) and the overall business impact is (MEDIUM). Therefore based on table 6, from a technical prospective, it appears that the overall severity is HIGH and from a business prospective, the overall severity is MEDIUM.

Deficiency of password complexity enforcement

Threat agent factors					Vulnerability factors			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
9	4	7	9		9	5	6	3
7.25					5.75			
Overall likelihood = 6.5 HIGH)								

Technical impact					Business impact			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability		Financial damage	Reputation damage	Non-compliance	Privacy violation
9	9	9	7		3	1	5	5
Overall technical impact = 8.5 (HIGH)					**Overall business impact = 3.5 (MEDIUM)**			

The overall likelihood is (HIGH), the overall technical impact is (HIGH) and the overall business impact is (MEDIUM). Therefore based on table 6, from a technical prospective, it appears that the overall severity is CRITICAL and from a business prospective, the overall severity is MEDIUM.

Missing or weak input validation at the server

Threat agent factors					Vulnerability factors			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
6	4	7	2		9	3	6	9
4.75					6.75			
Overall likelihood = 5.75 (MEDIUM)								

Technical impact					Business impact			
11	Loss of integrity	Loss of availability	Loss of accountability		Financial damage	Reputation damage	Non-compliance	Privacy violation
7	7	9	7		3	1	5	5
Overall technical impact = 7.5 (HIGH)					**Overall business impact = 3.5 (MEDIUM)**			

The overall likelihood is (MEDIUM), the overall technical impact is (HIGH) and the overall business impact is (MEDIUM). Therefore using table 6, from a technical prospective, it appears that the overall severity is HIGH and from a business prospective, the overall severity is MEDIUM.

Lack of password retries limit

Threat agent factors					Vulnerability factors			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
1	1	7	9		3	5	6	3
4.5					4.25			
Overall likelihood = 4.375 (MEDIUM)								

Technical impact					Business impact			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability		Financial damage	Reputation damage	Non-compliance	Privacy violation
9	9	9	7		3	1	5	5
Overall technical impact = 8.5 (HIGH)					**Overall business impact = 3.5 (MEDIUM)**			

The overall likelihood is (MEDIUM), the overall technical impact is (HIGH) and the overall business impact is (MEDIUM). Therefore using table 6, from a technical prospective, it appears that the overall severity is HIGH and from a business prospective, the overall severity is MEDIUM.

Improper encoding of output

Threat agent factors					**Vulnerability factors**			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
9	4	7	2		9	3	6	3
5.5					5.25			
Overall likelihood = 5.375 (MEDIUM)								

Technical impact					**Business impact**			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability		Financial damage	Reputation damage	Non-compliance	Privacy violation
7	7	1	7		3	1	5	5
Overall technical impact = 5.5 (MEDIUM)					**Overall business impact = 3.5 (MEDIUM)**			

The overall likelihood is (MEDIUM), the overall technical impact is (MEDIUM) and the overall business impact is (MEDIUM). Therefore using table 6, from a technical prospective, it appears that the overall severity is MEDIUM and from a business prospective, the overall severity is MEDIUM.

Disclosing exception details to clients

Threat agent factors					**Vulnerability factors**			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
9	4	7	2		7	3	6	3
5.5					4.75			
Overall likelihood = 5.125 (MEDIUM)								

Technical impact					Business impact			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability		Financial damage	Reputation damage	Non-compliance	Privacy violation
9	9	9	7		3	1	5	5
Overall technical impact = 8.5 (HIGH)					**Overall business impact = 3.5 (MEDIUM)**			

The overall likelihood is (MEDIUM), the overall technical impact is (HIGH) and the overall business impact is (MEDIUM). Therefore using table 6, from a technical prospective, it appears that the overall severity is HIGH and from a business prospective, the overall severity is MEDIUM.

Weak encryption keys

Threat agent factors					Vulnerability factors			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
9	4	7	2		9	3	6	3
5.5					5.25			
Overall likelihood = 5.375 (MEDIUM)								

Technical impact					Business impact			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability		Financial damage	Reputation damage	Non-compliance	Privacy violation
7	7	1	7		3	1	5	5
Overall technical impact = 5.5 (MEDIUM)					**Overall business impact = 3.5 (MEDIUM)**			

The overall likelihood is (MEDIUM), the overall technical impact is (MEDIUM) and the overall business impact is (MEDIUM). Therefore using table 6, from a technical prospective, it appears that the overall severity is MEDIUM and from a business prospective, the overall severity is MEDIUM.

Now that we have found the severity of each of the identified vulnerabilities, it is easy to make a decision on what to fix first. The identified severities play major role in the making of decisions during mitigation process. Table 5.7 summarizes the identified vulnerabilities, their corresponding severities, their impact on the organization and possible solutions.

Vulnerability type	Severity		Impact on the organization	Possible solutions
	Technical (prospective)	Business (prospective)		
Account storage	HIGH	MEDIUM	If location or storage of passwords is not secure, password file can be compromised.	Store password file in a secured location/storage .
Deficiency of password complexity	CRITICAL	MEDIUM	Easy passwords can easily be guessed	Use complex passwords
Weak input validation	HIGH	MEDIUM	If exploited, it can result in format string overflow attacks	Make sure inputs are properly validated
Lack of password retries limit	HIGH	MEDIUM	Guessing of account passwords	Limit password retry max. to three trial
Improper encoding of output	MEDIUM	MEDIUM	Read/Write of sensitive information	Insure that proper output encoding is done
Disclosing exception details	HIGH	MEDIUM	If exploited, it can result in buffer overflow attack	Make sure the exception details doesn't provide detailed information
Weak encryption keys	MEDIUM	MEDIUM	Information disclosure	Use of strong (secure) keys

Table 5.7 Severities, Impacts and solutions of the vulnerabilities detected threat risk modeling

Chapter 6

Discussion

As stated in section 1.1, it is claimed that, transparency makes systems more secure. That is vulnerabilities are found and fixed more quickly if the source code of an application is open to everyone, closing up security holes faster and this can increases software security. Anyone who is interested in improving the software is also free to create a better, more secure version of the software.
Trusting the above arguments of open source software advocates however, organizations might be tempted to deploy open source software before they evaluate the software for the presence of vulnerabilities. This may pose several security challenges to the organizations unless the software is evaluated for security vulnerabilities.
In sections 5.1 and 5.2, the results obtained using static source code analysis tools showed that the test case application, rdpdesk-3.2, contains large number of source code vulnerabilities. The results showed that flawfinder and RATS detected 3167 and 1282 vulnerabilities respectively. Of the total vulnerabilities detected using flawfinder, 87.81% were of the type buffer overflow, ~3% of the type format string overflow, and 2.02% of the type integer overflow. Race condition, random number and etc were the remaining ~8%. From the 1282 vulnerabilities detected with RATS, 61% were buffer overflow, 26.2% format string overflow, and 0% integer overflow. Race condition, random number, DNS forgery and etc. accounted for the rest ~13%. It is clear that if any of these vulnerabilities are exploited by an attacker, they can cause a serious damage (such as exposure of sensitive data, execution of malicious code etc.) to the organization using the application. As seen above, the static source code analysis tools detected large amount of vulnerabilities both in number and type wises. However, due to the limit in the number of rule sets defined, the tools still have some limitations on the types of vulnerabilities they can detect. For instance, RATS detected DNS-forgery, Trojan horse insertion and Time Of Check Time Of Use (TOCTOU) types of vulnerabilities on the test case application where flawfinder was not able to address them. On the other hand, flawfinder detected integer overflow, some shell and port related vulnerabilities that were not covered by RATS. More importantly, the tools are only capable of addressing flaws on the source codes. This happens because they are mainly designed for code reviewing. Despite of the individual limitation of the static source code analysis tools in analyzing all types of vulnerabilities however, I believe the combined results obtained from the these tools play an important role at improving the security of an application by evaluating the source code vulnerabilities. An important issue, especially to build a solid security strategy requires identification of all possible threats and vulnerabilities on an application. These include design and architectural vulnerabilities as well.

From the threat risk model developed in section 5.3, several design and architectural vulnerabilities were discovered in the application. Even though the method was exhaustive, the threat risk model

developed played a great role in the identification of design and architectural flaws that could not be addressed by the static source code analysis tools. For instance, lack of password retry limits and disclosing of exception details are some among the rest of the vulnerabilities that were addressed using the threat risk model developed. Note that, if some of the identified threats or vulnerabilities are found to be mitigated during the design or implementation stage, they can be easily omitted from the list of identified vulnerabilities. In this way, the threat risk model can be used to focus on the most dangerous vulnerabilities in a more efficient way.

Static source code analysis plays a major role in the detection of source code vulnerabilities of an application with short time as well as low costs (including some free of charge). The use of Threat risk modeling, on the other hand, is an interesting approach that can be used to explore and identify design and architectural vulnerabilities of an application. The combined use of both approaches can be used to improve the security of open source application by verifying the source code as well as design and architectural vulnerabilities.

Chapter 7

Conclusion and Future Work

The two approaches – static analysis and threat risk modeling play significant role in the detection of software security vulnerabilities. The combined result of both approaches can improve the security of an open source application by detecting source code vulnerabilities as well as design and architectural vulnerabilities. Static analysis is used in the detection of source code based vulnerabilities whereas threat risk modeling is used for addressing the design and architectural flaws in an application by iteratively assessing the application. In this project, I have tried to use both approaches on a test case open source application, rdpdesk-3.2, to improve its security level.

The result obtained from the use of the two static analysis tools, flawfinder and RATS, showed that the tools detected varied types of vulnerabilities on the application. This happens because the tools are programmed to detect different rule sets. On the other hand, using Microsoft Threat Analysis & Modeling v2.1.2 several design and architectural flaws were detected on the test case application. This approach was used to address the structural vulnerabilities that would otherwise have been impossible to detect them using the static analysis approach.

Therefore, organizations that deploy open source software on their software stack can benefit greatly by adopting the security best practices used in this project.

Static analysis tools are not precise that is they generate some false positives and false negatives. Further work could be done on the identification of these false positives and false negatives and omit them from the results in order to have precise static analysis results and focus on mitigating existing vulnerabilities instead of wasting resources on non-existent vulnerabilities.

Appendices

Appendix A Glossary and Abbreviations

Glossary

Analysis - this process as a method of studying the nature of something or of determining its essential features and their relations.

Argument - an element to which an operation, function, predicate, etc, applies, esp the independent variable of a function.

Attack - any attempt to destroy, expose, alter, disable, steal or gain unauthorized access to or make unauthorized use of an asset.

Buffer overflow - What happens when you try to store more data in a buffer than it can handle.

Bug - an error or fault, as in a machine or system, esp in a computer or computer program.

Closed source - intellectual property, esp computer source code, that is not made available to

Compiler - a computer program that translates a program written in a high-level language into another language, usually machine language. Compare interpreter.

Data - a single piece of information, as a fact, statistic, or code; an item of data.

Error - a deviation from accuracy or correctness; a mistake.

Execute - to carry out; complete; perform; do: to execute an order.

Expert - a person who has special skill or knowledge in some particular field.

Exploit - to take advantage of (a person, situation, etc), esp unethically or unjustly for one's own ends.

Firewall - an integrated collection of security measures designed to prevent unauthorized electronic access to a networked computer system.

Flaw - a defect impairing legal soundness or validity.

Insider - a person with access to exclusive information.

IT - the development, implementation, and maintenance of computer hardware and software systems to organize and communicate information electronically.

License - permission to do or not to do something.

Malicious - motivated by wrongful, vicious, or mischievous purposes.

Memory - a part of a computer in which information is stored for immediate use by the central processing unit.

Open source - pertaining to or denoting software whose source code is available freely.

Operating system – a collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Operator - A symbol used as a function, with infix syntax if it has two arguments (e.g. "+") or prefix syntax if it has only one (e.g. Boolean NOT). Many languages use operators for built-in functions such as arithmetic and logic.

Parameter - a variable that must be given a specific value during the execution of a program or of a procedure within a program.

Peer review - evaluation of a person's work or performance by a group of people.

Programming - the act or process of writing a computer program.

Public domain - able to be discussed and examined freely by the general public.

Secret - kept from the knowledge of any but the initiated or privileged: a secret password.

Security - something that secures or makes safe; protection; defense.

Software - the programs used to direct the operation of a computer, as well as

Source Code - program instructions that must be translated by a compiler, interpreter, or assembler into object code before execution.

Stack - an area in a computer memory for temporary storage.

Technical - belonging or pertaining to an art, science, or the like: technical skill.

Threat - an indication or warning of probable trouble.

Time bomb - A subspecies of logic bomb that is triggered by reaching some preset time, either once or periodically.

Tool - anything used as a means of performing an operation or achieving a result.

Upgrade - a new version, improved model.

Vulnerability - open to moral attack, criticism, temptation, etc.: an argument vulnerable to refutation.

Worm - a program that duplicates itself many times in a network and prevents its destruction.

It often carries a logic bomb or virus.

List of Abbreviations

AST	Abstract Syntax Tree
CERT	Computer Emergency Response Team
DNS	Domain Name Server
DoS	Denial-of-Service

GDI	Graphics Device Interface
GPL	General Public License
HTML	Hyper Text Markup Language
ICA	Independent Computing Architecture
IT	Information Technology
IIS	Internet Information Services
LGPL	Lesser General Public License
MiM	Man in the Middle
MITRE	Massachusetts Institute Of Technology Research And Engineering
MS ISA	Microsoft Internet Security and Acceleration
OS	Operating System
OSC	Open Source Components
OSE	Open Source Edition
OSI	Open Source Initiative
OWASP	Open Web Application Security Project
PCI	Peripheral Component Interconnect
RATS	Rough Auditing Tool for Security
RC4	Ron's Code4 (RSA Variable-Key-Size Encryption Algorithm - Ron Rivest)
RDP	Remote Desktop Protocol
RSA	Rivest, Shamir, and Adleman
SANS	System Administration Networking and Security
SEI	Software Engineering Institute
SLA	Service Level Agreement
SLOC	Source Lines Of Codes
SQL	Structured Query Language
STRIDE	Spoofing, Tampering, Repudiation, Information Disclosure, Denial-of-Service, Elevation of Privilege.
TOCTOU	Time Of Check Time Of Use
URL	Uniform Resource Locator
VNC	Virtual Network Computing
XSS	Cross Site Scripting

Appendix B System setups

Flawfinder

1. Install Linux/Ubuntu
2. Download flawfinder to your system by opening a terminal window and executing the following command;
   sudo apt-get install flawfinder
3. Unpack the software;
   tar –xvsf flawfinder-.tar* ; where * is the version of the software.
4. After unpacking it change directory to flawfinder-*.
   *cd /path/to/flawfinder-**
5. Install the software by issuing the following command
   make install
6. After the software is installed, issue the following command to start the scanning;
   ./flawfinder /path/to/target-folder

If you want save the analysis result in a text file, issue the following command;
./flawfinder /path/to/target-folder >> file-name.txt

RATS

The setup for using RATS however requires installation of expat in your system in order for RATS to be installed successfully.

1. Download expat-2.0.1.tar from;
   http://sourceforge.net/projects/expat/
2. Unpack the software;
   tar –xvsf expat-2.0.1.tar
3. After unpacking it change directory to expat-2.0.1.
   cd /path/to/ expat-2.0.1
4. Finally install expat by issuing the following command;
   make install
5. Download rats-2.3.tar.gz to your system from;
   https://www.fortify.com/ssa-elements/threat-intelligence/rats.html
6. Unpack the software;
   tar –xvsf rats-2.3.tar.gz
7. After unpacking it change directory to rats-2.3.tar.gz.
   *cd /path/to/*rats-2.3.tar.gz
8. Install the software by issuing the following command
   make install

9. After the software is installed, issue the following command to start the scanning;
./rats /path/to/target-folder

If you want save the analysis result in a text file, issue the following command;
./rats /path/to/target-folder >> file-name.txt

Appendix C Static source code analysis tools used for the experiment

1. Flawfinder

Flawfinder is an open source program that scans source code in order to detect possible security vulnerabilities. The vulnerabilities detected are organized according to risk level.

Flawfinder is a very useful tool that can be used to quickly find and get rid of some vulnerability before a program is released to the public for use.

The tool functions on Unix as well as Linux systems. One can decide to port it to be used on windows systems. This analyzing tool needs Python 1.5 or a greater version to run.

Flawfinder was written to boost the use of static source code analyzers to find security vulnerabilities in programs. The program is made available under General Public License (GPL) version 2, hence it is open source software as defined by the Open Source Definition.

Flawfinder Functions

The tool contains a built-in database of C/C++ functions with well-known security problems such as buffer overflow risks, format string problems, race conditions and poor random number acquisition.

Some of the buffer overflow risks in the database are: strcpy(), strcat(), gets(), sprintf(), and that of format string problems are : printf(), snprintf(), and syslog(). The race conditions problems are: access(), chown(), chgrp(), chmod(), tmpfile(), tmpnam(), tempnam(), and

mktemp(), whiles random() is a poor random number acquisition problem.

The tool works by using the database and the good aspect is that the database comes with the analysis tool. The tool takes the source code text and performs matching with the database problems. However text inside comments and strings are ignored.

Flawfinder is aware of gettext; a common library for international programs. Hence it treats constant strings passed through gettext as if they were constant strings. This approach helps to lessen the false hits in international programs.

The list of potential security flaws produced by the tool is called “hits”, and they are organized according to the risk level; the riskiest are placed at the top of the list. The risk level is not only determined by the function, but also the value of the parameters of the function.

For instance, constant strings are considered less risky than fully variable strings in many situations. Sometimes, Flawfinder determines that a particular construct is not risk, thus reducing false positives. The tool gives better information and prioritization other than just running test on the source code. It has the capability to ignore comments and the insides of strings as well as analyze parameters to determine the risk levels. However, this does mean that Flawfinder is a perfect tool. It does have the knowledge of data types of function parameters and does not do data flow analysis.

Actually, not all the problems that can be detected using this tool are security vulnerabilities and it is also true that not all vulnerabilities can be detected using Flawfinder. Moreover, the tool does not really understand the semantics of codes. What it does is to carry out simple text pattern matching whiles it ignores comments and strings. This notwithstanding, Flawfinder as a static analysis tool can be very useful in locating and removing vulnerabilities in software.

Reviewing of Patches

When there is a need to review only a set of changes that were made to a program instead of the reviewing the whole program, one can still use Flawfinder to do that. If the changes made are confined to a section of the code, the review can be done manually, otherwise it is difficult. There is an added automated support in Flawfinder 1.27 that enables it to review only the changes in the program.

In order to do this, one has to first create a “Unified Diff” [76] file and compare the older version to the current version. This is followed by running Flawfinder on the newer version and that give it “--patch (-P)” [76] option that points to the patch file.

This will work since Flawfinder will perform its function. However, it will only detect hits that are in the patch file. The tool functions by reading the unified diff file in order to detect the files that have been changed and the lines that have changed in the code. Precisely, it uses

"Index:" or "+++" lines to ascertain the files have been changed. The tool then uses the line numbers in "@@" regions to find out the chunk line number ranges, and it then uses the initial +, -, and space after that to determine which lines really have changed.

One of the challenges in this case has to do with statements that span more than one line. The statement might begin from one line and extend to another line. A change on the second line introduces vulnerability on the second line. Flawfinder deals with this situation by displaying security vulnerabilities that are reported one line before or after any line that has changed.

This seems to be acceptable compromise. However, the problem is that Flawfinder will not detect if the code enforcing security requirement is deleted since it does not have the capability to do that.

Hit Density

Hit density is the number hits per thousand lines of the source code. It is a useful relative indicator of the likelihood of security vulnerabilities in source codes. This is one of the metrics used by Flawfinder in reporting vulnerabilities.

The security level of two source codes can be determined by comparing their hit density levels. This can be done by running Flawfinder test on the two source codes to determine the hit densities of the two. The one with a higher hit density is the one with a worse security record even if none or few of the cases reported were actually security problems.

High hit density in a program is an indicator that the programmer used very dangerous construct which are difficult to use in the right way and that most often than not lead to vulnerabilities. The hits might not necessarily be vulnerabilities but the fact is that programmers who continuously use such constructs are likely to make mistakes to create conditions for vulnerabilities.

When comparing two source codes, it is necessary to take the sizes into consideration. This is because a program with a smaller size will have a higher hit density when compared with a much larger size program. In this case, one cannot conclude that the code with a larger size is more secure. This is in view of the fact that density is a fraction and size is the denominator.

However, it is easier to assess programs that are smaller; therefore, it is likely that such programs can be reviewed directly in order to detect vulnerabilities.

2. RATS

The word RATS is an acronym for Rough Auditing Tool for Security. This open source analyzing tool was developed by Secure Software security engineers. The company was acquired by Fortify Software, Inc. RATS as a static analysis tool can be used to scan source codes written in C, C++, Perl, PHP and Python. The results of the scan highlights common security related programming errors such as buffer overflows and TOCTOU race conditions.

Using RATS to scan source code produces result that list potential trouble spots on which a security analyst needs to focus. The result also gives a description of the problem and potentially suggested remedies. It also shows a relative assessment of the potential severity of each of the problem captured in the result. This helps in analyzing the result of the scan. One of the good features of RATS is that it performs some basic analysis to rule out conditions that are obviously not problems.

The tool does not have the capability to find every error. It also detects other problems that are not vulnerabilities. In fact it only performs a rough analysis of source code. Therefore manual inspection of code is still necessary, but the process is greatly improved with the use of this tool.

RATS static analysis tool is free software, thus it can be copied, distributed, and modified as permitted by the terms of the GNU Public License. The latest edition that was released is version 2.3.

In order to use RATS, one needs to install expat. It is often installed in /usr/local/lib and /usr/local/include. However, there is a need to specify --with-expat-lib and --with-expatinclude options to configure on some systems, so that it can find the installation of the library and header.

One can simply run make in the distribution's top-level directory to build the program after the configuration script has finished successfully. Once this has been done and the tool installed correctly, it can be used to scan source codes.

RATS is used in the command line mode and it accepts few command line options as described below. The tool also accepts a list of files to scan on the command line. In case no files to scan have been specified, it will use “stdin”.

The command options that be used on the command line of this static analysis tool are: rats

[-d] [-h] [i] [l] [-r] [-w] [-x] [file1 file2 ... file*N*].

The command options have different functions as explained below:

Rats [-d] command option specifies a vulnerability database to be loaded. There may be multiple -d options and each of these databases that is specified will be loaded.

Rats [-h] command option will display a brief summary of the usage.

Rats[-i] command option will cause a list of used function calls which accept external input to be displayed after the vulnerability report has been generated.

Rats [-l] option command specifies the language to be used irrespective of the filename extension. The valid language names that are currently used are: "c", "perl", "php" and "python".

Rats [-r] command option will cause references to vulnerable function calls that are not being used as calls themselves to be generated.

Rats [-w] command option will set the warning level. The valid warning levels are 1, 2 or 3.

Level 1 includes only default and high severity. Level 2 means medium severity. Level 2 is the default warning and level 3 indicates low severity vulnerabilities.

Rats [-x] command option will cause the tool not to load the default vulnerability databases (which are in the installation data directory, /usr/local/lib by default).

The tool is not difficult to operate. When it is started, it scans each file that is specified on the command line and then produces a report when it has finished scanning the source code. The types of vulnerabilities that are reported in the final report depend on the data in the vulnerability database or databases that are loaded and the warning level that is in use.

The output of the analysis is not difficult to understand. For each of the vulnerabilities that are captured in the report, a list of files and the line numbers of the code where the vulnerability occurred is given. To make analysis of the result easy, a brief description of the vulnerabilities and suggested actions are provided. [77]

Appendix D rdpdesk-3.2 Threat risk Model

Business Objectives | Roles | Components | External Dependencies | Data | Application Use Cases | Confidentiality Threats | Integrity Threats | Availability Threats

Threat Model Information	
Name	Rdpdesk-3.2 Threat Model
Description	This is a threat risk model that was developed for a test case application rdpdesk-3.2

Business Objectives

The business objectives define the objectives behind the implementation of the system. These objectives are supported by use cases.

Business Objectives

Roles

Roles define the privilege levels users have. They are logical groups of users who use the application, and can perform the necessary functions defined by the application. These groups are then divided into two sub groups, User Roles and Service Roles. User Roles contain users who interact with the application. Service Roles contain users under which services are executed.

User Roles			
Name	Description	Auth. Mechanism	# of Identities
Administrator			
Client			

Service Roles			
Name	Description	Auth. Mechanism	# of Identit
C/C++ Role			
Windows7 Role			
Microsoft SQL server 2008 Role			
RSA Security RC4 cipher Role			
Microsoft Windows terminal services (RDP) Role			
Citrix ICA Role			
VNC Role			

Components

Components define the high level building blocks of the system. Components are decomposed logically into services and objects through which the user interacts with the system. Both components and objects can talk to each other in order to fulfill a user action.

Components

Name	Roles	Type	Tech. Type	Run As
Microsoft SQL server 2008				Microsoft SQL server 2008 Service Account
RSA Security RC4 cipher				RSA Security RC4 cipher Service Account
Microsoft Windows terminal services (RDP)				Microsoft Windows terminal services (RDP) Service Accou
Citrix ICA				Citrix ICA Service Account
VNC				VNC Service Account

External Dependencies

External dependency is a type of component which is external to the application and application does not have control over the behavior or the implementation of the dependency.

External Dependencies

Name	Description	Dependency Type

Data

Data can be logically broken down into data elements grouped based on the classification. Data is actively stored in one or more components which can be accessed by the user.

Data

Name	Description	Data Elements	Data Classification	Data Stores
User Account				

Access Control

Role	Access Control	Condition
Administrator	**C R U D**	
Client	**U**	

Logging Response

Access Control

Role	Access Control	Condition
Administrator	**C R U D**	
Client	**R**	

Message

Access Control

Role	Access Control	Condition
Administrator	**C R U D**	
Client	**R**	

software

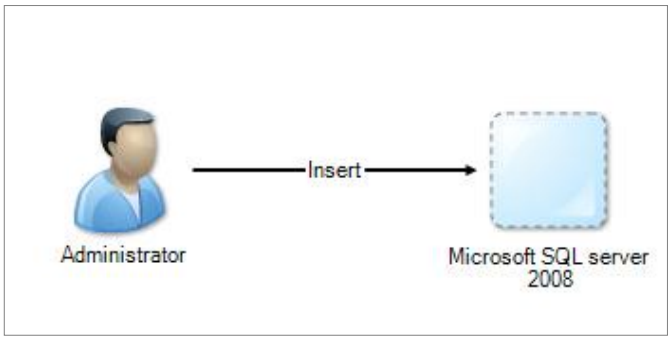
Access Control					
Role	Access Control				Condition
Administrator	**C**	**R**	**U**	**D**	
Client		**R**			

Application Use Cases

Use cases are fulfilled by a sequence of calls. Each use case is targeted towards a specific business objective. Each call captures the user's action on a component. Each use case achieves a net data effect on a data.

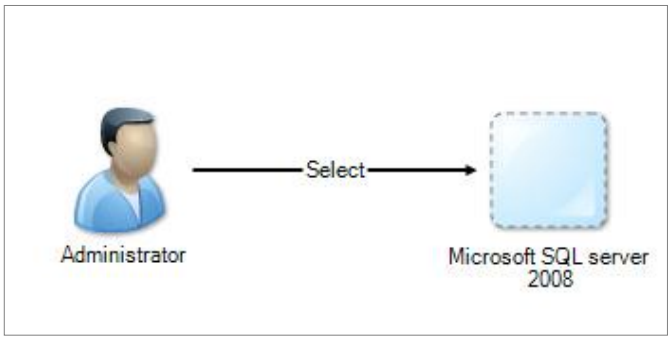
Application Use Cases			
Name	Description	Roles	Net Data Effect
Administrator Create User Account		Administrator	**C** User Account

Calls			
Caller	Action	Component	Authorization
Administrator	Insert	Microsoft SQL server 2008	Delegate Caller



Administrator Read User Account	Administrator	**R** User Account

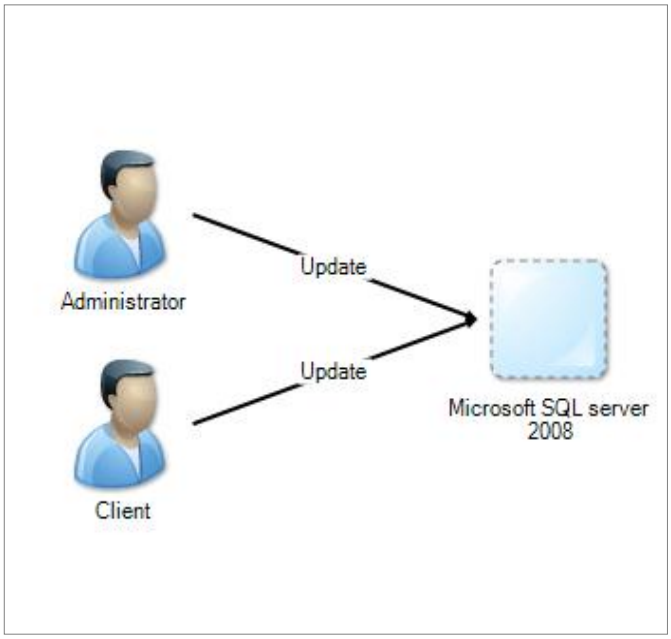
Calls			
Caller	Action	Component	Authorization
Administrator	Select	Microsoft SQL server 2008	Delegate Caller



Administrator, Client Update User Account	Administrator	**U** User Account

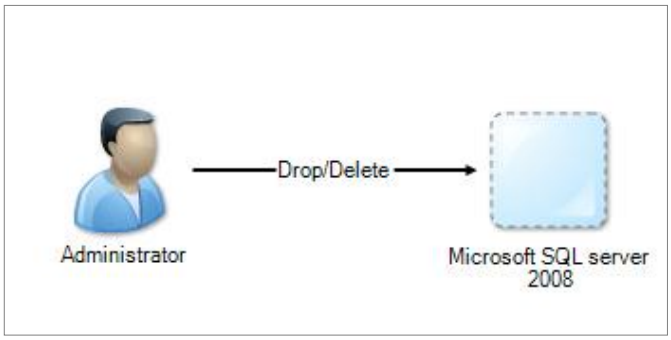
Client

Calls			
Caller	Action	Component	Authorization
Administrator	Update	Microsoft SQL server 2008	Delegate Caller
Client	Update	Microsoft SQL server 2008	Delegate Caller



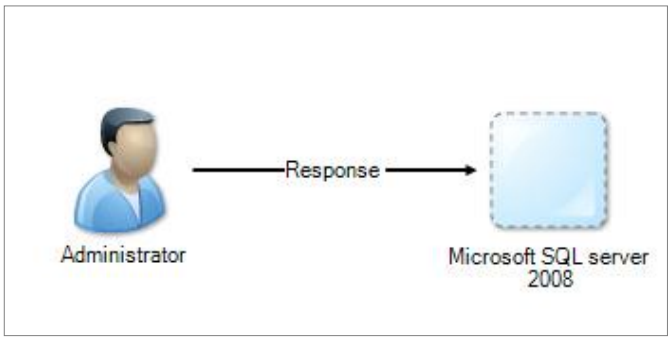
Administrator Delete User Account Administrator **D** User Account

Calls			
Caller	Action	Component	Authorization
Administrator	Drop/Delete	Microsoft SQL server 2008	Delegate Caller



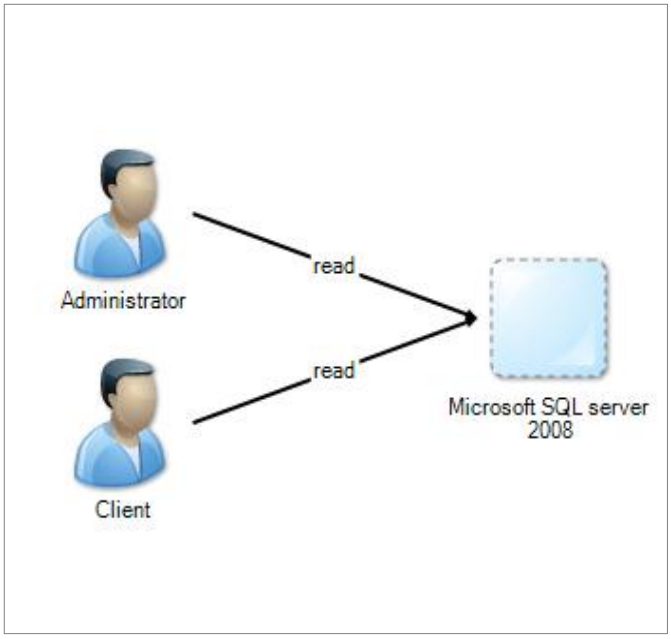
Administrator Create Logging Response Administrator **C** Logging Response

Calls			
Caller	Action	Component	Authorization
Administrator	Response	Microsoft SQL server 2008	Delegate Caller



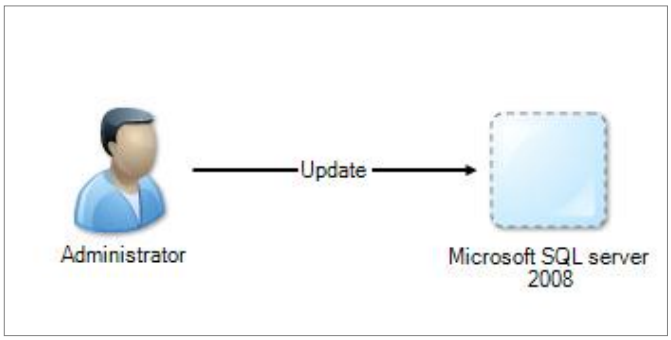
Administrator, Client Read Logging Response | Administrator
Client | **R** Logging Response

Calls			
Caller	Action	Component	Authorization
Administrator	read	Microsoft SQL server 2008	Delegate Caller
Client	read	Microsoft SQL server 2008	Delegate Caller



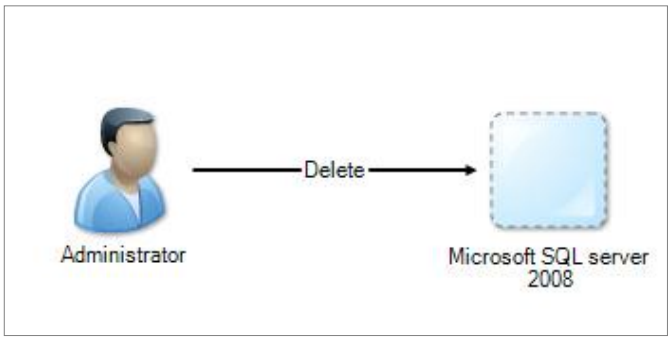
Administrator Update Logging Response | Administrator | **U** Logging Response

Calls			
Caller	Action	Component	Authorization
Administrator	Update	Microsoft SQL server 2008	Delegate Caller



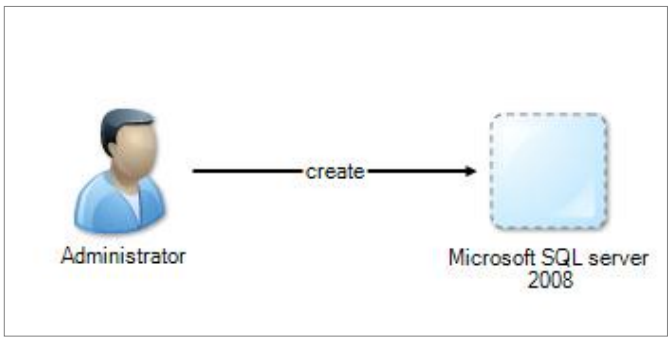
Administrator Delete Logging Response | Administrator | **D** Logging Response

Calls			
Caller	Action	Component	Authorization
Administrator	Delete	Microsoft SQL server 2008	Delegate Caller



Administrator Create Message | Administrator | **C** Message

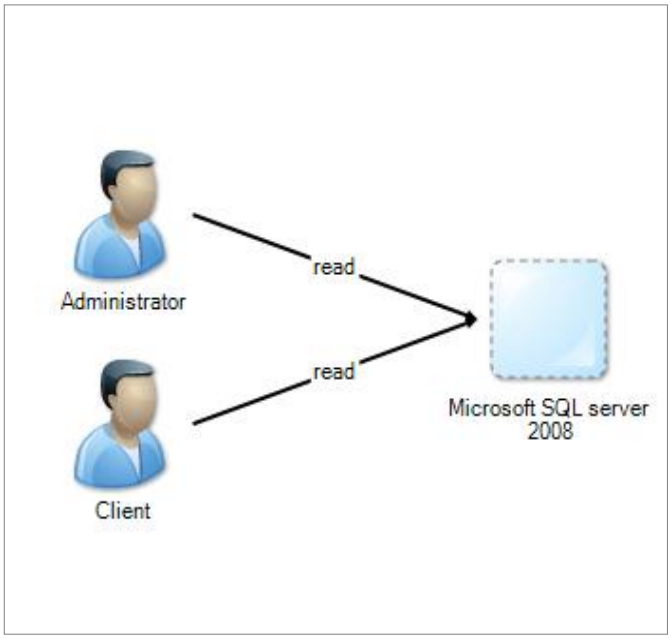
Calls			
Caller	Action	Component	Authorization
Administrator	create	Microsoft SQL server 2008	Delegate Caller



Administrator, Client Read Message | Administrator
Client | **R** Message

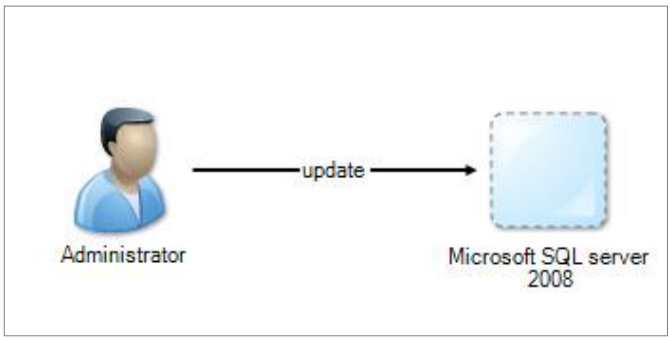
Calls			
Caller	Action	Component	Authorization

Administrator	read	Microsoft SQL server 2008	Delegate Caller
Client	read	Microsoft SQL server 2008	Delegate Caller



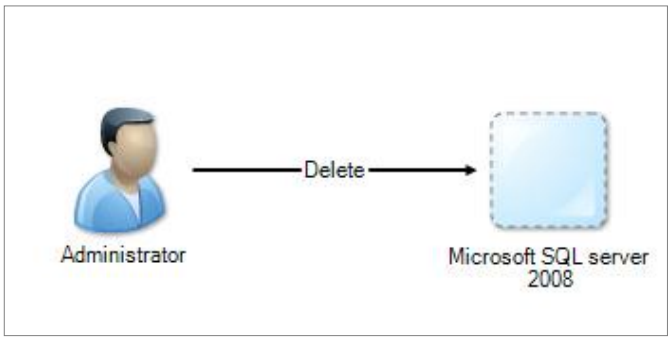
Administrator Update Message Administrator **U** Message

Calls			
Caller	Action	Component	Authorization
Administrator	update	Microsoft SQL server 2008	Delegate Caller



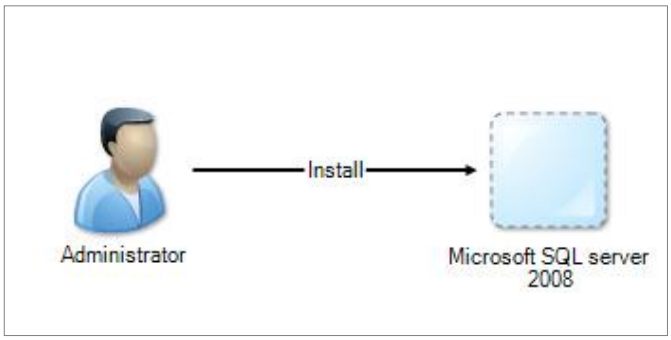
Administrator Delete Message Administrator **D** Message

Calls			
Caller	Action	Component	Authorization
Administrator	Delete	Microsoft SQL server 2008	Delegate Caller



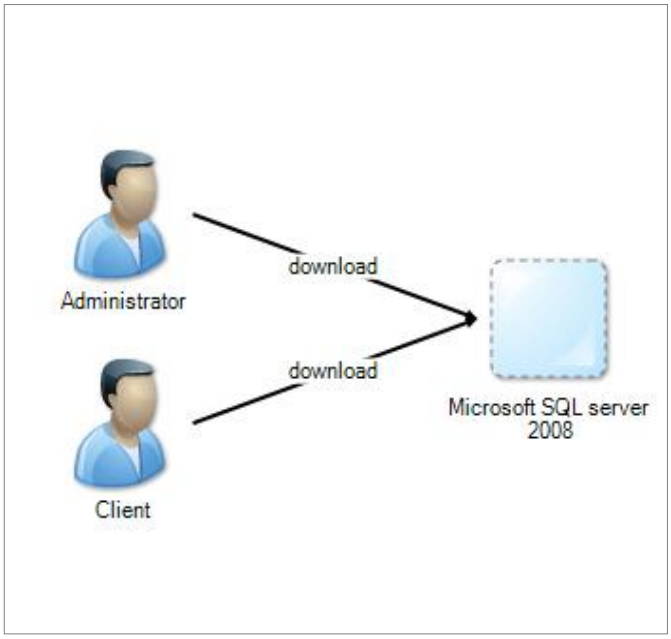
Administrator Create software Administrator **C** software

Calls			
Caller	Action	Component	Authorization
Administrator	Install	Microsoft SQL server 2008	Delegate Caller



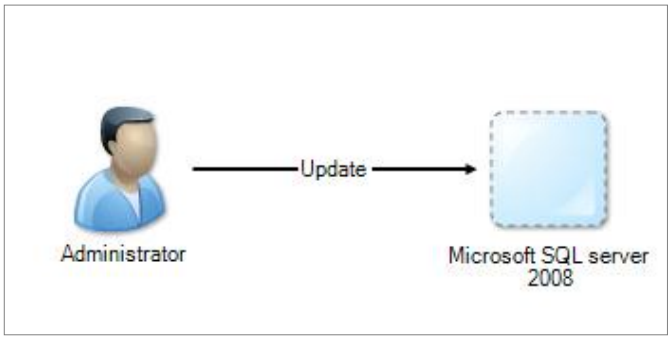
Administrator, Client Read software Administrator Client **R** software

Calls			
Caller	Action	Component	Authorization
Administrator	download	Microsoft SQL server 2008	Delegate Caller
Client	download	Microsoft SQL server 2008	Delegate Caller



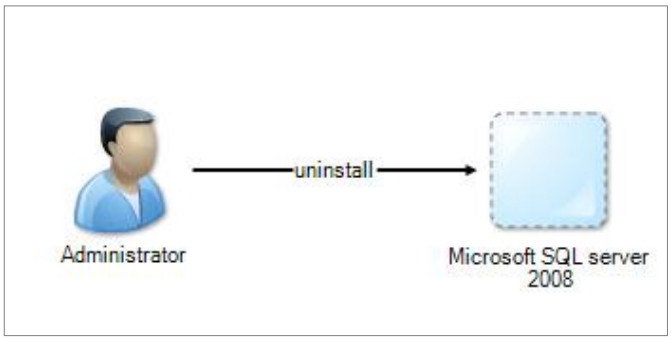
Administrator Update software Administrator **U** software

Calls			
Caller	Action	Component	Authorization
Administrator	Update	Microsoft SQL server 2008	Delegate Caller



Administrator Delete software Administrator **D** software

Calls			
Caller	Action	Component	Authorization
Administrator	uninstall	Microsoft SQL server 2008	Delegate Caller



Threats

A threat is defined as an undesired event, a potential occurrence, often best described as an effect that might damage or compromise an asset or objective. It may or may not be malicious in nature.

Confidentiality Threats

Unauthorized disclosure of <Insert> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Select> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)

- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Update> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs

- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Drop/Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Response> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <read> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <read> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging

- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <create> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <read> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <read> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user

- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Install> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement

- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <download> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <download> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames

- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Unauthorized disclosure of <uninstall> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Integrity Threats

Illegal execution of <Insert> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user

- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Select> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement

- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Update> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Drop/Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Response> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames

- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <read> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <read> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding

- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <create> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <read> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <read> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user

- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement

- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Install> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <download> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <download> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames

- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Illegal execution of <uninstall> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding

- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Availability Threats

Ineffective execution of <Insert> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Select> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement

- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Update> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Drop/Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames

- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Response> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <read> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding

- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <read> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <create> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <read> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user

- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <read> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement

- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Delete> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Install> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <download> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames

- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <download> using <Microsoft SQL server 2008> by <Client>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <Update> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding

- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

Ineffective execution of <uninstall> using <Microsoft SQL server 2008> by <Administrator>

Countermeasures

- Canonicalization : Only accept primitive typed identified (e.g., integers) which are mapped to filenames
- HTTP Replay Attack : Provide a secure end-to-end communication channel between server and client (e.g., SSL)
- One-Click Attack : Make the request that encapsulates the users action unique for each authenticated user
- Repudiation Attack : Implement proper and effective logging
- Response Splitting : Perform context-sensitive encoding
- Session Hijacking : Use strong random numbers for session IDs
- SQL Injection : Use parameterized SQL statement
- SQL Injection : Use stored procedure with no dynamic SQL

References

[1] Karen Mercedes Goertzel "Introduction to software security" 09-01-2009
URL: https://buildsecurityin.us-cert.gov/bsi/547-BSI.html

[2] "Defining Open Source Software (OSS)", accessed 01-02-2012
URL:
http://dodcio.defense.gov/sites/oss/Open_Source_Software_%28OSS%29_FAQ.htm#Q:_What_is_open_source_software_.28OSS.29.3F

[3] "What is Open Source? ", Search enterprise Linux.
URL: http://searchenterpriselinux.techtarget.com/definition/open-source

[4] "An overview of open source software license"
URL:
http://www.americanbar.org/groups/intellectual_property_law/resources/an_overview_of_open_source_sware_licenses.html

[5] "Open source initiative"
URL: http://www.opensource.org/licenses/alphabetical
http://www.opensource.org/

[6] Michael Bloch. "The principles of Open Source Software."
URL: http://www.tamingthebeast.net/articles5/open-source-software.htm

[7] Chris Bell, "how secure is open source" 27-02-2006
URL:http://computerworld.co.nz/news.nsf/spec/38856F521C723CC7CC25711F001106A8

[8] David A. Wheeler, "secure programming for Linux and Unix", v3.010, 3 March 2003
URL:http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/open-source-security.html

[9] SANS institute InfoSec reading room, "Security Concerns in Using Open Source Software for Enterprise Requirements"
URL:
http://www.sans.org/reading_room/whitepapers/awareness/security-concerns-open-source-software-enterprise-requirements_1305

[10] Elias Levy "Is Open Source really more secure than closed?" 17-04-2000
URL: http://www.securityfocus.com/news/19

[11] Bruce schneier "Open source and security", 15-09-1999
URL: http://www.schneier.com/crypto-gram-9909.html#OpenSourceandSecurity

[12] Gary McGraw. "Software security Building security in" 01-2006 Addison-wesley software security series. Pages 4-6, 42-46

[13] Mile Lennon. "Internet TVs - The Latest Attack Vector: Researchers Hack Internet Enabled TVs, Discover Multiple Security Vulnerabilities", 03-01-2011.
URL:http://www.securityweek.com/researchers-hack-internet-enabled-tvs-discover-multiple-security_vulnerabilities

[14] Safe code. "Software assurance: an overview of current industry best practices." 02-2008
URL: http://www.safecode.org/publications/SAFECode_BestPractices0208.pdf

[15] "Open web application security projects" 09-12-2011
URL: https://www.owasp.org/index.php/Category:Vulnerability

[16] "Buffer overflow." 21-02-2009
URL: https://www.owasp.org/index.php/Buffer_Overflow

[17] Alexander Ivanov Sotirvo. "Automatic vulnerability detection using static code analysis", 2005

URL: http://gcc.vulncheck.org/sotirov05automatic.pdf
[18] "Buffer Overflow in Sendmail" Advisory CA-2003-12, 29-05-2003
URL: http://www.cert.org/advisories/CA-2003-12.html
[19] Carsten Eiram, "Microsoft IIS FTP Server NLST Buffer Overflow Clarifications" 02-09-2009
URL: http://secunia.com/blog/62/
[20] Michael Howard "SDL banned function calls". 06-2011
URL: http://msdn.microsoft.com/en-us/library/bb288454.aspx
[21] "Integer overflow", 22-05-2009
URL: https://www.owasp.org/index.php/Integer_overflow
[22] David LeBlanc. "Integer Handling with the C++ SafeInt Class." 07-01-2004.
URL: http://msdn.microsoft.com/en-us/library/ms972705.aspx
[23] "Format string problem", 23-02-2009
URL: https://www.owasp.org/index.php/Format_String
[24] Tim Newsham "Format string attacks", 09-2000
URL: http://hackerproof.org/technotes/format/FormatString.pdf
[25] Ulfar Erlingsson. "Low-level Software Security:Attacks and Defenses" 11-2007
URL: http://research.microsoft.com/pubs/64363/tr-2007-153.pdf
[26] Gary McGraw. "Software [In]Security: Attack Categories and History prediction" 25-08-2009
URL: http://www.informit.com/articles/article.aspx?p=1393066
[27] Screen OS Reference Guide. "Attack Detection and Defense Mechanisms" Release 6.0.0, Rev. 03
URL: http://www.juniper.net/techpubs/software/screenos/screenos6.0.0/CE_v4.pdf
[28] Brien M. Posey. "Buffer-overflow attacks: How do they work?" 31-01-2005
URL: http://searchsecurity.techtarget.com/news/1048483/Buffer-overflow-attacks-How-do-they-work
[29] "Heap overflow". 21-02-2009
URL: https://www.owasp.org/index.php/Heap_overflow
[30] "Stack overflow". 28-02-2009
URL: https://www.owasp.org/index.php/Stack_overflow
[31] "Cross-site Scripting (XXS)". 12-08-2011
URL: https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29
[32] "Cross-site Scripting attacks", Acunetix web application security
URL: http://www.acunetix.com/websitesecurity/cross-site-scripting.htm
[33] "SQL-injection" 12-06-2011
URL: https://www.owasp.org/index.php/SQL_Injection
[34] "SQL-injection: what is it? ", Acunetix web application security
URL: http://www.acunetix.com/websitesecurity/sql-injection.htm
[35] Stephen Kost. "An Introduction to SQL Injection Attacks for Oracle Developers" 01-2004
URL:http://www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf
[36] "Understanding denial-of-service attacks" US-CERT
URL: http://www.us-cert.gov/cas/tips/ST04-015.html
[37] Gary McGraw, "Software security: The trinity of trouble" 15-02-2006
URL: https://freedom-to-tinker.com/blog/gem/software-security-trinity-trouble
[38] Steve McConnell, "Diseconomies of Scale and Lines of Code" 17-07-2006
URL: http://www.codinghorror.com/blog/2006/07/diseconomies-of-scale-and-lines-of-code.html
[39] Mark Stamp, "Information Security Practices and Principles" 2006 JohnWiley & Sons. Pages 180-190, 267-271.
[40] Todd Landry. "The Alphabet Soup of Software Security Guidelines" 15-06-2010
URL:http://www.klocwork.com/blog/2010/06/the-alphabet-soup-of-software-security-guidelines/
[41] OWASP Foundation "OWASP Secure Coding Practices Quick Reference Guide" Version2.0 2010.

URL:
https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide
[42] Viega, John & McGraw, Gary." Building Secure Software: How to Avoid Security Problems the Right Way". Boston, MA: Addison-Wesley, 2002.
[43] Sean Barnum & Michael Gegick. "Securing the weakest link" 19-09-2005
URL: https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/principles/356-BSI.html
[44] Carlos Ballester Lafuente. "Evaluating Security in Open Source Consumer Applications" 05-2007
URL: http://ntnu.diva-portal.org/smash/get/diva2:348445/FULLTEXT01
[45] "Defense in Depth" SearchSecurity 05-2007
URL: http://searchsecurity.techtarget.com/definition/defense-in-depth
[46] Sean Barnum & Michael Gegick. "Defense in depth" 13-09-2005
URL: https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/principles/347-BSI.html
[47] Eric vanderburg. "Fail secure - the correct way to crash", JurInnov security spotlight. 05-01-2011
URL: http://www.jurinnov.com/fail-secure-the-correct-way-to-crash
[48] Sean Barnum & Michael Gegick. "Failing securely" 05-12-2005
URL: https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/principles/349-BSI.html
[49] Jeff Langford. "Implementing Least Privilege at your Enterprise" SANS institute InfoSec reading room. 05-07-2003
URL: http://www.sans.org/reading_room/whitepapers/bestprac/implementing-privilege-enterprise_1188
[50] "Bell-LaPadula" computer security a brief look
URL: http://sites.google.com/site/cacsolin/bell-lapadula
[51] Ross Anderson. "Security Engineering: A Guide to Building Dependable Distributed Systems", First edition chapter 8: multilateral security
URL: http://www.cl.cam.ac.uk/~rja14/Papers/SE-08.pdf
[52] Gary McGraw and John Viega. "Keep it Simple" Dr.Dobb's the world of software development. 01-05-2003
URL: http://drdobbs.com/article/print?articleId=184414984&siteSectionName=
[53] William Jackson. "Static vs. dynamic code analysis: advantages and Disadvantages." 09-02-2009.
URL: http://gcn.com/Articles/2009/02/09/Static-vs-dynamic-code-analysis.aspx?Page=1 Accessed on 12 April 2011
[54] Nathaniel Ayewah et al. "Using Static Analysis to Find Bugs." IEEE software 2008
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4602670
[55] Viega, John, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. 2002. "ITS4: A static vulnerability scanner for C and C++ code."
URL:
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=898880 http://www.acsac.org/2000/papers/78.pdf
[56] Joel Jones. "Abstract Syntax Tree Implementation Idioms"
URL: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf
[57] Foster, Jeffrey. "Type qualifiers: Lightweight specifications to improve software quality." Ph.D. thesis, University of California, Berkeley 2002.
URL: http://www.cs.umd.edu/~jfoster/papers/thesis.pdf
[58] Jeff Foster et al. "CQUAL" 30-11-2004
URL: http://www.cs.umd.edu/~jfoster/cqual/
[59] "Proceedings of the10th USENIX Security Symposium" USENIX Association (13-17)-08-2001
URL: http://research.microsoft.com/pubs/74359/01-shankar.pdf
[60] James Nesome & Dawn Song "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software" 07-2005.
URL: http://bitblaze.cs.berkeley.edu/papers/taintcheck-full.pdf
[61] Michiel Scholten. "Taint Analysis in Practice" 11-12-2007
URL: http://aquariusoft.org/files/bachelorproject.pdf
[62] Dumitru CEARA. "Detecting Software Vulnerabilities Static Taint Analysis" 09-2009.

URL: http://tanalysis.googlecode.com/files/DumitruCeara_BSc.pdf
[63] S Vidalis and A Blyth. "Understanding and Developing a Threat Assessment Model" 10-2002
URL: http://comp.glam.ac.uk/staff/svidalis/.../threat%20methodology.doc
[64] James bayne. "An Overview of Threat and Risk Assessment." SANS institute InfoSec reading room. Version 1.2f, 2002
URL: http://sans.org/reading_room/.../overview-threat-risk-assessment_76
[65] "Threat Risk Modeling" 09-29-2010
URL: https://www.owasp.org/index.php/Threat_Risk_Modeling
[66] Shawn Hernan, Scott Lambert, Tomasz Ostwald and Adam Shostack "Uncover Security Design Flaws Using The STRIDE." MSDN magazine 11-2006.
URL: http://msdn.microsoft.com/en-us/magazine/cc163519.aspx
[67] J.D. Meier, Alex Mackman, Blaine Wastell. "Threat Modeling Web Applications" Microsoft Corporation. 05-2005
URL: http://msdn.microsoft.com/en-us/library/ff648006.aspx
[68] Danny Dhillon. "Developer-Driven Threat Modeling" Security & Privacy, IEEE (07-08)-2011
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5765924
[69] Jeffry A et al. "Threat Modeling: Diving into the Deep End." Software IEEE volume 25 issue 1, 01-2008
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4420064
[70] Steven F Burns. "Threat Modeling: A Process To Ensure Application Security." Version 1.4c, SANS INSTITUTE 05-01-2005
URL: http://www.sans.org/reading_room/whitepapers/securecode/threat-modeling-process-ensure-application-security_1646
[71] Gary McGraw. "Risk Management Framework (RMF)" Homeland security 21-09-2005
URL: https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/risk/250-BSI.html
[72] Justin Kerr. "How to: Install Linux free, with no formatting or repartitioning required." 28-07-2008
URL: http://www.maximumpc.com/article/howtos/howto_install_linux_risk_free_with_no_formatting_or_repartitioning_required
[73] "The Expat XML Parser"
URL: http://expat.sourceforge.net/
[74] "Microsoft Threat Analysis & Modeling v2.1.2" 29-03-2007
URL: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=14719
[75] "Remote Desktop and Server Administration.", RD connection manager
URL: http://rdpdesk.com/
[76] David A. Wheeler. "flawfinder."
URL: http://www.dwheeler.com/flawfinder/
[77] "RATS."
URL: https://www.fortify.com/ssa-elements/threat-intelligence/rats.html
[78] "Description of race conditions and deadlocks", Microsoft support, Article ID: 317723 December 6, 2006 - Revision: 2.3
URL: http://support.microsoft.com/kb/317723
[79] "The Mode Bits for Access Permission"
URL: http://www.gnu.org/software/libc/manual/html_node/Permission-Bits.html
[80] Sean Barnum, "STAT - Vulnerable to TOCTOU issues" April 17, 2007
URL: https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/coding/841-BSI.html
[81] "The OWASP Risk Rating Methodology" January 18, 2012
URL: https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology
[82] Azzam Mourad et al. "Security Hardening of Open Source Software"

URL: http://users.encs.concordia.ca/~debbabi/pdf/Security%20Hardening%20of%20Open%20Source%20Software.pdf
[83] Immonen et al. "Trustworthiness Evaluation and Testing of Open Source Components", 11-12 Oct. 2007.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4385514
[84] Anbalagan et al. "Towards a Unifying Approach in Understanding Security Problems", 16-19 Nov. 2009.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5362096
[85] Schryen et al. "Increasing Software Security through Open Source or Closed Source Development? Empirics Suggest that We have Asked the Wrong Question", 5-8 Jan. 2010.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5428450
[86] Aggarwal et al. "Integrating Static and Dynamic Analysis for Detecting Vulnerabilities", 17-21 Sept. 2006.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4020095
[87] Torri et al. "An evaluation of free/open source static analysis tools applied to embedded software", 28-31 March 2010.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5550368
[88] Peng et al. "A comparative study on software vulnerability static analysis techniques and tools", 17-19 Dec. 2010.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5689543
[89] Chatzieleftheriou et al. "Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities", 18-22 July 2011.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6032220
[90] MAGNUS ÅGREN "Static Code Analysis For Embedded Systems", August 2009.
URL: http://publications.lib.chalmers.se/records/fulltext/111920.pdf
[91] "Software Development Process"
URL: http://www.computerhope.com/jargon/s/softdeve.htm
[92] Deb Shinder. "Is open source really more secure?" 06, April 2005 Misc Network Security
URL: http://www.windowsecurity.com/articles/Open_Source_Secure.html