universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Threat Intelligence based Impact Analysis for Software Vulnerabilities in Microsoft Environments“

verfasst von / submitted by

Bernhard Hirsch BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2019 / Vienna 2019

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Dr. Gerald Quirchmayr

Abstract

As an ever increasing number of software vulnerabilities are found and computer systems grow bigger and bigger, more and more administrators are in need of knowing which software installed within their computer network is vulnerable to attacks. In this master thesis, we will focus on finding a way to identify vulnerable software installed on Microsoft Windows machines.

We will start by investigating available solutions to identify software vulnerabilities. Based on CVE files, a comparison of vulnerabilities against installed software can be done without additional administrator rights. Such an approach is more secure, as no additional rights are needed to perform the analysis, and the framework can be adapted to network environments.

Based on the developed framework a prototype will be created to test it. The conclusion shows that such a solution is possible, that it can be deployed in Microsoft Windows environments, and that it can be optimised for further development.

Abstract

Da eine immer größer werdende Zahl von Softwareverwundbarkeiten gefunden wird und Computersysteme immer größer werden, müssen mehr und mehr Administratoren wissen, welche installierte Software verwundbar gegenüber Angriffen ist. In dieser Masterarbeit konzentrieren wir uns auf das Erkennen von Verwundbarkeiten in installierter Software auf Computern mit Microsoft Windows.

Wir beginnen damit, verfügbare Lösungen zur Identifikation von Softwareverwundbarkeiten zu untersuchen. Basierend auf CVE-Daten wird eine Analyse der installierten Software auf Verwundbarkeiten ohne zusätzliche administrative Rechte durchgeführt. Ein solcher Ansatz ist sicherer da keine weiteren Rechte zur Ausführung benötigt werden und das Framework an die Umgebung angepasst werden kann.

Basierend auf dem entwickelten Framework wird ein Prototyp erstellt um das Framework zu testen. Die Zusammenfassung zeigt, dass eine solche Lösung möglich ist, sie für Computer mit Microsoft Windows erstellt werden kann und dass dieses Framework weiter angepasst werden kann.

Contents

1 Introduction

This master thesis presents the research on "Threat Intelligence based Impact Analysis for Software Vulnerabilities in Microsoft Environments". It focuses on the construction of a framework able to analyse installed software in Microsoft environments for known security risks.

In this master thesis, we start by analysing related works and literature that exist in this field and have influenced the development of the purposed framework. This analysis will be categorise according to the following points:

1. Threat intelligence

   This section analyses scientific publications which identify methods on how to acquire information about threats.

2. Software vulnerability

   Scientific publications in this section analyse examples of vulnerabilities in software systems.

3. Handling of threats and vulnerabilities in Microsoft Environments

   followed by the the main aspects that have been influencing development, namely:

   (a) Identified challenges.
   (b) Solutions which the master thesis can build on.
   (c) Existing solutions and scientific publications which the core analysis model can build on.

   This section gives examples of basic methods defending against vulnerabilities and security functionality introduced in Microsoft Windows Vista.

The second part of the master thesis introduces a prototype of the framework development and implementation. The framework will be described as support for administrators to scan and detect software and vulnerabilities within the system under observation. The framework will do so by checking the software used in the system under observation, comparing the used software against lists of vulnerabilities, and informing the administrator if the software has been marked vulnerable in one of the vulnerability lists. The structure is as follows:

1. Motivation

   This chapter describes the motivation that influenced the aims of the framework and how the decisions came to be made.

2. Aim of the concept

   This section will describe the goals and non-goals of the concept.

3. Initial Structure

   With knowledge collected in the sections before an initial structure will be described and evolved.

4. Framework

   The structure in its final state it will be described as a framework using UML diagrams.

5. Implementation

   This section focuses on the prerequisites and initial assumptions for the practical implementation of such a prototype.

Based on the initial literature survey, the thesis will introduce a framework to identify software vulnerabilities. This framework targets scalability and simplicity to increase the robustness of the framework and to allow administrators to adopt the framework depending on their execution environment. The framework aims to reduce the effort of administrators when comparing software vulnerabilities with the software used. The later prototype presents the framework's benefit as well as the opportunities for optimisation.

2 Threat intelligence for vulnerabilities in software solutions

This section begins the literature review by covering the problems of identifying threats within a system under observation. The focus lies on a few examples of online algorithms able to identify new threats during runtime. While two of the named examples discussed in this section try to detect threats by analysis methods, another approach introduces a message service from the government of the United States of America as a contra point. It has been done this way to explore the pros and cons of such methods and to develop a stable and reliable software solution.

2.1 Detecting Insider Threats by Monitoring System Call Activity

Nguyen et al. describe an analysis method to detect malicious behaviour in systems under observation[16]. This analysis aims at the abuse performed by insiders to attack the infrastructure with authorising permissions. While this method introduces effective attack measures, an implementation of this method in addition to those mentioned later would be out of scope.

2.2 Privacy Impact Assessment for Automated Indicator Sharing(AIS)

AIS describes the source of the TAXII and STIX-Messages as cooperation between individual companies and the government of the United States of America. These messages are created by private companies or governmental institutions when a threat is found. The messages are sent to the DHS National Cybersecurity and Communications Integration Centre (NCCIC, national related to the United States of America), where they are computed, adopted, and sent to all participants of the AIS program. These private entities can than adapt their IT infrastructure to protect it from the detected threat.

During the construction of the framework, this messaging system has proven inappropriate. This is because vulnerabilities have been shared with this messaging system by relating to CVE numbers. Since only the use of a CVE parser would give sufficient results, it would not make sense to implement a TAXII/STIX parser as well and, therefore, the development of the TAXII/STIX parser was abandoned.

2.3 An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities

Zhang et al. describe an alternative solution to find software vulnerabilities by using the National Vulnerability Database[24]. In this paper an estimate of the ability to find the next vulnerability based on the National Vulnerability Database and the given system under evaluation is calculated. In the conclusion section, the paper states that such an approach was not effective. In this master thesis such a priori approaches will not be used as the precision is too low.

3 Types and sources of software vulnerabilities

3.1 Introduction and structuring

The definition of software vulnerabilities is not standardised. For example, [29] (Part I, Chapter 2) states "A vulnerability is a bug in software that enables an attacker to bypass security." Based on this definition, the following structure is used to analyse possible vulnerabilities and their relation to the development of the framework:

- Network and inter-process communication
  - Communication initialisation requests and intercepted communication initialisation requests
  - Attacks on existing communication lines
- Hardware-related software vulnerabilities
  - Vulnerabilities during process execution
  - RAM readout and overwrite
  - Data change
- General software related vulnerabilities
- User errors

3.2 Network and inter-process communication

3.2.1 Communication initialisation requests and intercepted communication initialisation requests

Communication between two processes has to be initialised. This is mainly done by a waiting process and a requesting process which attempts to send a communication initialisation request. With TCP and UDP, these actions are described using standards ([20] and [19]). On the application layer this behaviour has not been standardised, except socket programming on the layer of operating systems [12](Chapter 2).

At this point, malicious applications may attempt to build up a communication with the vulnerable software on the application layer. This is done either by

- sending a communication initialisation request
- answering on an incoming or intercepted communication initialisation request

In both cases, the software being attacked has to detect whether the communication partner is malicious or not. If this detection cannot be performed, vulnerable information may be sent outside or destructive orders may be received and executed.

3.2.2 Attacks on existing communication lines

An attack on already existing communication lines is done by changing or observing communication packages. This is done by software that is placed in areas where the packages have to pass (network interfaces or routers or similar). A complete description of this man-in-the-middle-attack can be found at Kurose et al.[12](Chapter 5).

3.3 Hardware-related software vulnerabilities

A hardware-related software vulnerability is defined as vulnerabilities based on the driver software for hardware components.

3.3.1 Vulnerabilities during process execution

This category focuses on vulnerabilities in hardware components that enables the attacker to read or change data during its calculation. Two popular vulnerabilities are Spectre [13] and Meltdown [15]. These vulnerabilities allow the readout of secured memory areas by executing malicious code.

3.3.2 RAM readout and overwrite

Operating systems are responsible for the RAM management of the executed programs [27] (Chapter 3). As some programs store important information in the RAM like file destinations, usernames or passwords, a readout of their RAM can be seen as a software vulnerability in the operating software (because the operating system guarantees a separate RAM section for each program to work with).

3.3.3 Data change

Permanently stored information or programs should only be changed by authorise users. Vulnerabilities may enable unauthorise users to change such data. Modern operating systems take responsibility for these changes by implementing, for example, access control systems[27] (Chapter 9).

3.4 General software related vulnerabilities

This section focuses on errors which can occur in software. Tan et al. describe three types of bugs: “Memory bugs are bugs that are caused by improper handling of memory objects. “Concurrency bugs are synchronisaion problems among the concurrent tasks in concurrent programs” (Lu 2008), including data races and deadlocks. Semantic bugs are inconsistencies with the requirements or the programmers’ intention that are not memory bugs or concurrency bugs.” [26]

Because of the potential complexity of such errors, they are hard to find. Their appearance may be influenced by a variety of variables so that reproduction may not always be possible.

Beside the difficulty in finding these bugs, Tan et al. also explained the rising amount of bugs compared to the complexity of code[26]. This statement may increase the importance of declaring bigger software projects as more vulnerable.

3.5 User errors

While the previously concerned sections tackle the question where (in the software's logic) a vulnerability is situated, this section is concerned about the responsibility a software application has concerning its user. This point mainly focuses on the responsibility of operating systems, but may also consider architectural decisions of third-party programs.

Considering the user as a threat, there are two categories of possible attack scenarios and their reasons:

- A user with bad intentions tries to insert, change or delete data from the system by invoking unauthorise actions or tries to execute malicious software.
  - Insider attack:
    An employee from a company starts to attack it. Such attacks are a threat to companies as the attacking insider knows what he can achieve and how to harm the company.
  - Stolen user credentials:
    To computer systems, stolen user credentials may appear as if an authorise user is giving (malicious) instructions.
- A user with good intentions tries to insert, change or delete data from the system by unauthorise actions or tries to execute malicious software.
  - User Error
    This category considers misbehaviour unintentionally performed by users. The question for this analysis is: To which degree is the software considered responsible if a user can make wrong decisions? As for the software, the software has to guarantee the clarity of the user's actions. Therefore, software can be held vulnerable if the user is not able to identify the consequences of its actions.
  - Attack on the system
    It may turn out that the attacker changes the look of common system software or even replace it with a malicious version. This may result in faked log-in-screens or suppressed security warnings. These symptoms are described in [27](Chapter 9). This can be seen as a software vulnerability as the software is supposed to prevent such actions and save the user from these attacks.

4 Handling of threats and vulnerabilities in Microsoft Environments

This section gives an insight into vulnerability handling in Microsoft Environments. This section lacks information as the main features of vulnerability handling in Microsoft Environments are proprietary.

4.1 Vulnerability detection based on aggregated primitives

This patent describes an architecture for an intrusion detection system [1]. While most parts and functions are under this patent and, therefore, may not be used in a master thesis, the structure may be found useful for the resulting program.

4.2 Microsoft's new window on security

This article describes the security features used in Microsoft systems introduced with Microsoft Windows Vista[2]. Although this article is written for an unsupported version of Microsoft Windows, it details the problems, their solutions introduced by Microsoft, and the problems that still remained.

4.3 Modern operating systems (Chapter 9 and 11)

The chapters of this book cover basic security aspects of Microsoft Windows Vista[27]. While these aspects mainly contain information about simple characteristics of the security of Windows Vista, some aspects give deeper insight into the techniques.

5 Identified challenges

This section tackles the challenges that arise from the analysis made in Section 4. The challenges are ordered by their expected size of impact on the system under observation.

5.1 Identifying vulnerabilities

The previous sections gave an insight into which vulnerabilities are currently known. The challenge is to identify new or unknown vulnerabilities. There are different approaches to deal with this situation:

- Program analysis (as described in [26])
- Preventing unproven programs from execution or reducing their execution rights [27]
- Updating programs to the latest version
- Removing unproven programs from the system if the security cannot be guaranteed.

While these tactics are a good first line of defence, software vulnerabilities still exist and may even circumvent these defence mechanisms (like at Heartbleed [6]). So the question is how to predict vulnerabilities.

5.2 User related security measures

As mentioned above it is assumed that a user error is a software vulnerability if the malicious behaviour of the user could have been prevented by the design decisions of the programmer. Therefore, user related security measures are added to this master thesis.

User related security measures contain, among others, the fact that a user should be aware of his actions. This can be achieved by education but also by, for example, security policies or playing through scenarios. Also, design decisions are related to this category as those may make the user better aware of the consequences of his actions. Other measures like user rights management or multilevel-trust [27] are a stabilisation but can eliminate neither errors in their settings nor misbehaviour of the users.

Therefore, user related security measures will be stated as a challenge of design decisions and social commitments.

5.3 Network and process identification and communication

How can one ensure that the process responding is the process it pretends to be? While this question is simple for one-time communication, a communication line that is established over and over again may have been observed. How then can it be guaranteed that the other process really is the one it says? And even if the identity can be verified, how then can it be assured that the received message has not been changed during transportation.

The challenge would be to set up an appropriate security system to identify and encrypt the messages sent over the network.

5.4 Hardware security

The importance of hardware for software vulnerabilities lies in the responsibility the software has over unchangeable parts of the computer. For example, Spectre and Meltdown are considered harmful for Intel processors[13][15].

Such a scenario has to be considered for all parts of the hardware. As stated above, the operating systems are responsible for managing the hardware and contain a set of defence measures. Nevertheless, holistic hardware related management overall computer components is only established on the level of the operating system where the drivers are managed [27](Chapter 1).

For RAM vulnerabilities the challenge would be to prevent the readout of unused RAM pages. Both the readout of previously used pages and a cold-boot-attack are a challenge in this topic.

For data changes the operating system has to keep track of all rights on information (including both data and programs). In addition to this, in some cases, it may be preferable to also log the change and the changer of information.

Besides the RAM- and data change Use Cases, the challenge lies in the detection of hardware related vulnerabilities and the development of counter strategies.

5.5 User identification

While user related security measures concerns the behaviour of users inside and outside their companies, user identification asks how a computer identifies a user. This can be done by either username-password-identification or by special tokens (such as chip cards)[27] (Chapter 9). The username-password-identification systems mentioned have proven to be easy to use and can be applied in a vast field of identification problems. However, such systems are vulnerable to identity fraud.

Beside the question of authentication, the reason for it should be kept in mind as well because of rights the user has within a system. Therefore, rights management can be seen as a challenge as well.

This said, the challenge would be to identify appropriated models for checking the user identity. Additionally, the rights management within the software should be set so that there is a balance between the possibilities a user has without identification and the possibilities he has when logged in. As for identity fraud handling, the challenge would be the identification of an attack and the design issues to guarantee that misuse is not likely to happen.

5.6 Faulty program libraries

While creating a program, libraries can help to reduce the complexity of the code and reuse well-established mechanisms. While those may introduce a benefit to the development of the program, there is a risk of taking over vulnerabilities (like Heartbleed [6]).

The detection of vulnerabilities in shared libraries may lead to a more secure environment. The challenge is how to detect third-party library vulnerabilities while still working economically in sense of time, money, and resources.

5.7 Complexity

In the previous sections, a correlation between code complexity and the number of bugs per program has been stated [26]. Therefore, code should be kept simple to reduce the possibility of vulnerabilities. On the other hand, computers get more and more complex by adding new features without removing old ones.

An aim is the creation of simple code (modularise code, small code modules, etc.) and the use of outside libraries (preferable open source projects) to enable the desired functionality.

5.8 Future needs and future vulnerabilities

In software development, backward compatibility is important to keep older software running [2]. In standard software development, this mainly applies to network or process communication and backward compatibility of data but also refers to the possibility to extend the software in future releases.

On the other hand, this method may introduce new vulnerabilities. Legacy code has to be tested for old vulnerabilities and updated if needed, new code increases the complexity while introducing additional vulnerabilities. Under these circumstances the cooperation between legacy code and new code may influence the program's stability as well.

The challenge would be on the one hand to keep one's own framework and its resulting implementations simple and, on the other hand, to enable the framework to analyse legacy software.

5.9 Secure programming

The challenge in this category would be the analysis of known security measures which can be used during programming and describing those in the resulting framework.

6 Solutions which the master thesis is based on

Based on the identified challenges, a couple of solutions can be acquired from the papers and works related in the previous section to secure the result of the master thesis. This section covers solutions which may be adapted to the needs of the master thesis or kept out completely if they do not meet the purpose. Solutions identified in this master thesis before but not used will be declared as such in this section.

The structure will be taken from the Section 5 to increase the readability.

6.1 Identifying vulnerabilities

This section contains a few points which may increase the security of the written program. Those are:

- Bug testing and program analysis.
- Restriction of program execution with user rights management. As this is done by the operating system, it will not be considered in the following analysis.

6.2 User related security measures

Security measures concerning users can be categorise by design decisions and teaching strategies. As it turned out in the later development of the framework that the framework does not include an interactive user interface, this chapter is reduced to user rights management and consistency checks within the framework. This reduction of interaction between the framework and the user reduces the complexity of the program and introduces more reliable behaviour.

Teaching strategies include every kind of knowledge transfer from the development of the framework to the end user. This may contain manuals given or presentations taught to the target group. The implementation of training and guiding strategies is advisable but not used during the implementation of the prototype as it is out of the scope of this master thesis.

6.3 Network and process identification and communication

Considering the network communication and the identification between the processes, the most secure solution would be a certification authority. It would

guarantee that, besides changing IP addresses, the encrypted packages are from a known source. Because of the external work needed to set up such a certification authority within a system, it should be avoided for this master thesis.

Asynchronous encryption can be enabled with such a technology, although asynchronous encryption may be weakened if one of the computers is hacked and the public or private key or both are stolen. As this method cannot be considered secure on its own, it is considered to be used in a constellation with other security mechanisms described later. At this point, it should be stated that encrypted communication has not been considered for the prototype of the framework and it is not the only security mechanism to guarantee the executability and reliability of the framework.

6.4 Hardware securing

As for the hardware a good solution has not been found. The updating of the hardware drivers is essential but should be managed by either the hardware drivers, the operating systems or the administrator. The same holds true for RAM vulnerabilities.

Changes in program code can be secured as well. The code can be given to a hash function and the values are checked against a stored version of the hashed code from a previous check. To secure this mechanism the hash values of a program have to be checked by a different program. Later in this master thesis, a control program is considered to fulfil this task.

6.5 User identification

User identification could be made more sophisticated with, for example, identity cards or biometrical values (described in [27](Chapter 9)). As this would not fit into the frame of the master thesis, a simple username and password protection combined with execution rights of the operating system will be used. As for the protection of the user credentials, those values will be stored as a combination of both name and password and afterwards hashed with a mechanism like SHA512.

The following counter measures for the misuse of user credentials could be implemented:

- Insider attacks

  As for the problem of insider attacks, it is hardly affordable to implement a sophisticated insider attack detection system like mentioned by Nguyen et al.[16]. Therefore, it will be assumed that the user is conscious of his actions. As stated before the user credentials will be stored in a hashed version. Therefore, it is assumed that the mechanisms introduced by this master thesis can not prevent insider attacks.

- Misuse

  It is still possible that the user may misuse the system because of errors. This should be avoided by the design of the user interface and teaching. Furthermore, the user will be granted minimal decision freedom to guarantee the security of the program.

As for third party software, a detection cannot be given as it must be assumed that the operating user (or one that pretends to be the operating user) is aware of his actions. As the log-in and user management system is part of the operating system [27](Chapter 1) the correctness of the operating system and these functionalities is assumed.

As described before and realised later in this master thesis user interaction is reduced to a minimum. Therefore, GUI considerations will not be considered further.

6.6 Faulty program libraries

As for the master thesis, testing program libraries on vulnerabilities would be out of scope. Therefore, the detection of library vulnerabilities will be reduced to well-known issue statements on public web pages during the development process.

6.7 Complexity

To reduce complexity open source third party libraries will be used. The functionality will be reduced to a minimum to achieve a less complex program. Additionally, complexity can be reduced when the framework is built via a modular methodology.

6.8 Future needs and future vulnerabilities

This category represents methods to keep the framework and the resulting implementation adaptable to future needs. This adaptability is assured with the use of a modularise approach and a reduction in complexity. Also, it is advisable to use external libraries and adapt these to implement state-of-the-art features while removing known vulnerabilities.

6.9 Secure programming

To enable secure programming, security will be part of the architectural process. This includes analysis of known threats to the program libraries and the programming language, usage of programming patterns, and debugging programs. During the prototype's creation analysis tools like debuggers and static code analysis tools for securing the programming will be used.

7 Existing solutions and scientific publications which the core analysis model can build on

As described before the way to retrieve vulnerability information is essential. This master thesis will focus on second-hand information rather than on an experimental information retrieval system as described in [24]. Also, complex systems like [16] will not be considered because such an analysis may need additional rights to scan network packages throughout the network (It would need those rights also when watching over the local computer this way). As described later in this master thesis the main source of software vulnerability

will not be the AIS information exchange but the CVE standard because the TAXII message does not contain specific information about the vulnerability as the CVE identification does.

Based on the inherent security measures of the operating system, this master thesis will use information from [27] to build up the program with existing security mechanisms like user rights management to secure the execution and configuration of the framework.

Since the framework resulting from the master thesis is intended to observe computer systems, remote applications have to be taken into account as well. Therefore, security measures like those described in [14] will be considered to secure the identification and the secure transportation of packages over the network.

Beside existing scientific papers, a wide field on software evolved around the problem of software vulnerabilities. Many solutions, like Anti-Virus software or version control software, are proprietary. Open-source solutions, instead, mostly lack the quality in finding software vulnerabilities. Because of that, this paper will present and analyse OpenVAS as an open-source solution, fueled by additional, proprietary services.

OpenVAS (see [8]) is a server-based software vulnerability detection solution created in an environment set up and maintained by Greenbone Networks (see [7]). Its detection mechanism is based on the comparison of information acquired via the network interfaces of the computer being analysed with lists of known software vulnerabilities.

OpenVAS is shipped with its own operating system which can be installed on machines with it as the only operating system. This solution prevents attacks from other operating systems running on the same computer while providing flexibility to run this system in virtual environments.

OpenVAS receives its information about vulnerabilities from internet servers maintained by Greenbone Networks[8]. The benefits of this solution are its ease of implementation with an internal updater daemon and the flexibility that information can be kept up-to-date depending on the systems needs.

This software checks the vulnerability of a computer by checking version numbers or behaviour or both of software running on a target machine over the network. This method allows a stand-alone solution with regular security checks on the one hand while being able to observe every (assigned) machine on the same network (even over the internet) on the other hand.

OpenVAS is a good example of a secure and focused runtime environment: Its hardened operating system is optimised for the software running on it. It delivers a stable and easy to install solution ready to operate with minimal configuration means.

OpenVAS is hardly able to detect attacks on software running without a network interface or network connection. This is because it only scans software having open network ports.

Another method mentioned in this master thesis describes checking the integrity of software. Such methods are used to restrict the usage of manipulated software or to guarantee the integrity of running programs like computer games (e.g. Denuvo at *https* : *//denuvo.com* [23]). Such methods are used but, as this information is valuable, especially in the gaming industry, comparable software and structural insights into such software has not been found.

8 Motivation and aim of the framework

Security in Information Technology is a complex topic. Its aims sound simple: Guarantee that users can utilise a computer as well as possible. On the other hand, its solutions do not: Computers should be stored safely, protected from fire and water, and should be tested against data corruption, among others.

Concerning digital attacks, many more consequences have to be taken into account: user management, attack vectors or communication channels for example. Additionally, sophisticated software solutions like Microsoft Windows tend to show complex and unpredictable behaviour. Some of this behaviour may introduce additional attack vectors, require rights on sensitive data or need to be trusted.

Based on this and considering the scope of this master thesis, a simple and adaptable solution to increase security would be preferred. This master thesis aims to produce a minimalistic software framework capable of increasing security by reducing known attack vectors. In the following sections, such a solution will be presented which reports vulnerable software to the administrator.

Because of the complex interactions between different parts of the computer network, it is not advisable to create a framework that repairs any vulnerabilities found on its own. The reasons are:

1. The framework would need the ability to change software running in the network. This can lead to data corruption and requires access to higher levels of the operating system.

2. If the framework informs the administrator instead, it could perform the repair mechanism in an organise way. It can find out what is needed to perform the repiars, which software interacts with each other, how the data can be saved from corruption, and how the process needs to be restarted.

Therefore, the framework should inform the administrator when a vulnerability has been found instead of repairing it.

Based on the introduction above, the following sections describe the framework in closer detail. This section will start by analysing the possible aims and not-aims for this framework to target. Later the framework will be described closer based on the aims and the literature analysis.

As stated before, the finished framework of the master thesis should support the administrator by detecting vulnerabilities and informing him. The term ‘detection’ will be reduced to the comparison of a list of vulnerable software against the currently used software because investigative vulnerability detection may lead to imprecise results (examples given: [25] and [22]).

The framework should inform the administrator automatically when a vulnerability has been found in the system. Therefore, the framework should be able to acquire vulnerability information, preferably from different sources because:

1. different vulnerability sources focusing on different categories are available,

2. there may be a vulnerability database set up in the execution environment.

With respect to the previously stated adaptability to the vulnerability sources, the framework in total should remain adaptable to allow programmers

to tailor the framework's development depending on the environment. It is not a goal of this framework to provide solutions for specific vulnerability information sources, databases or environments because of the ever-changing software environment.

When information of any kind is received by the framework, the question about the storage of the information may arise. As this information is relevant to security, they should be stored encrypted, should be transferred encrypted, and should not include personal or critical data if possible. The framework will, therefore, consider a database storage system to guarantee a secure connection and encrypted storage. This ensures that encryption mechanisms and storage strategies are used that have been tested in other critical systems and are hardened against attacks.

Furthermore this master thesis focuses on an implementation in Microsoft environments. While this may be appropriate for the current implementation, the framework will be designed to be adapted to different kinds of operating systems to allow their use in other systems.

The resulting framework works on information relevant to security. Its aim is, therefore, to describe processes in which such information is computed in a compliant, reliable, and transparent way.

9 Framework concept development

This chapter describes the considerations used to evolve the framework. This section will go top-down to tackle architectural considerations, possible attack scenarios on the framework, and efficiency considerations to build a stable and robust solution. This structure targets on fulfilling the aims and implementing useful patterns that have been mentioned in the sections before.

9.1 General considerations

Currently, two different approaches are used to automatically detect software vulnerabilities: a host-based approach and a network-based approach. For this framework a host-based approach has been chosen. The framework (or parts of it) has to be installed on the analysed machine. This is done because network-based vulnerability detection (like OpenVAS, see section 7) is able to detect vulnerabilities of software connected to the network, but is not able to detect if programs running without network interface could be attacked.

Given the host-based approach, the following problems arise:

1. As every instance of the framework's implementation has to have a copy of a vulnerability list (at least in its working memory), the network traffic may increase as all these vulnerability lists (at least the latest) have to be shared between the instances (if they are not received directly from the vulnerability source for every instance).

2. Assuming a vulnerability is found, such a vulnerability may be found on several computers. To prevent spamming the administrator, the framework has to be orchestrated in a way that as few messages as possible are sent.

3. The framework needs to know which message it has sent to the administrator. Considering this, a storage system may be useful to record those messages. Such consideration leads to the question of whether this storage system can be used to store software vulnerabilities.

4. The framework may be altered or attacked during its execution. It, therefore, has to be hardened against the most likely attacks, has to have detection mechanisms when an attack has occurred (against itself), and has to have a reaction mechanism when an attack has been detected. This section will mainly be covered in the subsection 9.2.

The first point depends on the size of the network built up by this framework (the number of participating machines), the internet service provider, and the available bandwidth for this communication within the network, which in turn depends on the global network bandwidth, the existing traffic, and the reserved bandwidth for other applications like video chat or E-Mail etc.

The second point can either be solved by a centralised server instance keeping records about the messages to the administrator or orchestration of client framework implementation. A centralised server instance is preferable as commonly used orchestration methods like token organisations or election mechanisms are vulnerable to attacks and need to know the other client software's location [28].

The third point needs to implement a storage system that can, as described, be implemented on the client site either with a database software or by writing a proprietary information system. Since proprietary systems could introduce errors, using such solutions should be avoided.

Given the solutions described above the structure can be described as:

1. A centralised server instance which receives updates about vulnerabilities, propagates this information to the client instances, and sends a message to the administrator with the relevant vulnerability information if a client instance requests it.

2. Client instances on host machines which observe the installed software compare it to the received vulnerabilities and send a message request to the server if a program is considered vulnerable.

The next question arising concerns the message flow to the administrator: How and when is the administrator informed about vulnerabilities? The administrator can be informed by push or pull messages. Since a detected vulnerability can lead to security issues, push messages are preferred. This has to be combined with a secure communication channel to the administrator. Such a solution has to be created with respect to the execution environment. Because of the extensive availability in runtime environments, messenger services or short text messages, E-Mails or automated phone calls will be mentioned here. Because the execution environment needs to be considered, a general answer cannot be given here.

9.2 Attack Scenarios

This section considers possible attack scenarios on the operation of the initial structure and how these may influence the structure of the framework. The

structure will be evolved around those attack scenarios to improve the stability. This section will start with attack scenarios aimed at the client software on host machines and will go up to the connections, the servers of the framework, and their connections to the internet. If other methods are added to the structure during this process, they will be analysed at the end of this subsection.

9.2.1 Client-side attacks

The first attack scenario is aimed at the operating system and the connection between the client instance and the operating system. Such an attack can neither be detected nor avoided as the framework is not intended to observe the installed resources on the machine and is not able to force the operating system to provide a secured channel for inter-process communication. This kind of attack is able to falsify computed information of the framework and can, therefore, make it impossible to detect vulnerabilities on this machine when run successfully. Because such an attack cannot be detected this attack scenario will be ignored.

An attacker can attack the framework's client instance. This can be done by different approaches like code injection or RAM manipulation. Good prevention against such attacks is only possible on the level of the operating system (e.g. user rights), therefore, those settings should be set properly. Such an attack is able to insert malicious data or manipulate or delete existing data processed by this framework. This attack can lead to unrecognised software vulnerabilities as the information about these vulnerabilities is not sent to the server (and from there on to the administrator). As the client instance is running on multiple machines this risk is reduced when the software is installed on several machines at the same time. Additionally, the attacker may receive the propagated information about vulnerabilities from the server instance and may deduce the known and unknown vulnerabilities in the system under observation.

Detecting such attacks turns out to be difficult as the client has to be checked from a program, not affected by this attack. This can be done by either:

1. testing the client's behaviour by sending test questions,
2. observing the client's behaviour,
3. testing the client's source code by a second software.

The first point is not appropriate as checks of this kind can be circumvented as infected code may answer with a repetition of the older program's answer. As the software has to have been reverse engineered for such attacks, such detection mechanisms cannot guarantee that the received information is coming from unaffected software.

The second point includes statistical information. It can be achieved by using a learning algorithm which learns the dependence of input data of one client against another client. While this method can react to changes in the behaviour of clients, it may introduce false-true-errors, especially if a client changes its behaviour (if it is swapped between users or if the user is on vacation, etc.). Because of unknown implementations in other projects, this method is not used in this framework.

The third point would include controlling software running and checking the client software. Compared to the first point, such a program would be able to

report changes to the server independent of the client program. On the other hand, its existence is easily found as the software has to run in the background, is visible to the operating system, and all processes calling the task manager (or comparable programs).

Considering the stated solutions the third point will be implemented as it is the most stable solution. Additionally, this approach can trigger a message to the server (and to the administrator) that a change in the software has been detected.

Similar results to code injection would be the change or deletion of configuration data. The manipulation of this information could deactivate the framework because:

1. The address of the server entity may be manipulated or deleted and the server may, therefore, stop receiving information.

2. The database credentials may be deleted or manipulated and the machine's software could not be compared to the stored vulnerabilities and the attacker could know the vulnerabilities known to the framework.

3. The administrator's contact address is deleted or changed and the administrator cannot be informed about the vulnerability.

4. The information in the database is changed or deleted or the database files are deleted. This would lead to a scenario similar to the deletion of the database credentials.

This attack can be prevented by using access management mechanisms like user rights or similar. This method is only applicable to the level of the operating system and is, therefore, not considered. Instead, copies from the configuration files can be made and can be compared to the actual stored values. This is applicable for small files; the database will be secured by using hash values of its content and comparing the hash values to the actual content of the database. This method will be described later in more detail. If such an attack has been identified, the administrator should be informed when possible.

Additionally, attacks on the database can be hindered by storing only hashed values: The information will be obscured by using hash mechanisms and can, therefore, not be re-engineered. The comparison then will be done by hashing the name and the version of the software to check and compare the resulting values with the stored values. This method will be described later in detail.

In addition to the database approach, another attacking scenario would be to copy, manipulate or delete the log-in credentials used by the framework. A copy can harm the system as the attacker can get access to the infrastructure. This behaviour could be prohibited if the framework acted on the level of the operating system and would observe every read/write access to these credentials. As this is not in the scope of this thesis (because of the complexity and the dependence of the framework on the architecture of the operating system) it will not be considered further. Other approaches like user rights management will be considered as alternatives in this case. Such an attack can only be detected if the attacker and the attacked framework element are active on the database or the server at the same time. In this case, the administrator has to be informed.

If configuration information is manipulated or destroyed, the client would not be able to call the database or the administrator. Such an attack scenario cannot be prohibited on the base of the framework (as mentioned before), but it is recognise immediately as the information is used on a regular basis. The best countermeasures would be the storage of several copies of this information on different locations. When one piece of this information is manipulated, its counterpart can prove the change. When the copies are changed as well it may be hard to find an appropriated way to prove or restore from such an attack. At this point, no sufficient solution has been found that is able to both restore the configuration files and detect malicious behaviour when the attacker knows the framework architecture.

The next attacking scenario would be a DoS-attack: The process may be deactivated by killing the task or by blocking the output connections. Such an attack can render the framework element useless as the communication of vulnerabilities found can be blocked completely. If the framework element has been deactivated this may have the following reasons:

1. A malicious software ordered the operating system to shut down the process.
2. The computer is about to shut down.
3. The software did not start during the startup process.

A detection mechanism for such situations has to identify if the software stopped the execution because of case two or not. If this is the case it should stay silent and wait until the computer has shut down. Otherwise, the mechanism should inform the administrator. This can be effected by the control program mentioned before: The program checks the task list of all running tasks and informs the administrator if the observed software has not been launched for a given amount of time. This time span should be big enough for the computer to shut down but too small to allow attackers to harm the system or the framework.

If the software has been deactivated because of firewall settings or attacks on the network infrastructure (man-in-the-middle-attacks as well as damaged infrastructure) a controlling program on the network could detect such attacks by checking the last message timestamp of this client. While it is clear that such attacks cannot be prevented but detected and committed to the administrator, the detection mode is unclear. Whether the client is absent (for example the user is on vacation and the computer is not turned on), or if the client has been turned off regular for nomral reasons (for example the replacement of the machine) or if it is an attack. While such behaviour could be identified, a high false-positive rate has to be accepted. Therefore, this feature will not be implemented.

9.2.2 Network attacks

This section covers the connection over the network from the client to the server. This connection can be attacked by a man-in-the-middle attack. Such an attacker gains access to the messages and knows, therefore, the content of each message and may alter or block the corresponding message. Such an attack can, when used on the server software, deactivate the server, and stop the information flow to the administrator by blocking the messages (as described before).

As a solution regarding the message integrity and source, asynchronous encryption can be used. While this may be risky as the key has to be stored securely (user rights), it enables the parties to communicate and identify each other. A solution for detecting blocked communication lines on the level of the framework elements has not been found.

9.2.3 Server-side attacks

The attack scenarios referring to the server side of the framework do not differ from those on the client side but may be more important as the server represents the communication to the administrator and supplies the clients with information.

Depending on the exposition of the server instance, multiple server systems could be set up to harden the structure. While this method may interfere with the idea of sending only one message to the administrator per issue (as mentioned above), it secures the system as an attacker would have to attack several servers at the same time to guarantee that the framework is down. Additionally, this approach would mirror the organisational structure of the system under observation.

Attacks targeting the server instance can bring down the framework as either:

1. the administrator will stop receiving vulnerability information or
2. the client side will stop receiving vulnerability information and will not recognise newly found attack scenarios.

As described before, behaviour control mechanisms will not be introduced because of the lack of information on such systems.

As also described before, the server instance will be observed by a controller software as well as it has been described for the client instance. This control software will compare the server's software integrity and the configuration file's integrity. If an error is detected, instead of messaging another server the administrator will be informed directly.

As well as the connections between the client and the server the connection between the server and the outside can be compromised by a man-in-the-middle attack. As stated before, the attacker may deactivate the connection which will lead to an inactive framework or may read the messages. The first point cannot be prevented. It can be tested but may lead to false-true-errors and will, therefore, not be considered. The second point can be prevented by using asynchronous encryption, but this mechanism has to be offered by the other side. It will be considered for use where possible.

9.2.4 Check program

As well as the client or the server software the control program may be attacked. This can be done in similar ways as it has been described before:

1. Manipulating the software.
2. Falsifying the stored information.
3. Manipulating or blocking the transfer of information.

The first point can be tackled, as described above, by checking this software's integrity from another software. This task can be handled by the observed software to reduce the complexity of the framework.

The second point can be achieved by manipulating or deleting the framework's information. The controlling software does need, as well as the checked software, information about the databases, encryption keys, etc. If this information is manipulated or deleted, the controlling software may not be able to message the administrator about the detected attack. Therefore, this information needs to be stored securely. Considering the usage of this information and its structure a secure storage may turn out to be difficult from the framework's point of view. This information cannot be stored in databases as those would introduce new log-in credentials, the encryption of the files has to be done on the level of the operating system and these files should stay in directories where they are not accidentally deleted and can be found easily. Therefore, further security mechanisms besides the encryption offered by the operating system and copies of the files will not be considered.

The third point can be achieved by setting up firewalls or by running man-in-the-middle-attacks. As mentioned before such attacks could be detected with high false-true-error rate and can hardly be avoided. They will, therefore, not be considered.

An attack against the controlling software can harm the integrity of the system. It should, therefore, be considered that the implementation of the controlling software has a strong influence on the framework's reliability.

9.2.5 Chapter outcome

From the attack statements, the following architecture of the framework can be deducted:

1. Several clients, one on each machine, will run and collect information about the software on the machine.

2. On the same machines, a database is running to store the information received from the client program (the installed software on the system) and the server program (the software vulnerabilities).

3. Also, a control program will be installed checking the client software for changes. This control program will be checked by the observed software and vice versa.

4. The communication between the server and the client will be done asynchronously where the server has a private key and the client has the server's public key.

5. The server will run within a secure environment, receives the messages from the client and controlling software, obtain new vulnerability list from the internet, and communicate the vulnerabilities to the clients.

6. Also, the server will have a control program to detect if the server has been changed.

9.3 Efficiency considerations

While this structure can be considered solid (as far as possible), it has efficiency problems. All clients on all machines need to set up a database which does comparisons on the received vulnerability information on a regular basis. Also, each database needs to be checked against a list of software vulnerabilities coming from the server. At this point, these databases could be merged to save computational power and storage space.

The result of such a step would be a server-sided database containing vulnerability information and the information about the software installed on the observed machines. Given this, the update cycle for the vulnerability information will then be done by the server software after getting this information from the vulnerability provider.

The clients themselves would connect to the server database and write their software information into the database. This method offers modularisaion and stability of the framework as the communication on the network will be reduced to the communication between the framework's elements and the database. This allows implementing concepts like zoning of network areas within the framework.

A further improvement is that the centralised database can process queries about all vulnerabilities and host software with a single request (the required information that is stored is described later). Also when a correlation is found, the server software is now not dependent on the network connections of the client.

In addition to the former known attack scenarios, the server landscape is now more exposed as it has become more important. Therefore, a good solution would be a redundant network of several server and database instances collecting the same data and giving information to different administrators. While this may introduce the initial problem of multiple messages to the administrators, an administrator expecting a handful of multiple messages would be acceptable compared to the increased stability.

Furthermore, the integrity of the database has to be guaranteed while multiple instances of the server software read and write into it. This can be given by access rules for database entries as described later.

During the research for the framework, it has been decided to not focus the framework on an implementation of a microservice structure. Microservices are described as "architectures that focus on how services are described and organise to support their dynamic, automated discovery and use."[3] As this framework focuses on security and a secure way to achive the stated points has not been found during the research, it is considered to implement this framework as a static client-server-program and not as automated microservices. Depending on the environment, microservices can be implemented by the programmers by inserting additional functionality so that the decision can be reconsidered later.

During the development of the prototype, the behaviour of the database led to a divergence in design from database normalisaion. During the first implementations, the relational database had been set up according to the rules of database normalisaion. This implementation suffered long computations triggered by many joins of database tables and inaccuracy.

The problem of inaccuracy arose from the different spelling of software packages: The client machine would return the name of a software package as-is and this name may differ from the name inserted into the CVE database. This was

due to different reasons like spelling or the producer's name included in the software package's name (but removed from the software name inserted into the CVE database). Therefore, software names from both the CVE database as well as the client machines were split up and (initially) stored into a separated database tables.

Introducing two more tables in the database (one for the split names of the client machines and one for the CVE database names) raised the accuracy of the program while increasing the false-true-error-rate. Additionally, this method offered the only way to compare the tables of software names against each other while keeping the stored values hashed. On the other hand, this method introduced the previously announced bottleneck in computation as two more tables must be joined. Because of the algorithm, one of these tables needed to be joined a second time.

Information concerning software versions caused similar problems. The version formats from the CVE database and the installed software may differ. Therefore, the version computed by the framework is capped to include a maximum of three numbers separated by two points. This method does not solve the problem as the CVE database also contains shorter version entries and the false-true-error-rate increases. Further research may show how to handle the collection of version information, but research on this issue was out of scope.

The previously mentioned implementation of split software names introduced a bottleneck at the database. During the development of the prototype, the database performance was enhanced by restructuring the ER diagram (as seen in the section 10). The new structure maintained the accuracy of the framework while it speeded up the computation on the database. This computational bonus was big enough to prove that the computation time parallel to all elements increased.

9.3.1 Chapter outcome

From the given statements, the following architecture will be used to establish the framework:

1. Several clients, one on each machine, will run and collect information about the software on the machine.

2. Also, a check program will be installed on the server and client side checking the client and server software for changes. This check program will be checked by the observed software and vice versa.

3. Databases will be build up independently of the servers and clients so that one database may be maintained by several servers and/or clients.

4. The communication between the server, the client, the control software, and the database will be done asynchronously and secured by using asynchronous encryption.

5. The server will run within a secure environment, requests new vulnerability list from the internet, and communicate the vulnerabilities to the database.

6. The server will request a join over the tables of vulnerabilities and find software from the database and information about possible attack scenarios and pass this information on to the administrator.

9.4 Stored information

As described before, this framework stores information about the software and about vulnerabilities. This section focuses on recapitulating the stored information and its storage.

As previously mentioned, hash values will be used to identify changes. These values are generated by calculating an identifying value (the hash value) of them. This value is then used to state if the value under observation has been changed or is the same without storing the original values. This reduces the chances that the data can be identified or changed by intruders.

The client software stores the following information:

1. the network addresses of the databases,
2. a private key for the database,
3. when these databases were updated,
4. a hash value of the state of this databases and,
5. a hash value of the control program's program code.

The control program itself stores the information of the observed software and a hash value of the checked program's code. The server software itself stores the same information as the client software plus the target addresses for the sources of the vulnerabilities and addresses to contact the administrators.

The database stores the information given from both the server and the client software. Those are software vulnerabilities and software lists (each software hashed) to perform a comparison between the software and the vulnerability. Further details are described later.

When an attack occurrs, the value of the information needs to be analysed to determine further actions.

If the information is lost during an environmental disaster, it may be advisable to reimplement the framework. This is because all information can be considered lost instead of stolen or copied. Given that every piece of software starts building up its own database environment, this may slow down the database initially but may be more efficient in the long term as new software versions can be used. Also, the vulnerability list can be downloaded again from the provider and should be available at that time.

If the information is lost or manipulated because of an attack, the attack vector has to be analysed. If the attacker had access to the database and simply deleted the information, it would be advisable to set up a hardened version of the database and restore it with the backup files made before. This may introduce false-positive warnings if the servers and clients are not conscious of that event but may be faster than setting the database up.

The information stored by the clients themselves is more critical as a change or a deletion may indicate an attack. Therefore, an error will be raised if the files differ. This is done because:

1. If a NULL-content is set to describe empty or not-yet-set files, this may be utilised by attackers.

2. The storage of the files should not be done synchronously as an attacker may recognise the position of the controlling software by checking all open files. Also, it is not suitable as this method may introduce deadlock problems which will make the software more complex.

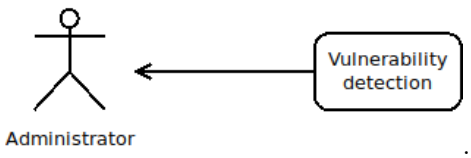
When the attack has been detected on a client or control program, the server program has to be informed (the server may send an alert to the administrator directly). This can be done by either contacting the server software directly or by posting a message into the database. The first approach may introduce more errors as the server software would have to build up a TCP or UDP server listening for the network traffic. The second approach can be acomplished by the already stated methods and is, therefore, preferred.

10 Framework Design

This section introduces the framework from different views (in this order):

1. Use Case diagram
2. Component diagram
3. Activity diagrams
4. Class diagrams
5. Sequence diagrams
6. ER diagrams
7. Activity Diagrams

10.1 Use Case

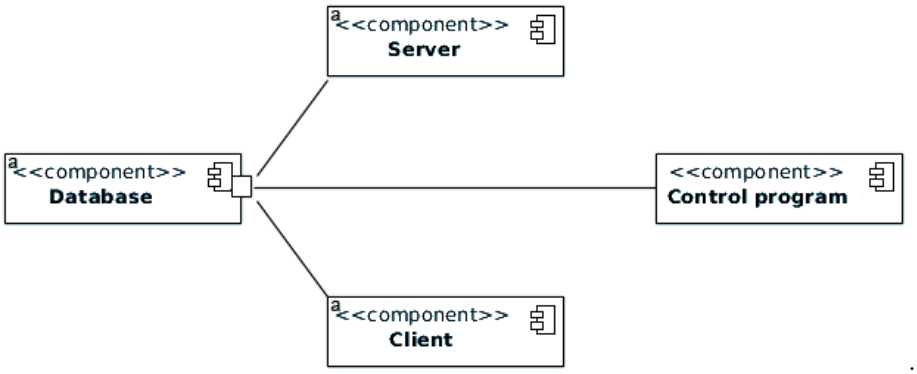


As described above the model's tasks are to inform the administrator about vulnerabilities. Therefore, the following tasks can be identified:

1. Retrieve information about vulnerabilities
2. Check if vulnerable software is used within a given system
3. Send a message to an administrative entity to check this vulnerability.

When considering a Use Case diagram as "a view into a use-case model" and the model as the acknowledgement of "the fact that systems support many different goals from many different stakeholders" (both [11]), those points have been merged together to one single Use Case. This is done because only one stakeholder (the administrator assigned to the framework to be informed if a vulnerability has been found) exists: He expects the program to work correctly and is using the functionality of the program. The facility providing the vulnerability information may be considered as a stakeholder; but as there will not arise any benefit or loss (from the facility's point of view) from not retrieving the information from this facility (or as it may even be implemented within the runtime environment), it is not considered as a stakeholder and, therefore, has no role assigned to it.

10.2 Component diagram



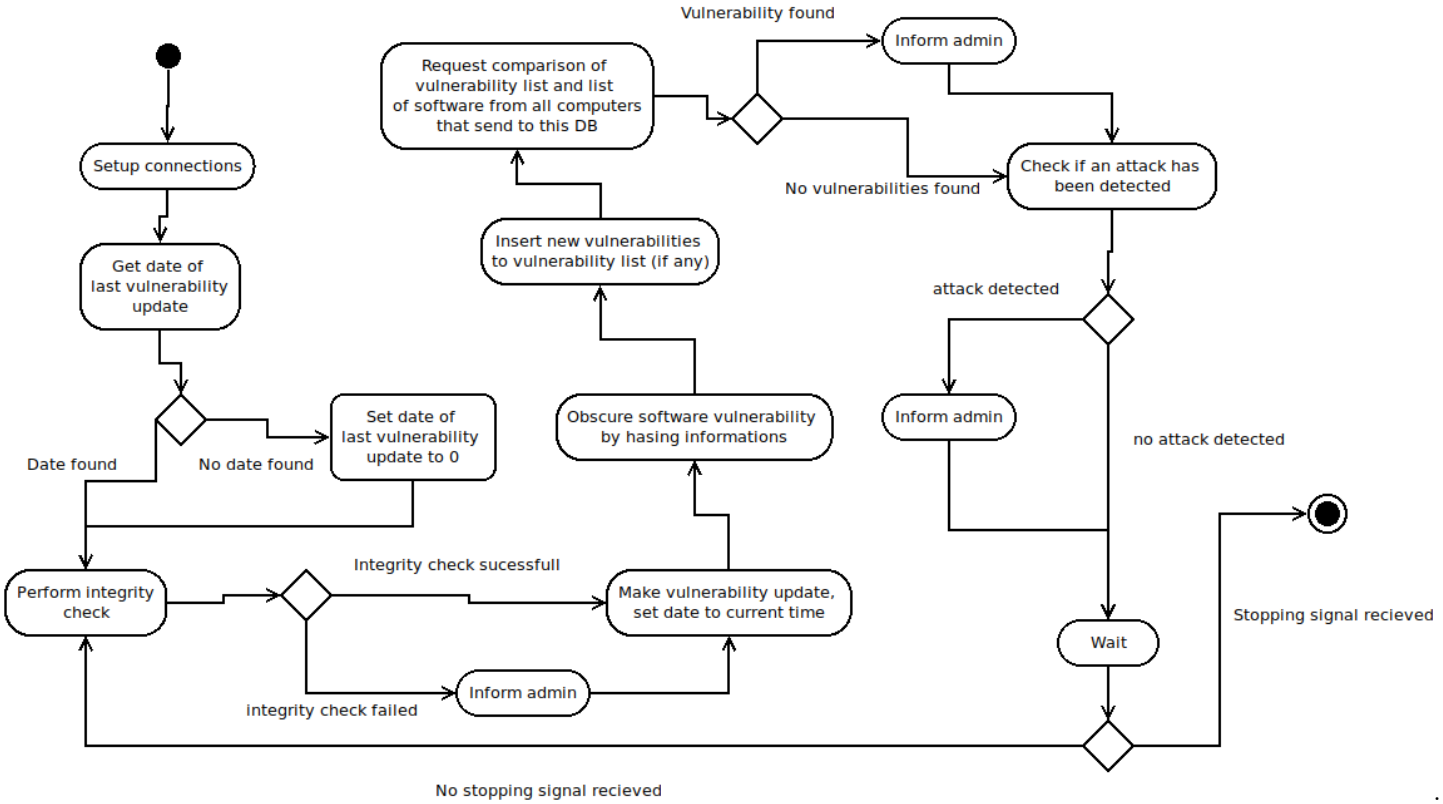
The Component diagram shows the four main objects of the framework as described before: The database, the client instance, the server instance, and the controller instance. The server and client instance do not have a direct connection but communicate over the database with each other. This introduces more stability to the framework while reducing complexity.

The controller instance will check the integrity of the server and client instance. Its communication is handled over the database as well.

Further settings like runtime environment or virtual machine applications are not shown as this has to be set up depending on the environment the framework is running in.

10.3 Activity diagram

10.3.1 Server Activity Diagram



The server starts by setting up the connections to the databases and requesting new software vulnerabilities. All new vulnerabilities that have been added since the last request should be requested. When the date of the last request cannot be found (or the found date is invalid) this timestamp will be set to 0 to request all vulnerabilities available. This finishes the startup of the server instance and fills the server side of the database with the latest data.

Starting the execution loop, the control software integrity and the configuration file integrity will be checked. This is done by comparing the hash value stored locally and the control files stored locally and the machine code of the control software. If an error occurrs or inequalities are found the administrator will be informed of this indicator of an attack. After that, the vulnerabilities list is requested from all registered vulnerability connections and the timestamp will be set to the current system time. This action is done in that order to secure the execution of the framework.

The server then obscures the information it got by hashing all received information and storing only the hashed information. This information is then stored in the database so that vulnerabilities can be compared to stored software values and cannot be read out.

After that, the database is requested to compare the list of software vulnerabilities against all stored computer software lists. As those lists are also hashed lists of versions and software names, the internal comparison mechanism of the database can be used as this is faster than a download of all information and a comparison by the server software.

When a program is detected which is stored in both lists, a message will be transmitted to the registered administrator containing the following information:

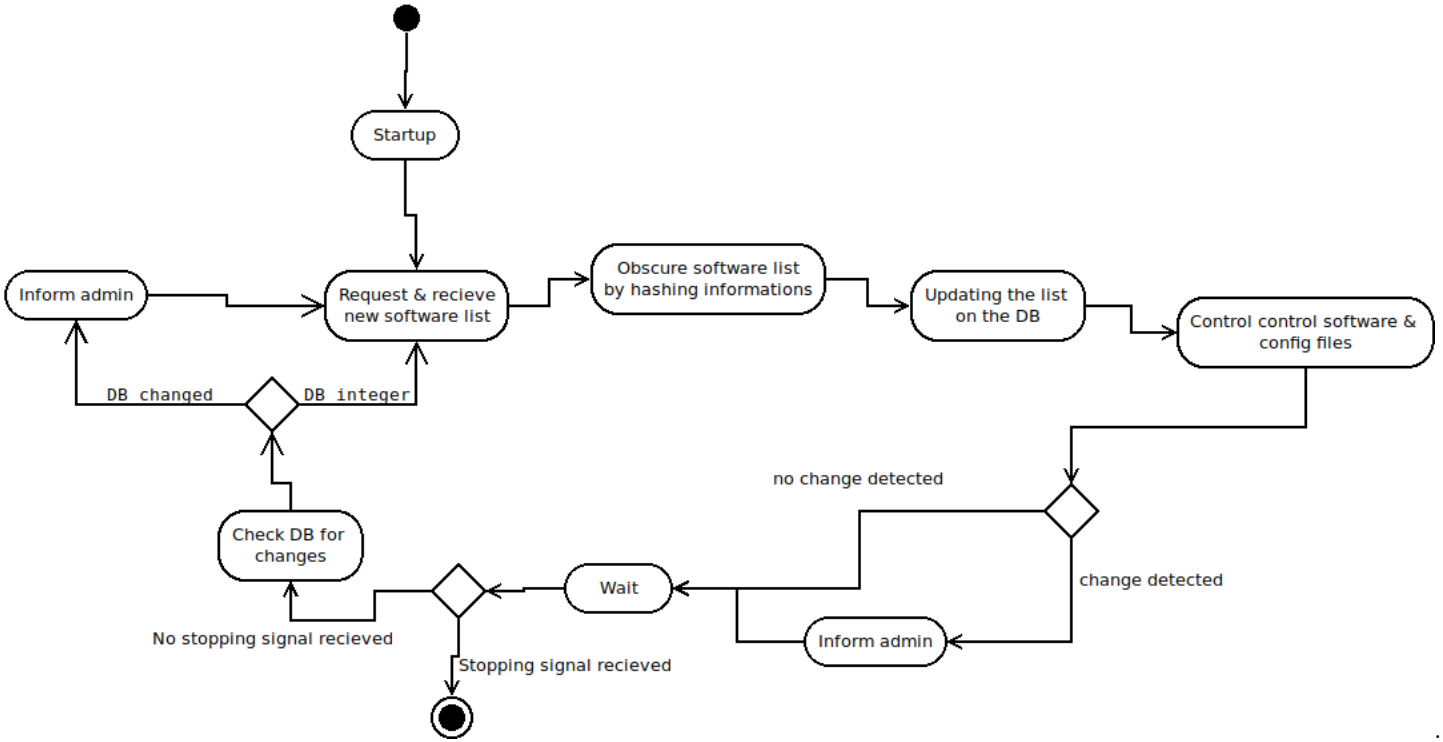
1. the name of the vulnerable software
2. the vulnerable version
3. an indicator of accuracy

This information is given back as described because:

1. Returning every computer having this software may spam the administrators of bigger environments
2. As mentioned before, the software is not stored by its name but by its name hash to prevent intruders from identifying the software used. Therefore, the vulnerabilities will be downloaded again, the values will be hashed again, and checked against the returned values from the comparison. This way the administrators will receive names instead of hash values and this makes work easier for the administrators.

Once this procedure is finished the server requests the lists of attacks which have been inserted into the database by the client and control instances. If this list is not empty, the administrator will be informed (and the list on the database will be cleared). After that, the software waits for a given amount of time and repeats the loop until it receives a stopping signal from the operating system.

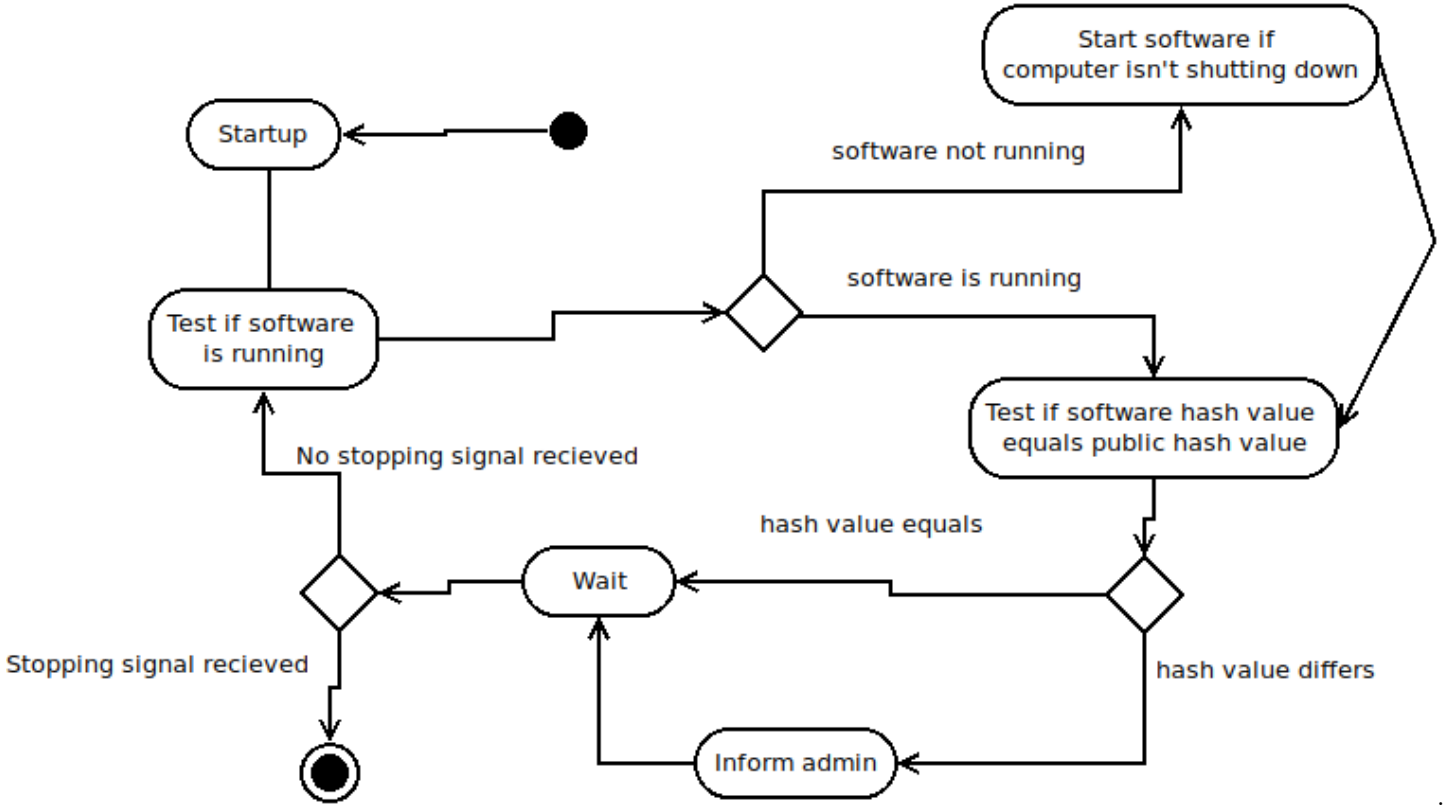
10.3.2 Client Activity Diagram



The client starts by setting up the system and database connections and requests a program list from the system connections. The information on this list will be hashed and stored in the registered database. When this task is done the control software and the configuration files will be checked for changes. If the control software's hash does not equal to the stored hash value, an entry in the database will be made that an attack could have occurred. After that, the software waits for a given amount of time and will restart the loop until a stopping signal from the operating system is received.

If no stopping signal is received from the operating system, the integrity of the database will be checked. This is done by comparing the values inserted by this client into the database to the hash value created by this client from the inserted values. If the test indicates that the values in the database have been changed, the administrator will be informed.

10.3.3 Control Activity Diagram

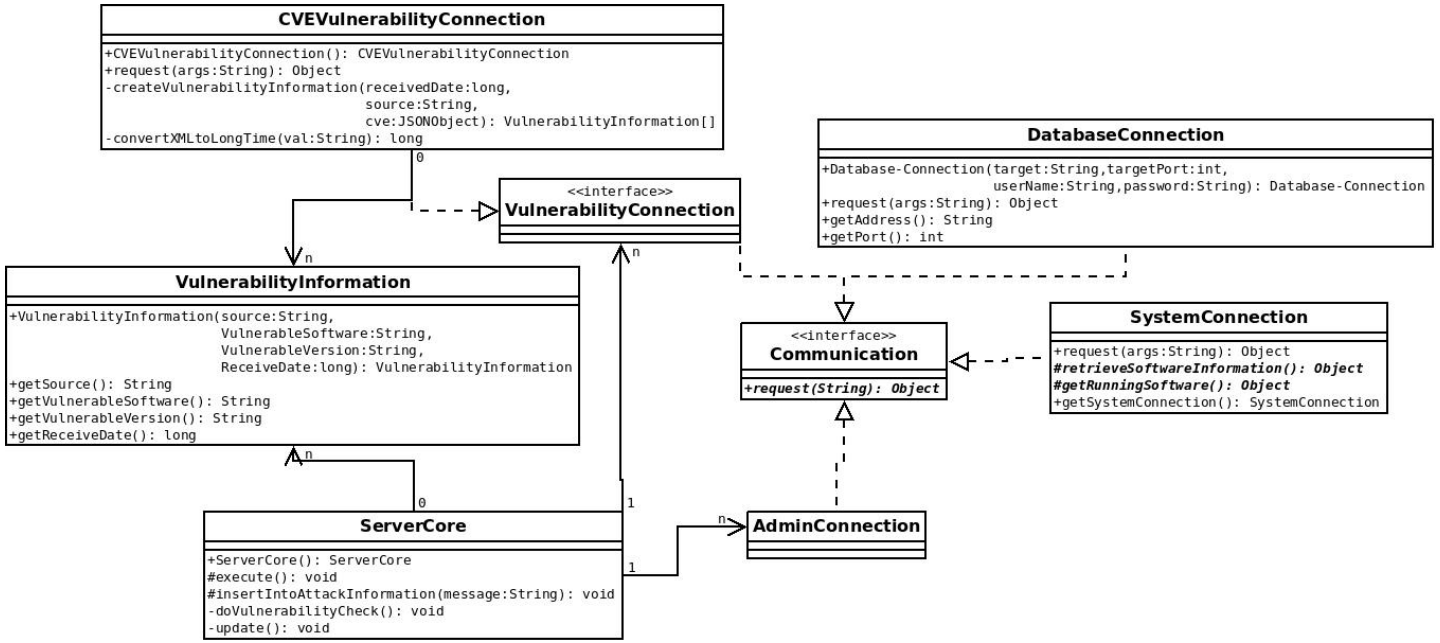


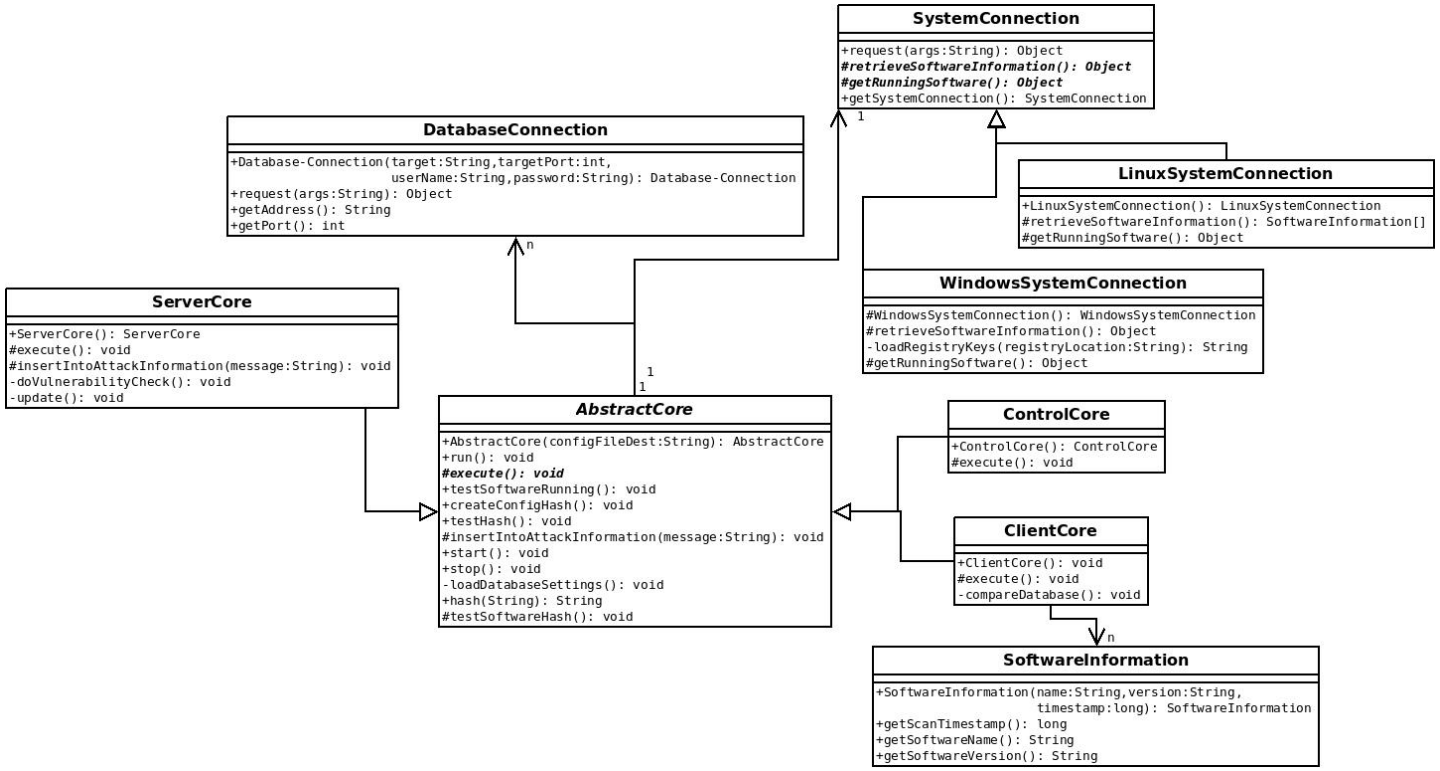
The control program itself observes the assigned software (which can be either a server or a client software or both) by checking if it is running in the background and if its hash code equals the locally stored hash code. Initially, this program starts and sets up its variables. Then an execution loop is started which contains the following points:

1. It tests if the observed program is running. This point states whether a mischievous software has not turned it off. If the software is turned off, it is restarted as long as the operating system is not in the process of shutting down.

2. The software's code is hashed and tested if this hash equals the locally stored hash code. If this is not the case it is likely that the software has been attacked and the administrator will be informed.

10.4 Class diagram




SystemConnection
+request(args:String): Object
#retrieveSoftwareInformation(): Object
#getRunningSoftware(): Object
+getSystemConnection(): SystemConnection
1
DatabaseConnection
+Database-Connection(target:String,targetPort:int,
userName:String,password:String): Database-Connection
+request(args:String): Object
+getAddress(): String
+getPort(): int
n
LinuxSystemConnection
+LinuxSystemConnection(): LinuxSystemConnection
#retrieveSoftwareInformation(): SoftwareInformation[]
#getRunningSoftware(): Object
WindowsSystemConnection
#WindowsSystemConnection(): WindowsSystemConnection
#retrieveSoftwareInformation(): Object
-loadRegistryKeys(registryLocation:String): String
#getRunningSoftware(): Object
ServerCore
+ServerCore(): ServerCore
#execute(): void
#insertIntoAttackInformation(message:String): void
-doVulnerabilityCheck(): void
-update(): void
1
1
AbstractCore
+AbstractCore(configFileDest:String): AbstractCore
+run(): void
#execute(): void
+testSoftwareRunning(): void
+createConfigHash(): void
+testHash(): void
#insertIntoAttackInformation(message:String): void
+start(): void
+stop(): void
-loadDatabaseSettings(): void
+hash(String): String
#testSoftwareHash(): void
ControlCore
+ControlCore(): ControlCore
#execute(): void
ClientCore
+ClientCore(): void
#execute(): void
-compareDatabase(): void
n
SoftwareInformation
+SoftwareInformation(name:String,version:String,
timestamp:long): SoftwareInformation
+getScanTimestamp(): long
+getSoftwareName(): String
+getSoftwareVersion(): String

The class diagram builds up around three blocks: The Communication class (and its implementations), the Core class, and the Information classes.

The Communication class provides an interface-like structure that displays classes capable of communicating with elements outside of the framework (and the database). It describes the function *request* that has a String of characters as input and an Object as output. This method has been chosen as it is an adaptable and stable solution for input variables and an adaptable solution for output variables. This function establishes a connection to a predefined endpoint (or a list of predefined endpoints) and executes a request given as a string. The answer of the endpoint is then processed and returned.

Implementation which represent this class will have to meet different prerequisites:

1. The connection should be established when the request-function is called. This is considered more useful than a permanent connection as those may break when not used (time to leave, etc.), may influence network traffic or may take space and processing time from the endpoint.

2. The error handling should be implemented in the implemented classes. This enables connection-dependent error handling and customise return values.

The AdminConnection class represents the communication to the assigned administrator. The main functionality is to transfer the given string in a readable way to the administrator. Preferably this message should be delivered over a secure line of communication. How this second line of communication or the secure message transfer are achieved in real environments depends on the implementation of this class and the environment. A return value from this class will not be considered (so this class will have to manage the behaviour if a message cannot be delivered).

The VulnerabilityConnection class provides connections to sources of vulnerability information. The input value will consist of a time stamp of the last request for this list and the return value will consist of a list of the received vulnerabilities. This class is abstract as different vulnerability sources may be implemented.

The DatabaseConnection class will manage connections and requests to an assigned database. The input value is a string as an SQL-order and the return value is a string containing the results.

The SystemConnection class is a connection to the underlying operating system. Its purpose is to call system dependent methods and programs. The *request* implementation sends commands to the operating system and returns the information given in a processable way.

The AbstractCore class executes functionality which is used by its implementations (ServerCore, ClientCore, and ControlCore). Those are:

1. Initialising database connections

2. Starting, stopping, and managing the behaviour of the execution loop.

3. Function stubs to implement a uniform core representation

4. hash(String):String This function hashes a given string into another string to obscure the value of a given string and enable comparison.

5. Testing of software under control in both program code and execution status

6. Managing the comparison hash values of the configuration files

The ServerCore class is the main class for server programs. Its purpose is to

1. manage the vulnerability lists by retrieving, organising, and storing the vulnerabilities,

2. compare the vulnerability lists to the lists of used software in the organisation.

The lists of vulnerabilities are stored on a database in the network to minimise the organisational overhead concerning the secure storage of the information. If a vulnerable software is found, the program is supposed to inform the administrator.

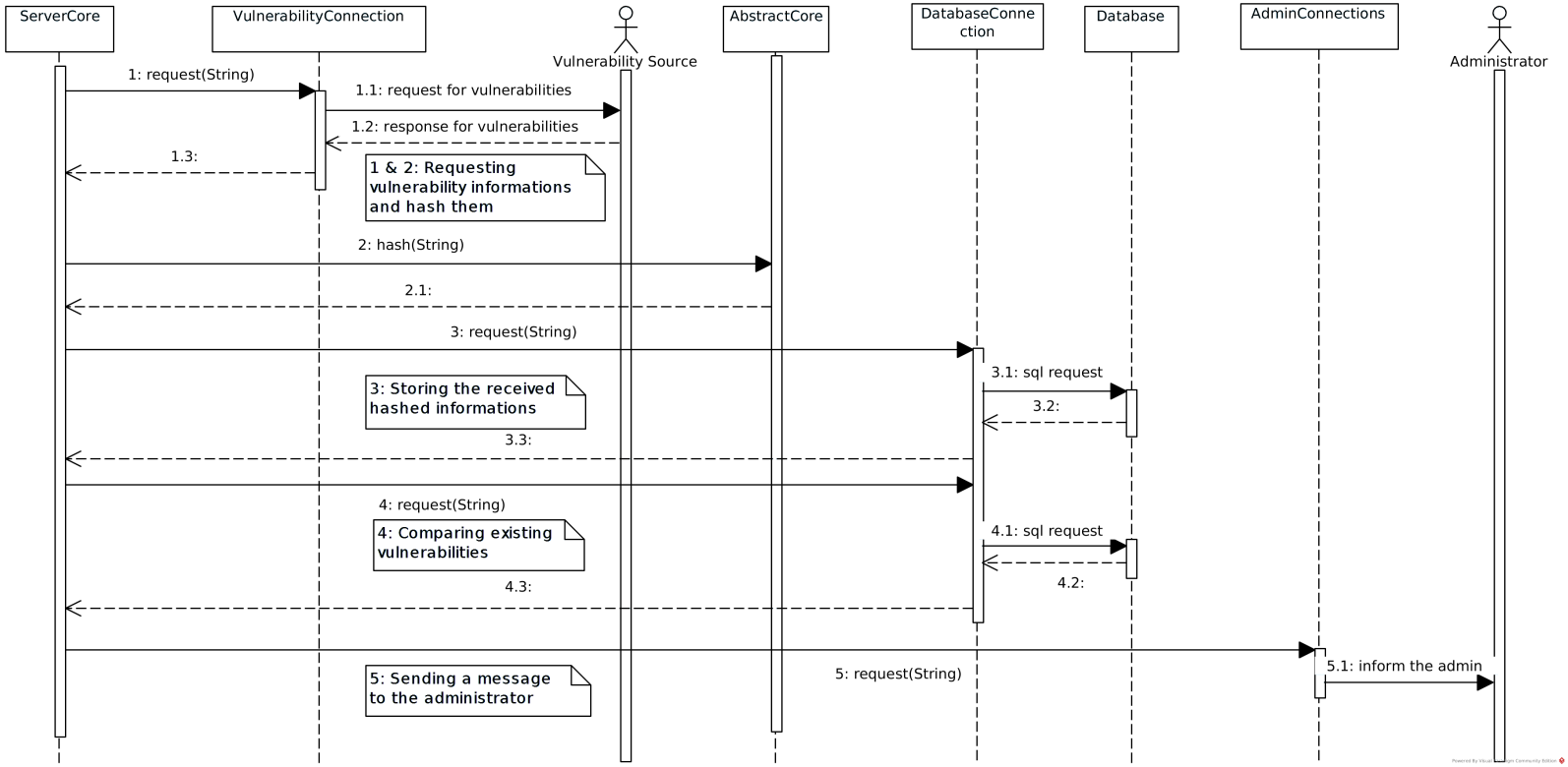
The ClientCore class is the main class for client programs. Its purpose is to organise the software list of the observed computer on the database. Further information is given later.

The ControlCore class initialises the program via the constructor function. It then calls periodically the testSoftwareRunning function to test if an observed software still exists in the task list of the operating system. The $testSoftwareHash$ function reads the binary code of the software and compares it to the last hash value seen.

The VulnerabilityInformation class and the SoftwareInformation class are storage classes as described in the Model-View-Controler pattern and store information received from the connection classes.

10.5 Sequence diagram

10.5.1 Server



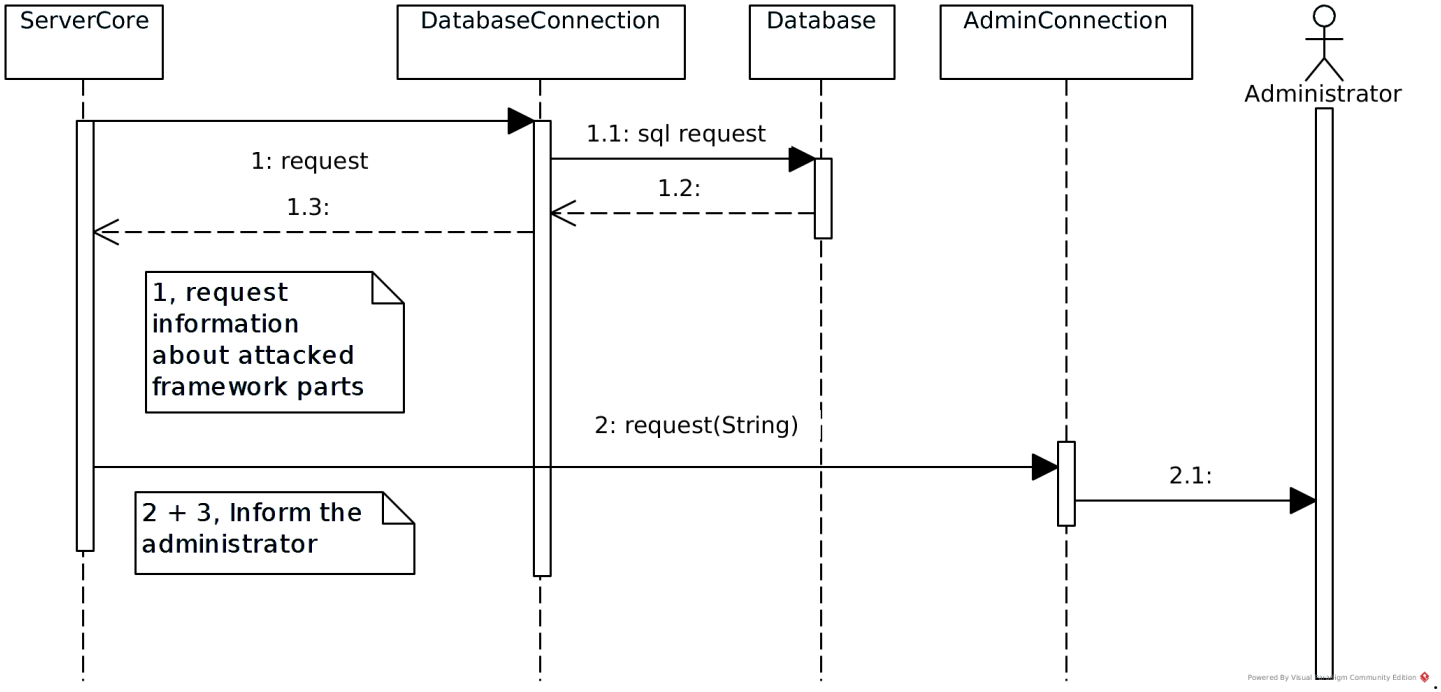
The first sequence diagram shows the structural procedure of the server software during routine execution. The steps are:

1. The server requests the latest vulnerabilities. The VulnerabilityConnection class translates this request into a form transferable to the vulnerability server, sends the message to the service, and receives a list of vulnerabilities. This list is then translated back to fit the VulnerabilityInformation class and then return this information to the server class.

2. After that, the AbstractCore is requested to store the hashed information in the database. The AbstractCore establishes a connection to the database by invoking the DatabaseConnection and sending the SQL request containing the storage purpose to the database.

3. When the storage is done the server software requests a comparison of the stored software vulnerabilities compared with the stored software in the database. This is done by requesting the AbstractCore to set up a DatabaseConnection class and sending the SQL request to the database.

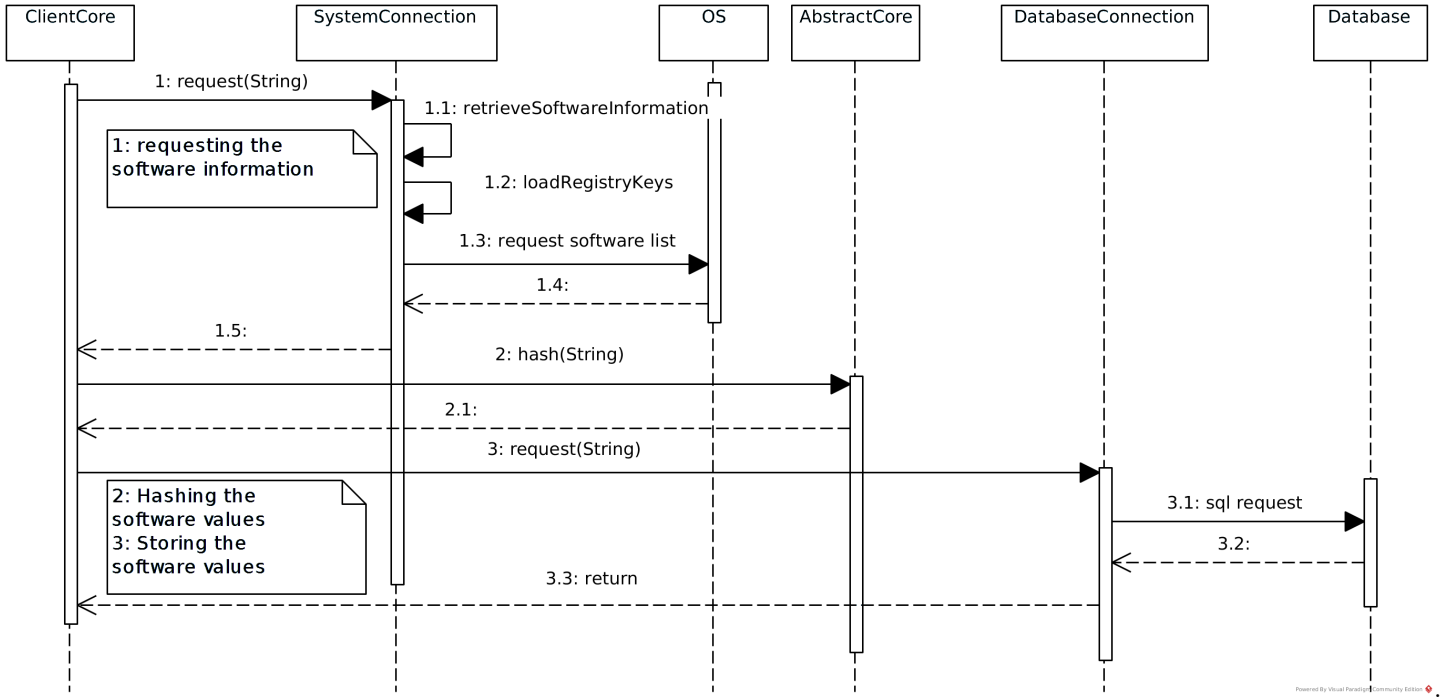
4. If such vulnerability is found, the AdministratorConnection will be established by initialising an AdministratorConnection class and transmits a message containing the vulnerability information to the administrator.


ServerCore
DatabaseConnection
Database
AdminConnection
Administrator
1.1: sql request
1: request
1.2:
1.3:
1, request information about attacked framework parts
2: request(String)
2.1:
2 + 3, Inform the administrator

The second diagram describes the server behaviour if an attack at the client side has been detected and the information has been stored in the database:

1. The server opens a connection to the database and reads all listed attacks and removes them from the database.
2. If the database is not empty, the database will be cleared and all stored information will be transferred to the administrator.

10.5.2 Client

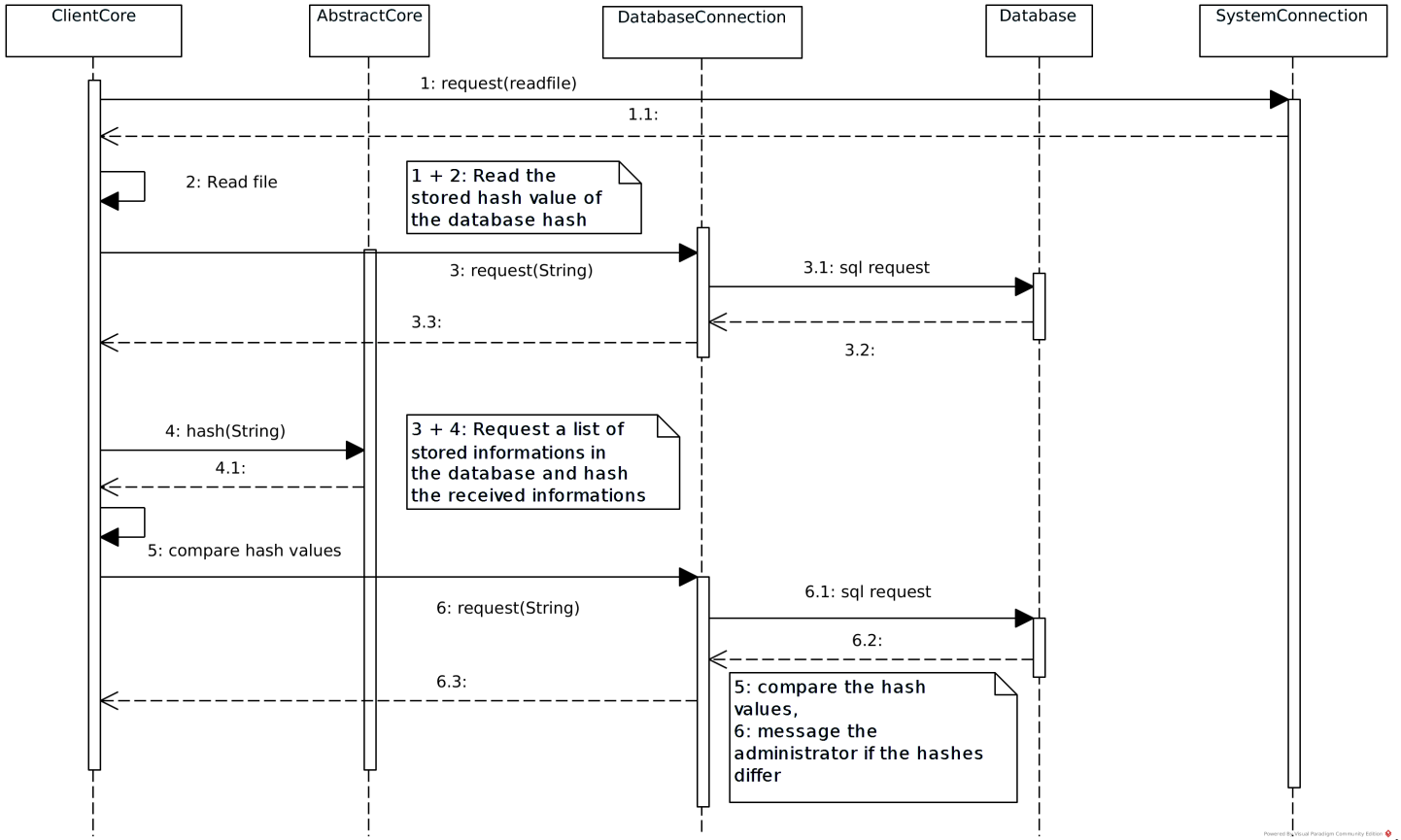

ClientCore
SystemConnection
OS
AbstractCore
DatabaseConnection
Database
1: request(String)
1.1: retrieveSoftwareInformation
1: requesting the software information
1.2: loadRegistryKeys
1.3: request software list
1.4:
1.5:
2: hash(String)
2.1:
3: request(String)
2: Hashing the software values
3: Storing the software values
3.1: sql request
3.2:
3.3: return

The sequence diagram shows the structural procedure of the client software during its execution of the execution loop. The steps are:

1. The software information is requested by calling the request function of the SystemConnection class. This request is separated into several subrequests and transferred to the operating system which will execute these calls and returns a list of the software installed on the system. This list is parsed to fit into an array of the SoftwareInformation class and returned to the client core.

2. The values of the SoftwareInformation class are hashed to obscure the stored information against intruders.

3. The software information on the database will be updated. Therefore, the function *DBrequest* of the AbstractCore will be called which establishes the connection to the database, sends the SQL code to the database, and returns the answer of the database.

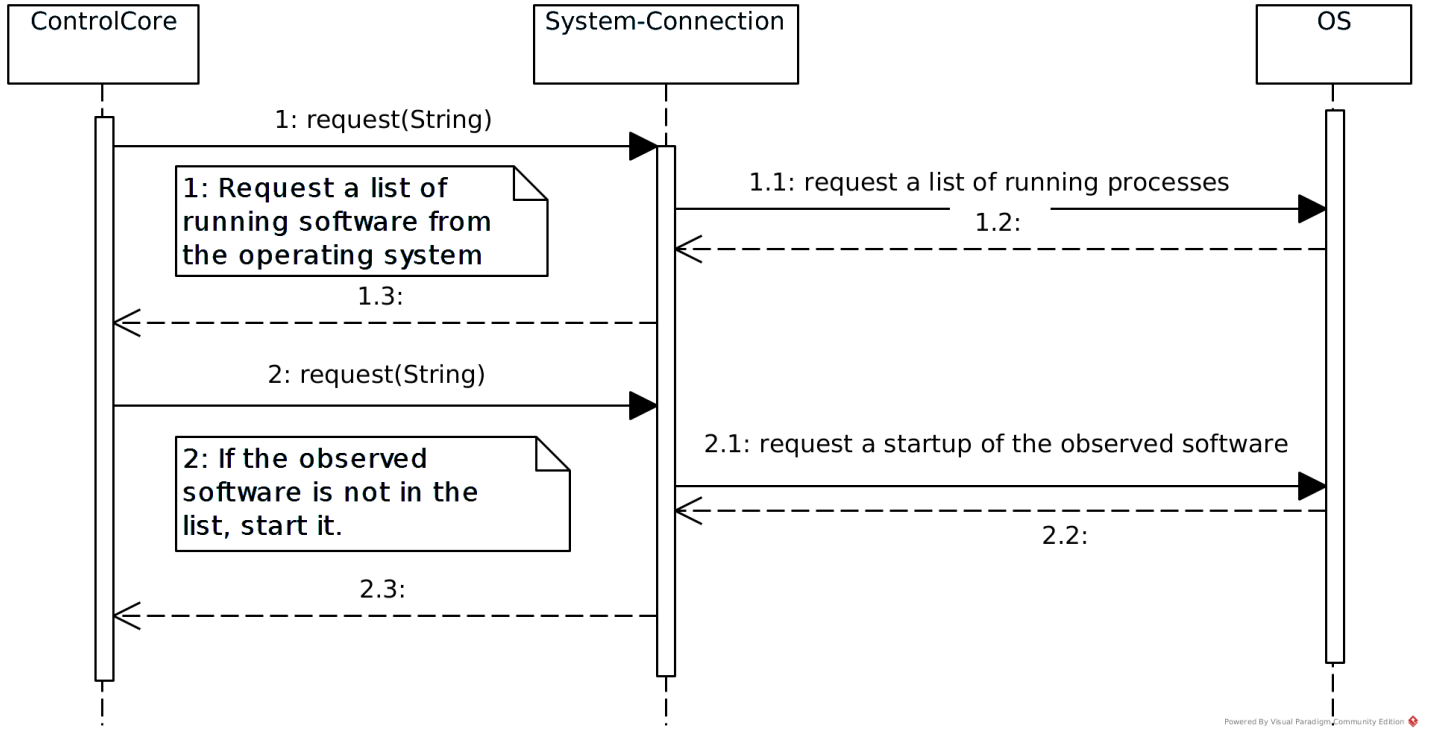

ClientCore
AbstractCore
DatabaseConnection
Database
SystemConnection
1: request(readfile)
1.1:
2: Read file
1 + 2: Read the stored hash value of the database hash
3: request(String)
3.1: sql request
3.3:
3.2:
4: hash(String)
3 + 4: Request a list of stored informations in the database and hash the received informations
4.1:
5: compare hash values
6: request(String)
6.1: sql request
6.2:
6.3:
5: compare the hash values,
6: message the administrator if the hashes differ

The second client sequence diagram represents the method for checking the database against corruption.

1. The file containing the hash value of the database is opened.
2. The file containing the hash value of the database is read. If the file does not exist or it is empty, a new file will be created and the administrator will be informed.
3. The database is requested to return a list of inserted values of this software instance. Servers will receive a list of all inserted vulnerabilities while clients will get their inserted software lists.
4. These results are then hashed as the full lists can neither be stored well on the local hard drive nor easily compared on the client machines.
5. This hash value is then compared against the last locally stored hash value of the database.
6. If the values are not identical it will store attack information on the database to inform the server that an attack may have occurred.

Although the check for the configuration files and for the integrity of the software elements is used in the client this method will not be described in this section. It is similar to the sequence diagrams mentioned in the subsection 10.5.3.

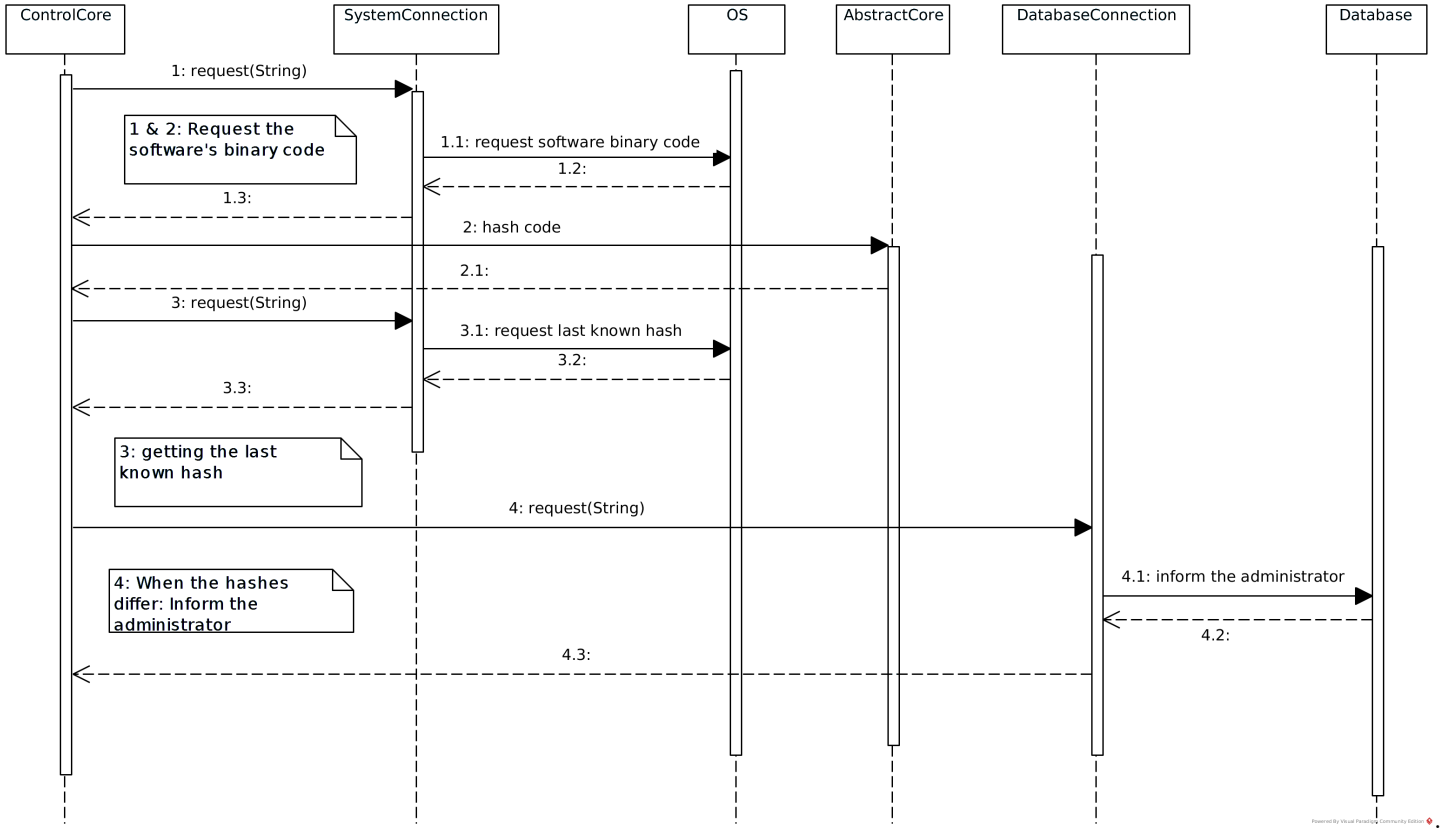
10.5.3 Control



Because of the size of the diagrams for the control software, those diagrams have been separated. The first diagram shows the first part of the control mechanism by checking the running tasks on the operating system:

1. SystemConnection is requested to list all running tasks under the operating system. This request is parsed to a command for the operating system and then given on to that system. The system returns a list of running tasks which is parsed into a list format for the ControlCore.

2. If the software is not running, the SystemConnection is requested to restart the program again by calling the $request(String)$ function. The given parameter is then parsed into a command for the operating system and then executed on this system. After that, the answer will be parsed into a readable format for the ControlCore and is returned.


ControlCore
SystemConnection
OS
AbstractCore
DatabaseConnection
Database
1: request(String)
1 & 2: Request the software's binary code
1.1: request software binary code
1.2:
1.3:
2: hash code
2.1:
3: request(String)
3.1: request last known hash
3.2:
3.3:
3: getting the last known hash
4: request(String)
4: When the hashes differ: Inform the administrator
4.1: inform the administrator
4.2:
4.3:

The second part of the control software targets the integrity of the observed software. This is done in the following steps:

1. A request is sent to the SystemConnection class to get the binary code of the observed software. The SystemConnection class then processes the command by sending it to the operating system, parses the returned values, and returns it to the ControlCore.

2. This binary code is then processed into a hash code by the AbstractCore and is then returned to the ControlCore.

3. This hash code is compared to the stored hash code created by the ControlCore the last time. This is done by invoking the request function of the SystemConnection again and loading the stored hash code.

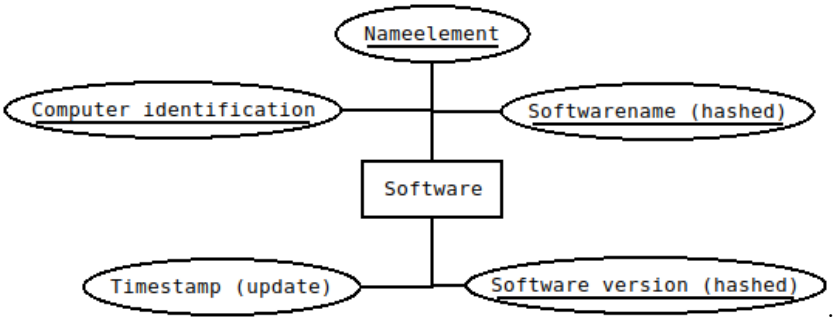
4. If the computed hash value and the stored hash value differ a change may have happened. This is then reported to the administrator by storing attack information at the database using the DBrequest function of the AbstractCore.

10.6 ER diagram

10.6.1 Installed software

The database consists of three different groups: The client part where the client implementation inserts its installed software packages into the database and keeps this information up-to-date, the attack information where the control and client implementation stores information about possible attacks, and the server's where the server implementation inserts the vulnerabilities retrieved from outside sources into the database and compares those vulnerabilities to the list of used software and the attack part.

Starting with the client's table the following implementation will be used:

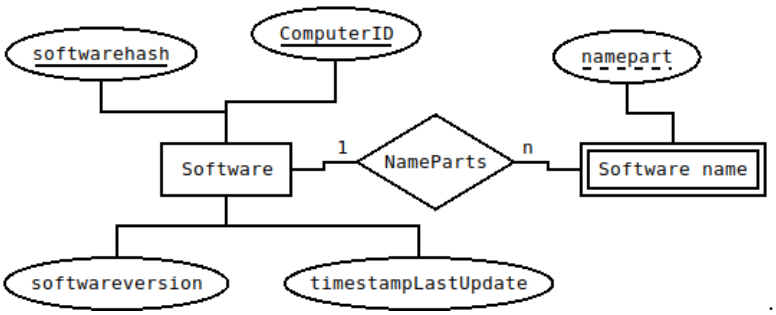


This implementation will be used because of:

1. The necessary information is stored:

   - The computer's name this information has come from,
   - the timestamp when this information has been updated last,
   - the software version installed on the computer,
   - one part of the name of the software.

2. An implementation like the following would have needed additional joints to identify which software did hold which software name. Such a joint has been prevented and the additional time needed to insert this value into the database is small enough to allow such an execution.

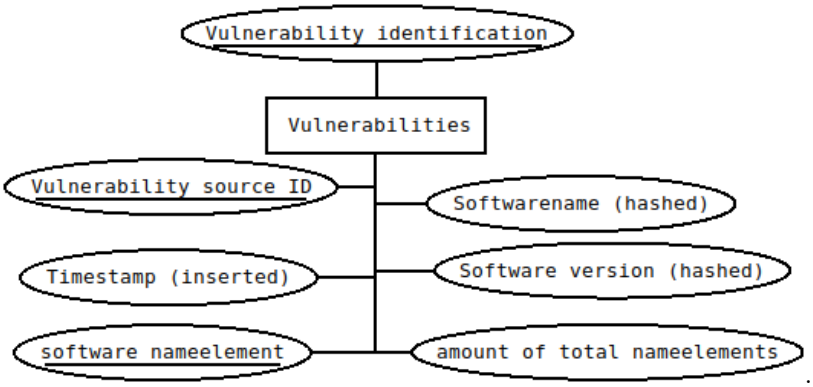


3. It allows a simple rights management system as the information is not split over several tables.
4. Complex m-to-n-relationships have been prevented.
5. The table can later be used to be compared to the server-sided inserts by a single join and, therefore, reduces the time needed for computation on the database.

The rights management for this kind of information can be described as follows:

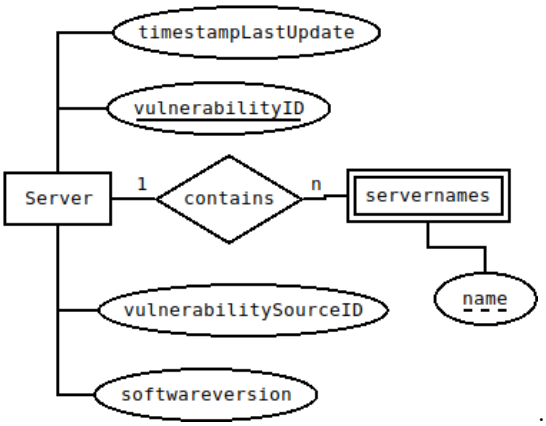
1. The client software which made an entry is able to read, change or delete this entry to guarantee that the information is up to date.
2. Client implementation are neither able to read nor change nor delete an entry that has not been inserted by them.
3. The server instances can read this information and can't change or delete it. This is to guarantee that comparison is enabled but an alteration of the entries is not.
4. The control instance has no access to this information.

10.6.2 Software vulnerability



This implementation will be used to store information about vulnerabilities because:

1. The necessary information is stored:
   - An identification string where this vulnerability information has been received from,
   - an identification string for the vulnerability,
   - a timestamp when this row has been added to the database,
   - a hash code for the vulnerable version,
   - one of the name elements for the name of the vulnerable software,
   - the total amount of name elements for the vulnerable software added this way,
2. An implementation like the following would have needed additional joints to identify which vulnerable software consisted of which name element and how many name elements one vulnerable software consists of. Such joints have been prevented and the additional time needed to insert this value into the database is small enough to allow such an execution. As mentioned before, counting the number of name elements would have needed one additional join which is prevented by storing the total amount of name elements of this vulnerable software name into the table.

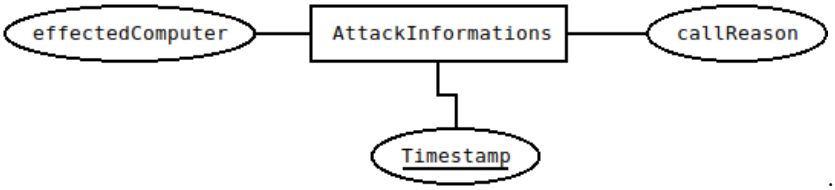


3. It allows a complex right management system as the information is not split over several tables.
4. Complex m-to-n-relationships have been prevented.
5. The table can later be used to be compared to the client-sided inserts by a single join and, therefore, reduces the time needed for computation on the database.

The rights management for this kind of information can be described as follows:

1. Neither the client nor the control instance has access to this information.
2. The server instance has access to this information by inserting and reading it. It cannot delete information in order to guarantee that mischievous programs cannot delete stored vulnerabilities.

10.6.3 Attack information



The database segment for the attack elements only exists to commit intrusion messages to the administrator. Therefore, this table consists of elements containing:

1. the affected computer by the computer's ID
2. the timestamp this message has been inserted
3. the reason this call has been made.

The control software (and the client software) will only be able to insert values while the server will be able to read and delete them. If the sending control program is observing a server and detects an attack (or vice versa) it will send a message to the administrator instead. As described before, this structure has been chosen to keep the modularisaion between the server, the client, the database, and the control program.

The computer's ID has been chosen for identification as the IP address may change over time. The reason for the call can be identified as follows:

1. configuration files have been changed, damaged or removed
2. the content of the database for this client does not correspond to the stored hash values
3. the client or the control software have been changed compared to the stored hash values or the hash values have been manipulated

The client and control instance have read and insert rights on this table to send error messages and compare the stored error messages with new ones (reduces the amount of sending messages to the administrator). The server can read and delete this table.

10.7 Reflection based on research principles

This subsection will focus on analysing the framework with the methods mentioned in [5] and [10].

10.7.1 Reflection against the seven guidelines introduced by Hevner at al. in [10]

This section will analyse the developed framework mentioned above by the seven guidelines mentioned in [10].

1. Design as artefact

   The requirement that the design has to produce artefacts such as models or methods is fulfilled as stated before.

2. Problem relevance

   This model's design process has started by analysing the problem space and constructing the framework around it.

3. Design evaluation

   The evaluation of the framework can be done by measuring it according to the following points:

   (a) Correctness: Does the framework fulfil the requirements mentioned before?

   The main requirement was to support administrators by detecting software vulnerabilities. This requirement has been fulfilled under the circumstances that a source of information, as well as the required information itself, is available. As this framework is not able to detect vulnerabilities on its own it is not able to fulfil this requirement if no information is available.

   Secure storage and information transfer are maintained by using a database from external vendors. This is done as those can offer secure storage and transfer of information while the framework does not need to handle them.

   Some information could not be stored in the database such as database credentials. That information will be stored next to the software in plaintext. This is done because the encryption/decryption key would have to be stored next to those files and would, therefore, render such measurements useless. This information is either stored hashed (a hash value is created out of the information, done for comparison) or will be secured by access rights of the underlying operating system (as even the key for decrypting the information would have this requirement to be secured).

   The software should stay adaptable for other operating systems besides Microsoft Windows. This is fulfilled by using Java, a platform-independent programming language, and by explicitly defining classes to interact with the underlying operating system. As all these requirements are fulfilled as much as possible this section of evaluation is considered successful.

   (b) Comprehensiveness: Does the framework cover what is needed?

   The perfect program to detect software vulnerabilities would be able to find all vulnerabilities when a target software is first analysed. Such a solution is not implementable as an appropriate method for vulnerability prediction has not been found yet.

   Instead, the framework uses information from vulnerability sources. As soon as these sources deliver information about vulnerabilities, it is processed and messaged to the administrator.

   Additionally, the framework is responsible for storing information securely. This is accomplished by using external databases as described before and by using access rights management implemented by the operating system.

(c) Usefulness: Does the design document allow checks against requirements and provide room for their implementation?

The documentation allows checks against the mentioned requirements. All requirements have been listed before in the section 8 and can be pinned down to the elements of:

i. Vulnerability connection (to receive vulnerability information)
ii. Administrator connection (to message the administrator)
iii. Database connection (to store information secure)
iv. AbstractCore, Server-, and ClientCore classes (to implement checking methods and executions)

The framework does provide room for implementation by abstracting the classes and allowing further extensions of mentioned classes. Additionally, this implementation is not bound to a specific runtime environment, operating system or programming language (the latter needs to implement object-oriented programming).

4. Research contributions

   Based on the literature review an assistant design is carried out which is scientifically sound and follows established design methods and criteria.

5. Research rigour

   The thesis follows a sound and proven methodological approach. The design document is checked against the requirements and the prototype implementation will be tested against requirements and against design documentation.

   The results achieved by the thesis will be compared with established literature in the field.

6. Design as a search process

   This thesis started with the aim of supporting administrators by handling software vulnerabilities within their Microsoft Environments. After the literature analysis, the aims for a profound solution have been stated and based on both an initial approach has been described. This approach has been checked against the concepts mentioned in the literature analysis and evolved around its requirements and possible attack scenarios. Based on this evolved structure the framework has been described and built.

7. Communication of research

   The basic concept of the master thesis in its advanced stage is presented in the Master seminar. Regular meetings with the supervisor and with colleges are used to discuss issues arising and to critically reflect the steps (chapters) of the thesis. In an ideal case, the thesis outcome would be submitted to a workshop or a conference.

10.7.2 Relation to "Great Principles"

This section tackles the analysis of the mentioned framework with respect to the "Great Principles of Computer Science" by [5] and is structured by its categories computation, communication, coordination, recollection, automation, evaluation, and design.

1. Computation

   This chapter focuses on the computability of the given information. It states that each state can be turned into a representation and that the computation is a series of states (comparable to the Turing Machine). Such an analysis would be out of scope as a too wide range of possible interactions would have to be taken into account. Therefore, this section offers a black box analysis for the evaluation of the complexity of the computation to give a proof and reliability analysis of the framework. These calculations are separated depending on their implementation in the framework.

   - Server:
     - Receiving and storing all vulnerabilities.
       Expecting that the received information needs to be transformed and stored in the database, each position is expected to be $O(n)$ with respect to the received vulnerability information. As the storage mechanism is handled by the database, the storage mechanism is expected to be $O(n)$ for the framework (it can be seen as $O(1)$ when all vulnerability entries are transferred in one bulk).
     - Comparing vulnerabilities and messaging the administrator.
       The framework delegates the comparison for vulnerabilities to the database. If this is not possible, the complexity would be $O(n*m)$ (where n is the amount of software entries stored in the database and m is the number of known vulnerabilities in the database). This can be reduced by using hashing mechanisms for one of the information lists: This would result in the complexity of $O(n)$ (storing all software in the hash table) $+O(m)$ (comparing the vulnerabilities against the hash table) $= O(n + m)$.
       When this information is computed, a readable format needs to be established. Therefore, all vulnerability entries will be downloaded again and compared against the found vulnerabilities. This complexity can be described as $O(n \cap m) * O(m)$ which can be reduced again with hash table mechanisms to $O(n \cap m) + O(m)$.
       When these lists are compared, they will be sent to the administrator: The complexity will be $O(n \cap m)$ to prepare the message and $O(a)$ to send this message (one for each administrator a).
     - Searching attacks and messaging the administrator.
       All attacks are read from the database, the stored information will be taken as is (no conversion), and will be concatenated to one message to the administrator ($O(n)$) which will be sent to all receiving administrators ($O(a)$)

   Given the description above the maximal complexity of each part is (adding all variables):
   $O(n)+O(n*m)+O((n \cap m)*m)+O(a)+O(n \cap m)+O(a)+O(n)+O(a)$
   As $O(n \cap m)$ is always smaller than $O(n)$, the first one will be replaced by the second one (other similar operations will be merged to make it more readable):
   $O(n) + O(n * m) + O(a)$

Again as $O(n)$ is smaller than $O(n*m)$ the first one will be replaced by the second one:

$O(n*m)+O(a)$

Therefore, the worst case is $O(n*m)+O(a)$. This complexity can be enhanced when using hash tables to $O(n+m)+O(a)$. As it cannot be stated which of m, n or a is bigger than each other (it depends upon the implementation), this formula will be considered to be the server's complexity calculation.

- Client:

  Receiving and storing software information found on the client machine:

  The client software receives the software installed on the operating machine and stores this information into the database. The receiving mechanism can be described as $O(n)$ (where n is the amount of software installed) because each software information needs to be adopted for further processing. After that, the software is stored in the database as described before and is described as $O(n)$.

  Because of this, the complexity of the client software can be seen as $O(n)$.

- Control:

  - Control program.
    The control software checks if its counter program is running. This mechanism can be stated as $O(n*m)$ where m is the amount of software running on the computer and visible to the control program and n is the amount of software to be checked. Also, this can be reduced by using hash tables to get a complexity of $O(n+m)$
  - Control program hash.
    The control software checks if the counter program has not been changed. This is done by loading the software's code, hashing it, and comparing it to the stored hash value of the last check. This has a complexity of $O(n)$ where n is the amount of programs to be checked.

  The control program has, therefore, a worst-case complexity of:

  $O(n*m)+O(n)$

  As $O(n*m)$ can be reduced by using hash tables and $O(n)$ is smaller than $O(n+m)$ the complexity will be described as:

  $O(n+m)$

  Again it can't be determined if n or m is bigger (which depends on the implementation).

The overall complexity of the framework can, therefore, be written as (worst case):

$O(server)+O(client)+O(control)$

which can be written as:

$O(softwareToCompare * vulnerabilityInformation) + O(administrators) + O(softwarePerMachine) + O(softwareToControl * softwareRunningOnMachine)$

As mentioned before this formula can be simplified to:

$O(softwareToCompare + vulnerabilityInformation) + O(administrators) + O(softwarePerMachine) + O(softwareToControl + softwareRunningOnMachine)$

When using the general O-notation everything needs to be reduced to n, so the resulting complexity is:

$O(n)$

The second point is scalability: How far can the software be scaled. Therefore, the maximal amount of computed data will be calculated:

- Every client reads out a list of software and stores it in the database
- Every server reads out a list of vulnerabilities and stores it in the database
- Every server checks the vulnerability list against the software lists from the database

Given that there are fewer attack messages in the database stored than entries in the software or vulnerability lists, this point will be ignored. Proof: Every client should have a maximal amount of attack messages types so the maximal amount is $< amountOfClients > * < amountOfAttackMessageTypes >$. So this value is capped while the maximal amount of entries in the vulnerability list is not explicitly capped. It can be proofed easier when the amount of software installed on the clients is bigger than the amount of attack message types, but this cannot be guaranteed and depends on the environment.

So the limiting factor is the database. The database is the bottleneck where the framework communicates with each of its parts. So the limiting factors are:

- Network communication and
- Database processing power.

The framework itself is scalable. This can be shown when adding additional elements:

- Every client or server software should have at least one control software to observe the client/software part. The limiting factor again is the database to store attack information in.
- Each computer can have an infinite amount of clients (when set up properly) and an infinite amount of databases to report to.
- Each server can have an infinite amount of databases to control and an infinite amount of vulnerability sources to receive information from.

Therefore, the framework can be seen as scalable as all elements depend only on the database and allow additional instances of itself. The framework does not rely on choreography or uniqueness of the single instances so no changes would be needed to be made to change the number of instances of the framework running in the network.

2. Communication

   This chapter on the Great Principles interprets information as messages. As these messages are manipulated during the execution, this Great Principle asks in which way these messages are manipulated, stored, and transferred within the framework. The following analysis of the framework will take a closer look at the control and data flow within the framework.

   The control flow does almost not exists. Every unit of the framework except the database is a unit on its own and does not rely on orders or control mechanisms from outside. A unit may control the machine code and the process status of other units but does not give orders of any kind during its execution.

   All units communicate with the database. There the communication request may be interpreted as a control flow from the framework to the database. The database then receives, stores, manipulates, and sends back the information requested. The rights management has been described as:

   (a) Clients insert and read attack information and insert, read, and delete software information from their computer only

   (b) Control software is allowed to read and insert attack information

   (c) Server software is allowed to read and delete attack information, insert vulnerability information, and read software information from client machines.

   These are the only rights on the database so that a machine which is allowed to insert is not allowed to read the information. Furthermore, information concerning the software list and the vulnerability list are only hashed values so that a readout may enable users to compare the values while preventing them from knowing what information this value contained before storing.

   This information is then read out by the server software for further processing. Therefore, it can be seen as an indirect control flow concerning attack information as these order the server software to inform the administrator by using the given information.

   The data flow follows the same direction. There are three sources of information:

   - The server storing information about vulnerabilities retrieved from outside sources,
   - The client storing information about possible attacks detected and of the software packages found on its host machine,
   - The control software storing information about possible attacks.

While at the second and third point the information flows into one direction, the server may request vulnerability information, software information, and attack information. When this information has been requested once, they will not be written back into the database but, when considered necessary, are given to the administrators assigned.

3. Coordination

   This Great Principle focuses on the way independent agents communicate to reach a common goal. It describes how the analysis should focus on the interaction between the agents, how they coordinate, and which roles they have. Instead of implementing a coordination mechanism, loose coupling is implemented because of the reasons mentioned in section 9 and 9.3.

   To recap, the agents communicate by putting messages and information into a database. Direct communication between the agents is not planned because this would increase the complexity of the program. By using a database, the agents do not rely on the availability of each other: If one agent is down because of 'home office' or a broken machine, every other agent can go on doing its work.

   Given later development focus on more accurate structures, additional information stored into the database could make the framework flexible. That way, the framework can be easily adapted to the need of the developer or the execution environment.

4. Recollection

   The Great Principle of recollection considers the source of, storage of, and access to information.It focuses on the reasonable usage of information and has become more and more popular in recent years (especially because of the GDPR implemented in the EU).

   For the presented concept the sources of information can be divided into these categories:

   - Attack information: This information is created by the programs of the framework during execution if unexpected behaviour has been encountered or if an unexpected change in the data occurred.
   - Software information: This information is received from the operating system by the client program.
   - Vulnerability information: This information is received from the vulnerability source by the server program.
   - Configuration data: This information is used to configure the behaviour of the framework parts by e.g. specifying the target database to use.

   The first three points are stored on a database. This reduces the overhead of developing a secure and consistent storage mechanism. Software- and vulnerability information is stored on the database hashed. This guarantees that the values can be compared to each other while obscuring this information to outside attackers with access to the database.

   The access rights are:

- Attack information: Client and Control software is allowed to insert and read information, the server software is allowed to read and delete the information
- Software information: The client software is allowed to insert and delete the information while the server software is allowed only to read the information
- Vulnerability information: The server is allowed to read and write information, deletion is not allowed.
- Configuration data: The access rights for the configuration data are handled by the operating system and are restricted for read and write access to the user who executes the framework.

5. Automation

   The section "Automation" of the Great Principles refers to the mapping of knowledge or behaviour from one system to another. Because of this description of automation, this framework can be described as doing no automation.

   The human interaction with the framework consists of setting its configuration files. These files are used to configure the program during the execution, the target databases, and the checked counter program (for the control implementation this is either a client or server software, for the client or the server it's a control implementation).

   After this setup, the interaction between human and the framework happens only when the program is messaging the administrator. Besides that, human-computer-interaction is not supposed to happen.

   A possible exception may occur when considering vulnerability information. The server software does not delete vulnerability information out of the system. Therefore, the deletion of old vulnerabilities out of the system to reduce the size of the database may be considered as a human interaction as well.

   Besides the communication with the administrator, the goal of the framework to automise the test for software vulnerabilities is no automation as described before because it does no mapping from one system to another. The framework reads out the software installed on computers and combines this information with the information of vulnerable software. During this process, there is no mapping from the real world into the cyberspace or vice versa.

6. Evaluation

   The evaluation of the framework will be done as described above in the section "Reflection against the seven guidelines introduced by Hevner at al. in [10]"

7. Design

   The section design mentions the way the models of this framework have been created. These have been created in an iterative way by:

   (a) Analysing the current situation in the field.

(b) Stating the requirements.

(c) Describing an initial approach.

(d) Describing possible drawbacks, additional requirements, and attack vectors of the initial approach.

(e) Hardening the initial approach.

(f) Defining and modelling the framework.

(g) Testing the framework by creating a prototype.

(h) Adapting the framework to the errors encountered during the prototype development.

The design relies on design patterns, among others:

- Model-Controler-Pattern (a reduced version of the Model-View-Controler-Pattern as the interaction between user and program is minimal)
- Singelton-Pattern (used in the SystemConnection-Class to guarantee that only one version of system connection can be initialised)

The quality of the design has been evaluated by comparing the design to well-known patterns as mentioned before and by comparing the results to other designs created in the field such as [1] or [27]. Additionally, the design has been implemented in a prototype described later where the results from the prototype have been taken for the further development of the framework.

11 Implementation

11.1 Aspects of the implementation

This subsection focuses on the operating figures the prototype offers.

11.1.1 Correctness

Does the program work correctly as it is intended? Because of possible bugs, this question cannot be answered with yes. Therefore, we can make two measurements:

First, we can test the code by using testing mechanisms. This code, as described later, has been written in Java and can, therefore, be tested using JUnit control mechanisms to test single functions and functionality of the program. This method has been implemented and used whenever a function could have been tested. The results showed that the function worked within their assigned tasks correctly.

While the first test focuses on scripted scenarios, the program has been measured in real-world scenarios on different computers. The prototype was able to acquire the needed information from the client computers (different versions of Microsoft Windows, developed for Microsoft Windows 7 but running also on Microsoft Windows 10) and send this information to the assigned database. The server was able to compare the collected vulnerabilities and the collected software information and to comparisons on these.

The prototype, therefore, can be considered as correct considering the narrow test spaces used above. As mentioned a complete guarantee cannot be given and needs to be verified by other researchers.

Furthermore as described in the subsection 11.1.2 the program expects uncertainty when handling software names and versions. Therefore, the program will introduce false-true-errors to maintain a high completeness level. This approach has been chosen because of the security relevance of the program.

11.1.2 Completeness

The developed prototype relies on information from different sources. The following part describes how the prototype handles information that does not guarantee completeness.

The server software will store possible vulnerabilities that have been found on computers in the network in a file in CSV format. This file contains three rows: The first row gives back the name of the vulnerable software, the second row gives back the vulnerable version, and the third row gives back the level of precision this entry has.

While the first two rows should be clear and self-explaining, the last row is a computation of the following criteria:

- The initial level is given from a comparison between the software names in the database and the software names in the vulnerability lists:
  - If they have less than 50% in common, this vulnerability is ignored
  - If they have $> 50\%$ in common, the initial level value is 2
  - If they have $> 90\%$ in common, the initial level value is 3
- After that the version will be compared:
  - If both versions are equal, the level will not change
  - If the version of the vulnerability list is empty (when the stored version in the CVE file is complex or empty), the level will be decreased by 1
  - If the version of the vulnerability list does not equal the software version, the level will be decreased by 2

After that analysis, all found comparisons will be stored in the CSV file. The method described above is used to ensure that spelling errors or misclassification will not remove values from the list.

Based on this concept the software now outputs vulnerabilities and possible vulnerabilities found in the database. The search space can, therefore, be described as:

$\sum(AllVulnerabilitiesCollected) * \sum(AllSoftwareSolutionsInstalled)$

As described before, this search space is considered to be executed by the database which provides a sophisticated method to handle the amount of information. At this point, it can be said that the computation showed the following behaviour.

The following list is sorted by the total amount of packages the clients have reported to the database. The tables show the times measured in total, the time needed to get the information about vulnerabilities (to describe these to the

administrator), the time needed to calculate on the database, and the time only the server has been executing. These values have been acquired from a server running Kubuntu 18.04 64 bit on an Intel Core i7-7500 CPU at 2.7 GHz for each of the four cores and 7.7 GiB RAM. As this computer also provides update routines, the database, and a GUI, these values can be seen as a bottleneck analysis.

- For 15 packages

Category	Total	Vulnerabilities	Database	Server only
Average	24.7 s	16.6 s	3.2 s	4.6 s
Median	23.0 s	15.6 s	3.2 s	4.6 s
$\sqrt{Var}$	4.2 s	4.1 s	0.09 s	1.1 s
Minimum	18.7 s	11.6 s	3.2 s	3.8 s
Maximum	36.6 s	28.3 s	4.2 s	15.8 s

- For 68 packages

Category	Total	Vulnerabilities	Database	Server only
Average	39.7 s	18.2 s	15.6 s	5.9 s
Median	37.6 s	16.1 s	15.7 s	5.9 s
$\sqrt{Var}$	6.9 s	6.1 s	1.0 s	1.4 s
Minimum	31.6 s	11.5 s	15.1 s	4.8 s
Maximum	69.0 s	43.0 s	23.6 s	16.9 s

- For 121 packages

Category	Total	Vulnerabilities	Database	Server only
Average	65.9 s	27.9 s	31.4 s	6.7 s
Median	63.4 s	25.6 s	31.0 s	6.6 s
$\sqrt{Var}$	11.8 s	12.0 s	2.8 s	1.5 s
Minimum	50.1 s	12.7 s	30.6 s	5.7 s
Maximum	96.2 s	58.0 s	51.6	16.6 s

From this computation several bottlenecks can be identified:

1. The retrieval of vulnerability information has an impact on the total execution time ranging from about 30% to about 50%. As the retrieve information has been almost equal in all cases it cannot be determined why this section shows this behaviour. These measurements may speed up by reducing the target time span to scan for possible vulnerabilities or store a list of vulnerability information on a database in the local network.

2. The database has only been given the structure of the tables to calculate with. It has not been adapted to the needs it has to fulfil (like using SQL indices) or adapted to the hardware it runs on. Therefore, these values can be seen as a guideline and may be improved in further steps.

3. From the hardware side, it can be said that the CPU has been a bottleneck: The database and the server program may have computed faster when a more powerful processing unit has been installed. Concerning the hard disc or the RAM, it cannot be determined in which way these have slowed down the processing.

4. The structure of the program and the database have been set up to support a clear understanding of the implementation of the framework and can be improved further. This starts at the repeated download of the vulnerability information, the structure of the database or the repeated checking if the control program is running.

The following table shows the correlation between all values and the stored amount of packages. The significance test has been calculated with a degree of freedom of 289 (291 values for a degree of freedom of $n-2$)

	Total	Vulnerabilities	Database	Server only
Correlation	0.8991	0.4786	0.9898	0.5430
Significance test	34.93	9.27	118.38	10.99

Considering every value above 0.5 as a strong linear correlation between these values[4] and a summed value of the t-distribution of 1.65 (for a degree of freedom of 289 and an α of 0.05) it can be stated that the time needed for computation depends on the amount of stored data. The following table describes the statistical difference between a linear estimation (x) and a polynomial estimation of degree 2 ($x^2 + x$) (with the number of stored packages as x):

	Total	Vulnerabilities	Database	Server only
p-value x	$< 2.2e-16$	$< 2.2e-16$	$< 2.2e-16$	$< 2.2e-16$
F-Test x	1220	85.88	$1.401e+04$	120.8
R^2 x	0.8085	0.2291	0.9798	0.2949
p-value $x^2 + x$	$< 2.2e-16$	$< 2.2e-16$	$< 2.2e-16$	$< 2.2e-16$
F-Test $x^2 + x$	698.6	55.93	9248	61.8
R^2 $x^2 + x$	0.8291	0.2798	0.9847	0.3003

The p-value of all results shows that the estimator is sure to have found the most suitable solution as the p-value indicates the variance the resulted estimation could have. Because of the similar F-Test and R^2 values for each category it is assumed that the polynomial estimation does not provide a significant benefit compared to the linear estimation. As mentioned before the database seems to compute almost linear on this task, the vulnerability retrieval computes non-linear because of unknown outside factors and the server itself does not provide a sufficient increase in the R^2 value to describe the data. Therefore, a linear approximation will be favoured to describe the behaviour of the server side of the program.

11.1.3 Complexity

The complexity of the program can be defined as following:

- The Control software has a complexity of $O(n)$: It compares on the checked software's hash, checks for configuration integrity, and send a message to the administration database when needed. Every used function has the notation of $O(n)$. Therefore, the hole function has the notation of $O(n)$.
- The Server software has a complexity of $O(n)$: All parts of the server software use the data several times but are not more complex than $O(n)$ as they walk through lists of entries (or compare those entries against hash tables which have a complexity of $O(1)$). It may be possible that the computation on the server side used to compare the vulnerability list

against the software list does not provide a complexity of $O(n)$, but this cannot be confirmed and is, therefore, considered out of scope. Therefore, all functions and subfunctions offer a complexity of $O(n)$.

- The Client software has complexity of $O(n)$: Also here information may be touched several times but the complexity of each use is $O(n)$.
- From the considerations made in this section and before the complexity of the framework can be considered as $O(n * m)$ where n is the amount of software to be checked and m is the number of elements in the vulnerability list. Given that the database is excluded from the framework as the complexity considerations made for building the database are unknown, the complexity can be reduced to $O(n)$.

11.1.4 Computability

The computability can be seen as the ability to calculate a given amount of information within a given time span. Therefore, the framework can again be split up into the three established parts and the database and can be analysed if it is able to conquer the search space of the framework.

The control software checks if the client or server software is running, if the software's machine code has been changed, and, if one of the conditions before is triggered, stores a report in the database. The complexity is $O(n)$ for both actions considering the number of programs to check as n. In a test environment the whole execution of this took about 12 seconds (the database was executed in the same local network, small changes by two to three seconds depending on the processing power) and is, therefore, scalable. From the current point, considering that every additional software takes the same amount of time, the control software can be scaled to about five additional checked targets to maintain an execution time of one minute.

The client software additionally executes a search mechanism on the computer for installed software solutions. The total amount of time for all actions the client software does are as following:

- Only virtual machines used

  Executed on: AMD A8-7410 APU with 2.2 GHz per four cores (only one core used for one virtual machine), 16 GB RAM (only 3252 MB RAM assigned per virtual machine), 15 software packages to message to the database, Windows 7 Home Basic N, about 30 minutes time of execution, equal virtual machines used, deletion, and insertion refers to the times needed to delete or insert information on the database. Because both the Microsoft Windows 10 host machine and the Microsoft Windows 7 guest machine execute background services and update mechanisms the following times are only bottleneck analysis:

  - One virtual machine:

Category	Total time	deletion	insertion
Average	8.3 s	0.4 s	0.7 s
Median	7.6 s	0.4 s	0.4 s
$\sqrt{Var}$	2.5 s	0.06 s	0.9 s
Minimum	6.8 s	0.3 s	0.3 s
Maximum	37.5 s	0.7 s	3.7 s

- Two virtual machines
  - Virtual machine 1

Category	Total time	deletion	insertion
Average	7.5 s	0.4 s	0.3 s
Median	7.2 s	0.4 s	0.3 s
$\sqrt{Var}$	1.8 s	0.3 s	0.09 s
Minimum	6.5 s	0.2 s	0.2 s
Maximum	29.9 s	3.4 s	1.0 s

  - Virtual machine 2

Category	Total time	deletion	insertion
Average	7.5 s	0.4 s	0.3 s
Median	7.2 s	0.4 s	0.3 s
$\sqrt{Var}$	1.8 s	0.3 s	0.09 s
Minimum	6.5 s	0.2 s	0.2 s
Maximum	29.9 s	3.4 s	1.0 s

- Three virtual machines
- Virtual machine 1

Category	Total time	deletion	insertion
Average	8.5 s	0.4 s	0.5 s
Median	8.2 s	0.4 s	0.4 s
$\sqrt{Var}$	2.2 s	0.1 s	0.2 s
Minimum	7.1 s	0.3 s	0.3 s
Maximum	34.5 s	1.5 s	2.0 s

- Virtual machine 2

Category	Total time	deletion	insertion
Average	8.2 s	0.4 s	0.3 s
Median	7.8 s	0.4 s	0.3 s
$\sqrt{Var}$	2.0 s	0.1 s	0.1 s
Minimum	7.0 s	0.2 s	0.2 s
Maximum	34.2 s	2.0 s	1.5

- Virtual machine 3

Category	Total time	deletion	insertion
Average	8.5 s	0.5 s	0.5 s
Median	8.1 s	0.4 s	0.5 s
$\sqrt{Var}$	2.1 s	0.2 s	0.1 s
Minimum	7.3 s	0.3 s	0.3 s
Maximum	33.8 s	1.8 s	1.5 s

The execution of the virtual machines was capped by the CPU as only one core was assigned to each machine. The RAM and the hard disc were

not exhausted and, therefore, have not been recognise as a limiting factor. Also, the database and the network connection have not been identified as a bottleneck because the times used for insertion and deletion have been close to each other. Because of the limiting CPU resources, complexity analysis and a resource usage forecast cannot be given.

- Only hardware based computers

  For this test two machines have been used. One of them was stronger than the other one. These two machines have run for 30 minutes for each test; one test for each machine where each machine was running alone and one test where both machines were running in parallel. The stronger machine was supposed to store 53 packages on the database while the weaker one was supposed to store 52 packages. Both machines have got background services and update mechanisms such as only bottleneck analysis were possible.

  - Two machines running alone
    - Machine 1:

Category	Total time	deletion	insertion
Average	17.5 s	0.4 s	0.3 s
Median	17.5 s	0.4 s	0.3 s
$\sqrt{Var}$	0.9 s	0.07 s	0.07 s
Minimum	15.4 s	0.2 s	0.3 s
Maximum	20.1 s	0.6 s	0.6 s

    - Machine 2:

Category	Total time	deletion	insertion
Average	7.5 s	0.3 s	0.3 s
Median	7.3 s	0.3 s	0.2 s
$\sqrt{Var}$	1.1 s	0.2 s	0.1 s
Minimum	6.6 s	0.1 s	0.2 s
Maximum	21.6 s	3.4 s	0.7 s

  - Two machines running parallel
    - Machine 1

Category	Total time	deletion	insertion
Average	15.9 s	0.4 s	0.4 s
Median	16.0 s	0.4 s	0.3 s
$\sqrt{Var}$	1.9 s	0.3 s	0.2 s
Minimum	12.9 s	0.2 s	0.3 s
Maximum	29.8 s	2.8 s	0.8 s

    - Machine 2

Category	Total time	deletion	insertion
Average	7.8 s	0.3 s	0.2 s
Median	7.7 s	0.3 s	0.2 s
$\sqrt{Var}$	0.9 s	0.1 s	0.02 s
Minimum	7.2 s	0.1 s	0.2 s
Maximum	20.7 s	1.5 s	0.4 s

  Because of the number of tested machines, a forecast about the complexity of the program cannot be given. The values above state that the insertion

and deletion of information on the database took almost as long as it took on the virtual machines and can, therefore, not be seen as a bottleneck (also because the correlation between these values and the total value are moderate[4] (between 0.2 and 0.4) and differ depending on the machine and the test procedure).

The bottleneck on both machines could not be determined completely. On both machines, there has been sufficient RAM to execute the program so memory swapping was not an issue. On both machines, the CPU had both times of waiting and times of running so that the bottleneck may lie in the structure of the program. The hard disc also had times of usage while these times where short and peaks did not lead to a full load.

Given these observations, it can be said that the client side of the framework can be improved. Further research may also lead to a smaller and faster implementation of the framework in total.

To all these tests it has to be said that the tests for the client side have been made on a Microsoft Windows machine for private customers (for virtual machines on both guests and hosts systems). This includes background services and programs as well as update mechanisms which could have been activated during the execution of the client program. These values above, therefore, may be interpreted as a bottleneck analysis because of possible interruptions during the execution.

The server software has been discussed in the subsection before and will not be mentioned again.

11.2 Assumptions

The implementation of the framework contains assumptions about the structure of the network and the behaviour of the underlying systems. This subsection describes these assumptions sorted by their appearance starting with assumptions about the structure and behaviour of the system under observations (including external programs like databases or operating systems) and then going from the server to the client and the control implementation.

11.2.1 System under observation

The system under observation is believed to meet the following requirements:

1. The operating system on machines running the client software is a Microsoft Windows system with version 7 or higher. It may be possible that systems before Microsoft Windows 7 are able to execute the program, but such a system has not been tested because these are not supported anymore from Microsoft. The server implementation was written in Java and was written and was tested to run in Linux environments. The software should, therefore, be able to run on Docker environments, although it was not tested in these environments.

   Apple computers of any version were not tested and are not supposed to run as crucial elements for detecting the operating system are not configured for Apple computers.

2. Because of the implementation as a Java program the computer systems executing parts of the framework are supposed to have a Java Virtual Machine installed. As the software is compiled with a Java Development Kit of version 11, it is supposed to be Java Virtual Machine of version 11 or above.

3. It is assumed that the software (each part of the framework) can connect to a local network. The server implementation additionally is assumed to be able to connect to internet resources. This is because:

   (a) All parts of the framework need a connection to at least one assigned database.

   (b) The server implementation needs a connection to vulnerability sources.

   When these requirements cannot be fulfilled the mentioned software will shut down. This way has been chosen because:

   (a) It cannot be determined if such a connection error is temporary or not. Given it is not temporary (like the database server has been damaged or the computer running the client software was taken to an employee's home) the software would consume network and processing resources.

   (b) The structure of the implemented software remains simple as no testing loops need to be implemented. Such an implementation increases the predictability of the system and stays simple.

4. The system under observation has fixed IP addresses for the database. This is assumed because the framework has not been realised as a Microservice. While Microservices would increase the stability of the framework, they would need either a DNS server (which has not been implemented) or an exploration algorithm which would increase complexity to the code and may make it easier for attackers to mislead the framework.

5. The database has to be set up in a way that it is reachable for the framework and implements the rights and diagrams mentioned in the section 10. For the prototype, a MySQL database has, therefore, been chosen as this database is able to implement these requirements and comes with support for the Java programming language. Additionally, the database is supposed to implement a reasonable amount of security.

   While the implementation of a secure SSL connection between the framework and the database has been dropped (because of the additional work for setting up a certification authority), the database supports logging which is essential for implementing the rights management. As the database is used for computation and comparison, it should support the implemented SQL statements (which are hardcoded in the program and can be analysed from the source code).

6. The prototype calls external programs and makes execution calls to start other programs. These calls can be divided into two subcategories and make the following assumptions about their environment:

(a) When calling a program which is part of the framework, it is assumed that the program is able to start within 10 seconds. This startup time should be used to reach the database so that further errors can be reported directly to the database. Otherwise, the calling program has to make these calls.

(b) External programs are used to test if the program that is checked is running and which software is installed on the computer. Both programs should give back a string of information and should stop afterwards so that permanently running software should not be used. The prototype uses:

i. the "reg" program to read out installed software on Microsoft Windows machines,

ii. the "wmic" program to read out running processes on Microsoft Windows machines,

iii. the "ps" program to read out running processes on Linux machines.

7. Because of settings currently not identified, the connection between the framework and the database has a low bandwidth. This is kept although it increases the time needed to insert data because it shows that the framework can be set up without additional settings on the database (except the shown diagrams).

11.2.2 Assumption on the implementation

This section mentions all considerations made to set up the server, client, and control implementation. Furthermore, all considerations done for the control program are described here because of its dependency with the general considerations.

(a) A logging method has been implemented but turned low. This is made to enable programmers, later on, to change the logger level within the source code to debug the program while programmers for productive environments can turn the logger to log only important messages.

(b) Each programme uses the same location for its configuration files: <workingDirectory> //config. This directory has to exist as the program will not start otherwise and contains the following files:

i. A configuration file which's name is either server.conf, client.conf or control.conf. These values indicate which type of program is running (used when these files need to be loaded. As these files have the same structure, they are loaded the same way via the AbstractCore).

ii. Hash files for the configuration and the execution file of the counter program. These files are used to check if the counter program (the control implementation when started as a server/client implementation or the server/client implementation when started as a control implementation) and the tester's

configuration files have been changed. When an attack or exchange of files occurred, these files differ from the calculated value and the program will raise a warning for attacks.

iii. A file containing information to connect to the database. This file contains the IP address of the database, the port, its username, and its password in cleartext. As stated before, because of the risk of losing this file, the database has been set up with strict right management.

These files were stored separatly to reduce complexity and to make debugging easier. As this is only a prototype to represent the functionality of the framework, this topic has not been considered vital for the demonstration and can be further optimised in later implementations. Also, these files were not encrypted to reduce complexity.

(c) The Java Virtual Machine is able to support SHA-512 hashing.

(d) The waiting time during the execution cycles is currently fixed with 60 seconds. This is enough time to check the program and not too much for the machine. In later versions, this value can be adopted.

(e) The connection to the databases is tested by invoking the function $InetAddress.isReachable$. This method uses ICMP ECHO REQUEST or the TCP Port 7 (Echo) [17] and can lead to wrong results when the server is unreachable because of firewall or server configurations.

The waiting time for this method is set to 10 seconds per request which should be enough and can be adopted for other purposes later.

(f) A class named GUIAdminConnection is kept for debugging reasons.

(g) The SystemConnection is not able to create an instance of itself for Apple computers. This is because the prototype has not been configured for such a computer. This hasn't been done because no such computer has been available for testing the software. Also, operating systems that differ from the description of Linux or Microsoft Windows may throw an exception (e.g. BSD).

(h) The library used to connect the framework to the database allows SSL secured connections between the database and the framework. This possibility has not been implemented because of the additional work needed to build up a certificate authority within the test environments. Therefore, the connections are not encrypted.

When a connection is not encrypted, a warning message is thrown but the execution does not stop. This message can be turned off but it has been considered to not do so because this message should sign that the communication could be made more secure when needed.

(i) Additionally, the server and the client implementation uses SQL statements to insert data into the database that inserts multiple lines. This method introduces errors when at least one of these inserts already exists in the database (and, therefore, inserts no entry at all) but it calculates much faster. Future research can tackle this problem because it is out of scope to implement sophisticated testing methods and the additional time bonus is too big to not implement this method.

(j) The client implementation removes all entries done by itself before it makes inserts. This is done to simplify the source code and the behaviour of the client implementation. Additionally, the described implementation does fulfil the requirements for security as the stored information is always up to date. The performance could be enhanced further, but such optimisation is out of scope.

11.2.3 Server implementation concerning assumptions

This section focuses on the assumptions made for detecting vulnerabilities.

(a) The server implementation uses a hardcoded internet address to contact the source of CVE vulnerability lists. This is done because:

  i. A small improvement can be implemented when the addresses are split and a target year is inserted into the address. This simplifies the source code.

  ii. The implementation of the ability to request from different websites would be out of scope for the prototype as this method would introduce errors and complexity.

(b) The server makes complete updates over the current and the last year whenever needed (see the following point). This is done to initialise the database with an appropriated amount of information while keeping the amount of required space and the probability of missing vulnerabilities small.

  Additionally, the implemented server only allows either the download of current lists (description following) or of year wise lists. Therefore, such an implementation allows initialising databases at the January or February while including the lists of the last year.

(c) The download of the complete lists is not suitable when the database is kept up to date because of resource usage. Therefore, the database will not make complete updates when the time between the last update and now is smaller than one month. If the last update has been made within the last month (this value is stored in the RAM or read out from the database by using the most current time stamp stored there), only recent changes and inserts are requested and updated. This reduces the resource effort needed.

(d) The server assumes the database to be coherent. Changes or missing entries in one or many tables influences the server behaviour because the server does not check the database on consistency. This behaviour has been used to allow many servers to operate on a single database.

(e) The server implementation stores an additional file with an entry for each vulnerability that is already reported to the administrator. This implementation reduces the number of reported files to the administrator and makes debugging and understanding of the prototype easier.

The client implementation does not make any further assumption that hasn't been described before.

11.3 Prototype implementation

This section describes the implementation of the framework as a prototype. It focuses on problems and decisions done during the development.

As stated before the prototype consists of four parts: the client, the control, and the server implementation as well as a shared part. The shared part contains the information used by all other parts plus every connection. Although implementing the shared part into every other part increases the size of the resulting artefacts, the project's structure is simpler, and more suitable to present at the master thesis' audition.

The prototype and, therefore, each part of it, is written in the Java programming language because:

1. The Java programming language is close to UML. Therefore, implemented Java classes look similar to its correspondent UML classes and make the code easier to understand.

2. The Java programming language can implement all necessary functions, classes, and libraries. Required was, among others, access to the command line of the operating system and a connection to the database. Java offered these features, either already built-in or as an external library.

3. C++ was not chosen because of its complexity considering memory management. Higher-level programming languages like Java can manage the memory on its own and reduce the likeliness of errors in total (by simplifying the code and by reducing errors).

4. The author of this thesis has no profound knowledge to start a project of this importance in other higher-level programming languages like C# or Python.

For the database, a MySQL database has been chosen. The author has made this decision because:

1. The database can be installed as a Docker image. Using such an image simplifies the installation of the database.

2. Docker offers several database systems for download. As all these systems can be considered equal, the author decided to use a relational SQL database to implement the ER-Diagrams and because more complex systems did not show additional benefit.

3. Among the remaining database systems, the author chose MySQL because of his experiences with MySQL.

Given that, this prototype uses a MySQL database within a Docker image. For later implementations, other databases could be considered.

The prototype is closely related to the framework and contains only a few additional functions. The classes are set up as described in the section 10 as well as the functions calls. Besides that, additional functionality was implemented to guarantee its suitability for presentation. An example, which has been mentioned before, is the server-sided file which stores the vulnerabilities already messaged to the administrator. Also, the framework treats errors and exceptions in a way to inform the administrator about it when possible. The message types are

1. the list of found vulnerabilities,
2. the SHA-512 hash algorithm could not be initialised (vital),
3. when a computer has stored an invalid report message in the database,
4. when important files cannot be accessed, their values cannot be read or their values were wrong (vital),
5. the hash values of the configuration files do not equal the stored configuration hashes or no stored hashes exist at all,
6. when the hash of the database inserts differ from the stored hash of database inserts,
7. the software to control has not been found (send from all sides: server/client as well as control implementation) or the program could not be started (vital),
8. a database connection threw an exception (vital),
9. an unexpected exception occurred (vital).

Items with the label 'vital' will stop the execution of the prototype. This is done because

1. some exceptions hinder the program from fulfilling its goal,
2. creating recovery strategies from these exceptions are difficult as additional information needs to be stored alongside the program and stored securely,
3. some errors may occur during the daily use of the computer such as starting up the computer without a connection to the internet.

11.4 Used Internet resources

As for the testing reasons the target CVE source is hard coded. This is done to simplify the testing and to guarantee that during the testing the same source will be used. The sources are:

1. `https://nvd.nist.gov/feeds/json/cve/1.0/nvdcve-1.0-<year>.json.gz`
2. `https://nvd.nist.gov/feeds/json/cve/1.0/nvdcve-1.0-modified.gz`
3. `https://nvd.nist.gov/feeds/json/cve/1.0/nvdcve-1.0-recent.json.gz`

The first source is used when no last time stamp has been found, when the time stamp is invalid or when a vulnerability has to be messaged to the administrator: The software then downloads the information of the last two years and insert it into the database. The second and third source are update sources recommend to update the database and used when the database is not older than one day.

During the development, internet resources have been used to develop the prototype. The following resources have been used to develop the program (last visited 22.02.2018):

1. https://stackoverflow.com/questions/26637168/how-to-convert-a-date-to-milliseconds to implement the date conversion between the CVE format and the time in miliseconds
2. https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html and https://stackoverflow.com/questions/696782/retrieve-column-names-from-java-sql-resultset to implement iterations over the results of the JDBC database statement
3. https://theitbros.com/how-to-get-list-of-installed-programs-in-windows-10/andhttps://social.msdn.microsoft.com/forums/en-US/94c2f14d-c45e-4b55-9ba0-eb091bac1035/c-get-installed-programs describe the used methods to gain the names and versions of the software installed on Microsoft Windows machines.
4. https://docs.docker.com/
5. https://docs.docker.com/get-started/
6. https://github.com/docker/labs/tree/master/developer-tools/java/
7. https://docs.oracle.com
8. http://goinbigdata.com/docker-run-vs-cmd-vs-entrypoint/
9. https://exceptionshub.com/how-can-i-create-an-executable-jar-with-dependencies-using-maven.html
10. https://mvnrepository.com/
11. https://www.w3schools.com/
12. https://nvd.nist.gov
13. https://docs.oracle.com/javase/7/docs/api/
14. http://taxiiproject.github.io/
15. http://hailataxii.com
16. https://api.xforce.ibmcloud.com/doc
17. https://otx.alienvault.com
18. https://github.com/TAXIIProject/java-taxii
19. https://dev.mysql.com/doc/refman/5.7/en/select-optimisation.html
20. https://stackoverflow.com/questions/6811522/changing-the-working-directory-of-command-from-java
21. https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlocks-handling.html
22. https://stackoverflow.com/questions/26761436/catch-duplicate-key-insert-exception

12 Conclusion

In this thesis, a simple concept for vulnerability detection has been developed, evolved, described, and tested.

This thesis started by analysing the current field of cybersecurity. Literature concerning threats, software vulnerabilities, and threat handling have been evaluated to identify current practices in the field of cybersecurity. It has been shown that the environment of Microsoft Windows has not yet been explored from a scientific point of view and that security-related publications in this field are rare.

Based on this analysis, the current challenges regarding such software have been mentioned. This broad spectrum of analysis has tried to give an insight into the complexity, the dimensionality of cybersecurity, and possible solutions like as few user interactions as possible, complexity reduction, usage of well known third party libraries, and security as part of the architectural process.

Given these insights and an analysis of existing solutions in both scientific publications and existing solutions, the aims of the framework have been defined. The framework should support administrators to find vulnerabilities within their computer network and should report it to the administrator. The information should be stored in a way that attackers may not be able to retrieve, encrypt, decrypt or manipulate this information.

From that point on the framework has been evolved. Starting from possible solutions that would meet the required criteria, the ideas were adopted step by step to fulfil the requirements of security. The final version is presented in the section 10 using the UML language to give a broad and standardised view of the framework. Because of the fact that the database was more powerful when features have been stored double, database normalisaion has been ignored.

Following the definition of the framework, the implementation as a prototype was made to proof the suitability of the framework. There, additional adaptions like the change in the database structure were discovered and were introduced into the framework. When the prototype was finished, it has been tested in real-world environments. No optimisations were made to the prototype with respect to the framework except the change in the database table structure. The results are listed in the section 11 and identify probable bottlenecks at the database and the structure of the prototype.

As stated above it turned out that a perfect security system is not possible (given that additional abilities like networking and internet access have to be guaranteed). During the research for the models presented above (and especially during the research at [9]), the author came to the conclusion that almost every connection and communication line can be attacked (or at least blocked) by man-in-the-middle-attacks. While this is no new information, it turned out that such attacks are, especially between the operating system and the client program, hardly avoidable. Therefore, another strategy has been chosen: Although programmers are not able to prevent attacks on software, they can count on the mass of information that is produced by all computers at the same time. An attacker may attack a handful of computers but will hardly be able to compromise all observed computers at the same time.

In addition to to this the meaning of the framework has become a question during the creation of this master thesis. "A framework is a class model, together with a free role type set, a built-on class set, and an extension-point class

set.“[21]. This definition is the most precise definition of a framework found during the research, but it does not state at which point the class is extendable or at which point static classes can be inserted into the framework. Therefore, the connection classes are chosen to be extendable as those classes manage the information flow within the framework. While this strategy does not give the freedom to manipulate the given program, it gives the freedom to adapt it to the execution environment.

Microsoft environments allow the user to view both running processes and installed software. This may give comfort as those values can be acquired without administrator rights but may also introduce security risks: Programs in user mode may observe computers for possible vulnerabilities. This design should be reconsidered and may be analysed further in the future.

Third party software may show interesting behaviour. For example, the database has turned out to be much faster and only slightly larger when storing pre-calculated values than calculating them on the fly. Such a behaviour perverts database normalisation as normalised relational databases need more computation time. Years ago, the author met a database administrator that did not care for database normalisation (and produced database tables that looked that way), he now understands one of the reasons and should be more carefully to listen to administrators about the hows and whys.

12.1 Comparison of the framework with the literature

The developed framework is compared with the analysed literature in this section. In this section, the following questions will be answered: Where is agreement or disagreement between the developed framework and the analysed literature and what foundations for further research does the developed framework lay.

Nguyen et al. [16] have described a method to detect insider attacks via system calls. This method has not been considered for implementation because of the effort and the target of the research: It aimed at identifying attacks in computer environments and purposed access to the system log of the machines. The purposed framework of this master thesis instead aimed at identifying vulnerabilities in Microsoft environments instead of attacks and did not need access to infrastructure on the administrator level. Additionally, a detection mechanism would have to be implemented. Such a detection mechanism can lead to wrong assumptions about the environment when the error rate is too high.

The AIS system, as described in [18], and its technology of TAXII messages were considered to be implemented. During the research for this system, the following reasons showed up why this system has not been integrated:

1. Initially, for testing reasons, this system was supposed to take part in the AIS program. Further requests to the embassy of the United States of America have not been answered. Because of the missing answers, the framework does not explicitly support information retrieval from the AIS program.

2. Although AIS was unavailable because of the mentioned reasons, the TAXII standard is an appropriated method for vulnerability sharing. But all found TAXII services presented messages on vulnerabilities which pointed to entries in the CVE database.

Therefore, TAXII support was dropped as CVE support needed to be implemented anyway. Given these reasons, CVE support only has been implemented.

Zhang et al. [25] described a way to predict software vulnerabilities by using the national vulnerability database. This approach has been dropped because of the error rate the method introduced. On the other hand, the thesis of Zhang et al. has been used to draw borderlines for the developed framework: It should run stable, have a small error rate, support the administrator in his daily work, and show current problems (instead of future ones based on prediction).

The work of James F. Kurose et al. [12] and Tanenbaum et al. [28] introduced possible attack scenarios from the network side. The framework used this information to focus on secure networking during development. Kurose et al. showed the importance of encrypted communication to secure the developed framework.

Also, the work of Andrew S. Tanenbaum [27] has influenced the development of the framework and have been implemented. Tanenbaum's work on operating systems described strategies implemented in operating systems. Although most of these strategies highlighted methods which could not be used and could not be hardened (because they were implemented deep in the operating system), is also showed mechanisms like user management which have been recommended for the implementation of the developed framework.

At this point, it should be stated that the used Microsoft Windows Home system gave access to the lists of installed software without administrator rights. While this information can be considered critical for further attacks, it is recommended to secure Microsoft Windows against such behaviour.

As described before, OpenVAS inspired the developed framework in many ways. The following list describes which points were taken from OpenVAS and which were altered:

1. Support for administrators by delivering information: Both, OpenVAS and the developed framework, aimed to support administrators. The main idea in both solutions is that the administrator has to solve the problem and not the program. This method was taken from OpenVAS because the administrator knows the company's structure and goals. With this knowledge, the administrator knows when to update/upgrade a system and to which version to upgrade.

2. Standalone environment: OpenVAS is shipped with its own operating system. The developed framework does not explicitly recommend a solution of any kind at this point. The prototype was developed with the Java programming language. This makes it possible to ship the server implementation of the prototype in Docker environments later when needed. The client implementation is not able to run within Docker. Further research needs to check if the whole framework can run in a Docker environment or in its own operating system.

3. CVE as vulnerability source: The developed framework does not explicitly state where the information for vulnerabilities has to come from. Both OpenVAS and the prototype use the CVE database as a source for vulnerability information.

4. Information retrieval: OpenVAS is a network-based vulnerability scanner. The developed framework instead is host-based because of software which

is not connected to the network: OpenVAS is not able to determine if this software is vulnerable or not. Therefore, the developed framework supports a host-based approach.

5. Automation: OpenVAS can be used by either setting up a task scanning for vulnerabilities on a regular base or by starting tasks manually. The developed framework scans only for vulnerabilities in fixed intervals because of simplicity.

6. User interface: OpenVAS introduces a graphics user interface for both the administrator and the user. The developed framework does not provide any user interface except configuration files. A graphical user interface was dropped because it does not impact the framework's functionality.

12.2 Which co-contributions has been laid

This thesis lays the foundation in several different areas:

1. User management: What is the user allowed to do on a computer? The user should be able to do his job, but does he need access to a list of all processes running on that computer? Does he need access to a list of all software solutions installed on that computer? On Microsoft Windows Home, a user can access a list of all installed software packages. On Linux, the ps aux instruction shows all running processes. This information can become critical.

2. Microsoft environment: Explicit information on security in Microsoft environments was hard to obtain. Because of the variety of security solutions and the Microsoft Windows Defender, it occurs that these data is valuable to companies. Further research is needed to determine if this was just a bad research by the author or if security in Microsoft environments is proprietary.

3. Multi-system-capability: Another target should be to prove that this framework can support networks with diverse operating systems. Modern computer networks are hardly homogeneous. Therefore, the developed framework must be able to work together with Linux servers, Microsoft Windows clients, and Apple clients. Such checking requires resources and may alter the final results.

4. Information retrieval: The way information is received and processed can be optimised. The current information retrieval was designed to proof the framework. Further processing into this section can speed up the process, secure the transport of information and return more accurate results.

References

[1] Vulnerability detection based on aggregated primitives.

[2] Danny Bradbury. Microsoft's new window on security. 2006.

[3] Steve Burbeck. The tao of e-business services. 01 2000.

[4] Jacob Cohen. A power primer. *Psychological Bulletin [PsycARTICLES]*, July.

[5] Peter J. Denning and Peter J. Denning. Great principles of computing. *Communications of the ACM*.

[6] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[7] Greenbone Networks GmbH. Greenbone networks, October 2018.

[8] Greenbone Networks GmbH. Openvas open vulnerability assessment system, October 2018.

[9] E. Hermann. A security policy model for agent based service-oriented architectures.

[10] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.

[11] Kurt Bittner Ivar Jacobson, Ian Spence. *USE-CASE 2.0*. Ivar Jacobson International, December 2011.

[12] Keith W. Ross James F. Kurose. *Computernetzwerke*, volume 5thd edition. Pearson Education, 2012.

[13] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[14] C. Lesjak, D. Hein, M. Hofmann, M. Maritsch, A. Aldrian, P. Priller, T. Ebner, T. Ruprechter, and G. Pregartner. Securing smart maintenance services: Hardware-security and tls for mqtt. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1243–1250, July 2015.

[15] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[16] Geoffrey H. Kuenning Nam Nguyen, Peter Reiher. Detecting insider threats by monitoring system call activity. 2001.

[17] ORACLE. Class inetaddress, April 2019.

[18] Andy Ozment. *Privacy Impact Assessment for the Automated Indicator Sharing(AIS)*, 2015.

[19] Jon Postel. User datagram protocol. 1980.

[20] Jon Postel. Transmission control protocol. 1981.

[21] Dirk Riehle. Framework design: A role modeling approach. 2000.

[22] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, Nov 2011.

[23] DENUVO Software Solutions. Denuvo, October 2018.

[24] Doina Caragea Su Zhang and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. *DEXA 2011 Part 1*, pages 217–231, 2011.

[25] Xinming Ou Su Zhang, Doina Caragea. An empirical study on using the national vulnerability database to predict software vulnerabilities. 2011.

[26] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, Dec 2014.

[27] Andrew S. Tanenbaum. *Moderne Betriebssysteme*, volume 3rd edition. Pearson Education, 2009.

[28] Steen Maarten van Tanenbaum Andrew S. *Distributed systems: principles and paradigms*. Pearson, 2014.

[29] Herbert H. Thompson and Scott G. Chase. *The Software Vulnerability Guide.*, volume 1st ed. Course PTR, 2005.