AntiVirus Program

Vulnerability Analysis Report

Viktor Listi

vili22@student.bth.se

Date Submitted: 28/03-2023

Abstract

This report aims to identify and assess the various potential security threats and vulnerabilities that could face the antivirus program that was made as a part of this project. The report will discuss risks and vulnerabilities to which solutions and potential fixes will be given. This is done throughout the report by first finding these vulnerabilities and after that discussing these in further detail for which in the end a solution to these will be reached. The report will only cover any potential vulnerabilities from the software side of the antivirus program and will not contain physical attacks from an outside attacker.

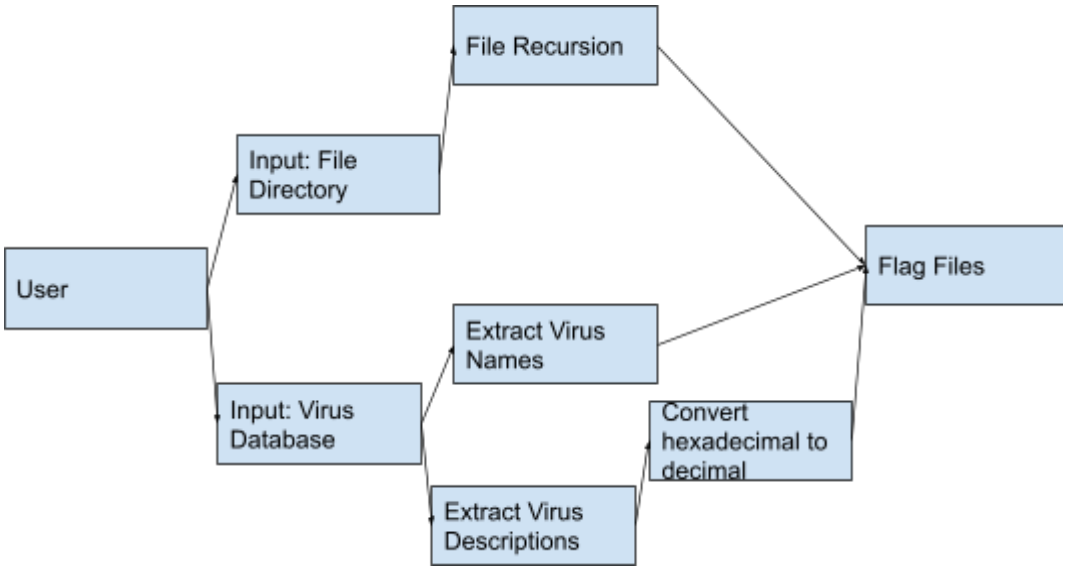
Introduction

The software that this report aims to assess vulnerabilities for is a antivirus program written in C++ that searches a directory hierarchy and the files inside for viruses based on an external database of known virus descriptions. The software does this by taking user input data in the form of a directory to search and a database from which it checks the files in the directory and compares the descriptions of the files in the directory to the descriptions of the known viruses in the database.

The program is built up of 5 functions excluding the main function, all of these have their own purpose, with the end goal of flagging the virus files. The first function uses the dirent.h library to recursively iterate through the directory hierarchy and save all the files contained in the hierarchy to a vector which is stored on the heap memory. After that the descriptions of the viruses contained within the virus database are stored to a vector and after that they are further split into two more vectors, one to store the names of the viruses and one to store the descriptions of the viruses in hexadecimal form. The description in hexadecimal is converted to decimal which gives ASCII values and when all have been converted the descriptions are then compared to the files, the comparison is only made for the first row of the file description. All these functions together aim to securely flag the virus files successfully, but bugs or vulnerabilities within the code can stop this from happening, therefore the goal of this report is to find these vulnerabilities.

Method

In order to identify the various vulnerabilities and threats that could face the antivirus software a variety of factors have been assessed in order to conclude the major weak points that the software contains. This is done by firstly identifying the data flow of the software by constructing a DFD (Data Flow Diagram) and examining it for potential attack points or potential user errors in the data flow, this does not detect any eventual bugs and those will need to be examined by other methods later on. The attack points in the DFD can be things such as where the data can be influenced in such a way as to cause a crash in the program or to give incorrect results.

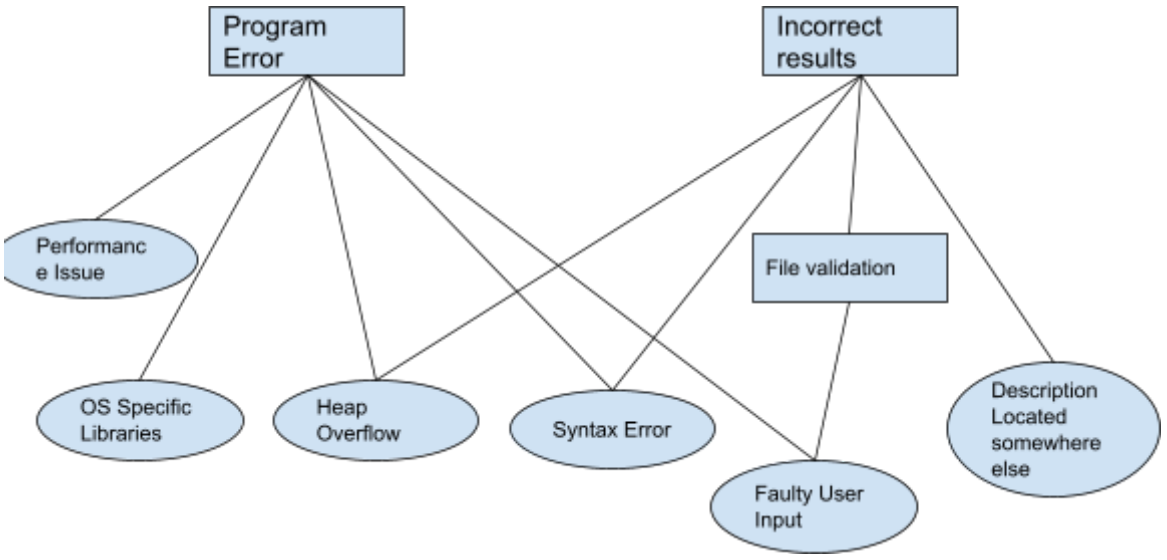


Picture 1, Data flow diagram for antivirus software.

By examining the DFD the conclusion can be drawn that the only part where user error can occur is the first step in the process where the user inputs data in the form of a file directory to search and a virus database to compare to. Errors here could be things such as administrative privilege errors, syntax errors in the database and incorrect file paths. From an attackers point of view the vulnerabilities are mostly the same as the previous points as unless the attacker changes the source code itself the only vulnerability is the input data itself. An attacker could manipulate the data in such a way as to cause a heap overflow or syntax errors which would lead to the software not being able to run or the software outputting incorrect results.

In order to find more vulnerabilities we can deploy Attack Trees to get a more detailed overview of the different access points and potential threats that could face the software and its integrity. With these methods any eventual bugs in the code that could lead to vulnerabilities can also be identified. By constructing an attack tree diagram a better view of

potential weak points, threats and vulnerabilities are set up to clearly see correlations between each other.



Picture 2, Attack Tree Diagram of software.

In the Attack Tree Diagram the potential things that could go wrong with the program are specified in the top, which are followed by any potential safeguards to protect these things from happening as seen in the current version of the software, these are displayed as rectangles. This is then followed by the potential threats that could lead to fatal errors in the output of the software which are displayed as circles. The Attack Tree displays the things that could go wrong with the software during runtime and the potential actors that could cause the error. This Attack Tree is developed by examining the weak points displayed in the DFD and also by manual review of the source code and testing the program with different scenarios. Any vulnerabilities during this testing have been applied to the Attack Tree. Using these methods a conclusion can be drawn as to which potential vulnerabilities can be found within the software.

Result and vulnerability analysis

By analyzing the potential vulnerabilities that have been concluded with the help of Attack Trees and the Data Flow Diagram we as programmers of the software can better understand the potential weak points of the software and can therefore better understand the risks associated with leaving these unhandled. If this software was to be deployed for use against actual viruses with different threat actors trying to bypass the software the vulnerabilities in the software could lead to catastrophic failures. In order to handle vulnerabilities extracted from the Attack Tree and the Data Flow Diagram in the best way possible the data can be

put into a table where we can compare the potential probabilities and consequences of each of the vulnerabilities. This will lead to a result where we can see which of the vulnerabilities are of the highest risk and will need to be fixed, and which of the vulnerabilities are of lower risk and can be left unhandled or be solved later as they are of less concern. Doing this we decrease the attack surface that a potential attacker can work on as there will be less weak points for them to exploit or for the unknowing user to make mistakes on.

Vulnerability	**Type (User/Bug)**	**Connection to requirement specification**	**Connection to proposed solution**	**Cost of solution**
Heap overflow	Bug	No	Yes	Slower performance of the software
OS specific libraries	Bug	Yes	Yes	None
Syntax error	User/Bug	No	Yes	None
Faulty user input	User	No	Yes	None
Virus description in other place	Bug	Yes	Yes	Slower performance of the software
Performance issue	Bug	No	No	Less security
Administrative privilege error	User	No	No	None

Table 1, Vulnerabilities displayed with type, connections and cost of solution.

Vulnerability	Probability (0-3)	Consequence (0-3)	Risk value (Probability x Consequence)
(4) Heap overflow	1	3	3
(1) OS specific libraries	3	2	6
(2) Syntax error	2	3	6
(5) Faulty user input	2	1	2
(3) Virus description in other place	2	3	6
(6) Performance issue	1	1	1
(6) Administrative privilege error	1	1	1

Table 2, Vulnerabilities displayed with probability, consequence and total risk value.

From these tables loosely based on the CVSS method conclusions can be drawn for each of the vulnerabilities and the risk they could cause towards the software's integrity. The ranking is based on the probability of the vulnerability being exploited by determining how many factors can cause it to happen, and the consequence is based on what would happen if the vulnerability is exploited based on what the consequence of the end result is. The program just crashing has a lower consequence value than for example the program giving wrong results as the wrong results can be hard or even impossible to detect without manually looking at each file in the flagging process. Looking at these on a case by case basis in order the rankings can be explained further.

1. The "OS specific libraries" vulnerability. The software uses an imported library that is not a part of the standard libraries in C++ to recursively iterate through the folder hierarchy, this library has to either be manually imported in order to use it in a Windows operating system but it is included by default in both Linux and MAC operating systems. Lacking this library would cause the software to crash causing the program not to run properly. The probability value of this vulnerability is high as most systems are Windows based causing the probability to be high. A suggested solution to this would be to change the source code to have two cases, if the dirent.h library

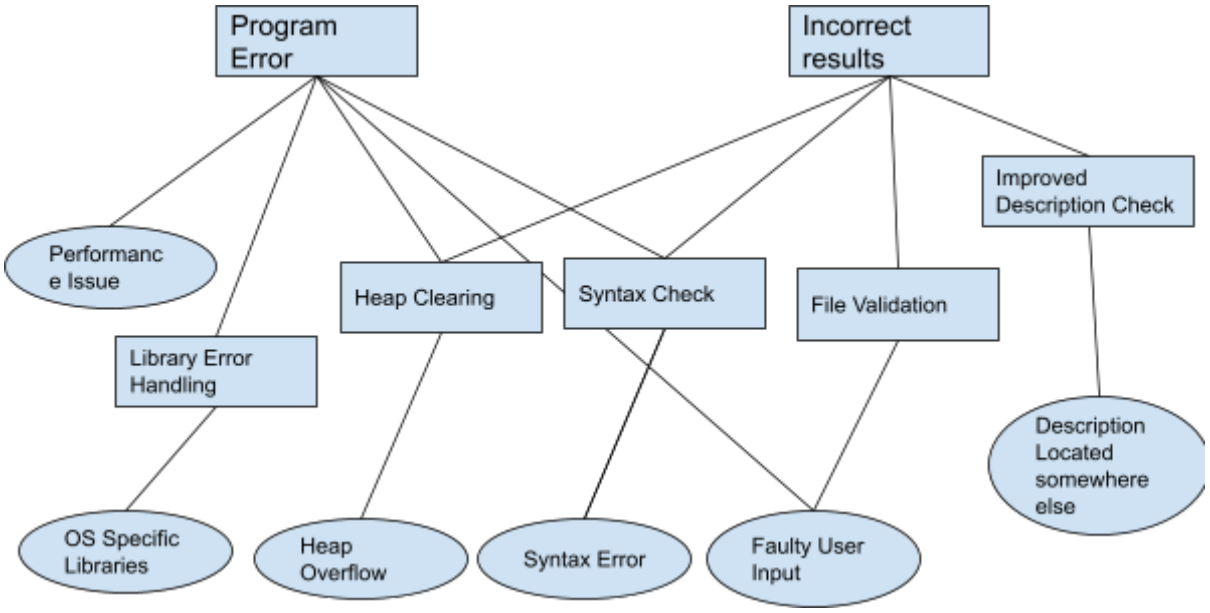
exists and is able to run then run that version but if it does not exist it should run with another method for iterating the folder hierarchies.

2. The “Syntax error” vulnerability on the table is based on syntax errors in the virus database. The chances for this happening is low if maintained properly but this being a vulnerability allows for outside attacks by influencing the database in such a way that it flags the incorrect files or that the software does not run at all. Both the probability is high and the consequence of it is high causing it to be a high priority vulnerability to fix. Currently the virus database contains a virus name and a virus description split by “=” to indicate the difference between them. The virus description is in hexadecimal form where two values indicate one decimal value. If the hexadecimal has syntax errors such as wrong symbols or too few values it can lead to errors. Many potential errors can occur in this part of the program, but this can be fixed with syntax control of the database by making sure there is a “=” on each row of the database and checking for errors in the description itself.
3. The “Virus description in other place” vulnerability is another high priority vulnerability as currently the software only compares the first row of the file description with the virus description, this opens an easy path for potential attackers to pad their viruses with filler in the beginning causing the software to not detect the known description in the file due to it not comparing data through the entire file. The consequence to this would be severe as this could mean that an attacker could pass a virus file without the software detecting it. A potential fix to this vulnerability could be that the software instead of just comparing the first row of the file description compares the known virus description to the entire file description.
4. The “Heap overflow” vulnerability is based on the heap memory being filled causing it to either overflow and write to memory already storing data or to crash the program itself. This can lead to both wrong data flagging and fatal errors leading to the program not running properly. The consequence for this is therefore ranked high as it can deeply impact the integrity of the program. The probability for this happening is however low as a lot of data has to be written to the heap in order for this to occur. A solution to this problem could be to wipe the heap memory after X amount of data has been written to it by continuously flagging the files and sending them to the log file. This could be done instead of first storing everything in the vectors stored on the heap memory and then sending everything through the flagging process at once.
5. The “Faulty user input” vulnerability is a mild one consequence wise as the software already has checks in place to see if valid paths for the directory hierarchy and virus database have been given. If these paths are not valid the software will warn the user and abort. The probabilities of this happening is however high as it is easy to mistype

when entering the input. There could be additional solutions to this in order to better handle the faulty user inputs but as it currently has a warning system in place another solution is not needed at the moment.

6. The last two vulnerabilities are of such low risk score that they can be combined into their own section, “Performance issues” and “Administrative privilege error”. Both the probability of these vulnerabilities happening and the consequences if they happen are low. Performance issues can be caused if the software has to work through a lot of data causing it to run slow but this does not affect the program but only the user experience of the program, fixing this would also lessen the security of the program as less code would have to be compiled in order to fix this, which would mean less security checks. The administrative privilege error can happen if the user is trying to access admin protected folders or files which the software is not allowed to do. There is no real solution for this as the user themselves would have to manually fix this problem.

With these improvements a new Attack Tree diagram can be constructed where you can see that there are far less paths where there could be errors as there are more safe guards.



Picture 3, Improved Attack Tree Diagram of software.

Summary

In summary, this report has identified a significant amount of security vulnerabilities in the antivirus software by the use of different methods such as Data Flow Diagrams and Attack Trees. Many of these vulnerabilities can potentially lead to severe breaches of integrity in the results of the software. These vulnerabilities originate from both the user and bugs within the software itself. The found vulnerabilities in the software have been ranked and been given suggested solutions to fix them. By fixing these vulnerabilities with the solutions suggested the program will be more secure, causing the results to be more reliable in the end even if challenged by someone who maliciously tries to impact the results of the software. With the results of this report you can see the importance of performing such vulnerability analyses as by doing this multiple flaws in the software have been discovered for which some of them would have severe consequences if exploited.

This analysis of the vulnerabilities does of course not find all the flaws in the software itself as there will always be things that you miss and new different approaches to breaking the program, but by finding these vulnerabilities the software will still be made safer overall.

References

[1] First.org, June 2019, Common Vulnerability Scoring System version 3.1: User Guide, 28/03-23, <https://www.first.org/cvss/user-guide>

[2] Ding, J. (2023/02/14). L3: Vulnerability and Threat Modelling. Department of Computer Science, Blekinge Institute of Science. https://bth.instructure.com/courses/4874/files/882063

[3] Ding, J. (2023/02/17). L4: Software Security. Department of Computer Science, Blekinge Institute of Science. https://bth.instructure.com/courses/4874/files/884498?wrap=1