WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

Seminar Paper

Security Concerns in Proprietary and Open Source Software

Julian Drexler

Student ID: 52007920

Subject Area: Information Systems & Society
Supervisor: Dr. Rony G. Flatscher

Date of Submission: 20.12.2023

Department of Information Systems & Operations Management, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR WIRTSCHAFTSINFORMATIK UND OPERATIONS MANAGEMENT DEPARTMENT OF INFORMATION SYSTEMS AND OPERATIONS MANAGEMENT

Contents

List of Figures

Abstract

In our highly connected digital world, new security vulnerabilities are found each day which makes the topic of software security highly relevant. This seminar paper gives an overview and comparison of different security aspects and concerns with a special focus on open source and proprietary software. Therefore, general software security topics like the development process and factors which can improve security are elaborated in this paper. In order to provide a broad perspective and contextualization, the paper draws on various surveys of relevant companies and foundations in the field of software security, as well a state-of-the-art papers which provide an overview of this topic. The comparison shows that both open source and proprietary software have similar general security approaches but differ in the potential risks. In conclusion, it was found that open source and proprietary software can only be looked at as being separated to a certain extent since there are various dependencies between the two through embedded open source code within proprietary software. As a result, there is no clear answer which of those concepts is more secure because of the many different aspects regarding software development, maintenance and organizational structures.

1 Introduction

Software security is a crucial and part in the digital and connected world we live in. Therefore security problems can have a global impact and can affect everyone. There are some examples, which had these large scale effects on the world which will be discussed in the parts examples of security issues in open source and examples of security issues in proprietary software. Every year there are many new security vulnerabilities discovered in different software and only in 2022 around 25.082 were made public (SecurityScorecard, 2023). This large number of found vulnerabilities show, that this topic is crucial and that companies as well as individuals should be aware of it. The market spending for digital security and risk management will according to Laurence Goasduff (n.d.) grow around 14% from 2023 to 2024. Nevertheless, companies and individuals choose between open source and proprietary software and decide where to put their trust. In a recent report by Gordon Haff (n.d.) describes that in enterprise solutions proprietary systems have a larger market share compared to open source, but the report suggests that this will change to an even market share.
This seminar paper provides a general overview of software security, covering both open source and proprietary software. The first chapter discusses general concepts of software security, while the second chapter focuses on open source software, including its organizational structure, funding, maintenance, and security issues. The following chapter discusses proprietary software, including licensing models, maintenance, open source components, and security issues. Finally, a comparison is made between proprietary and open source software in terms of security.

2 Software and Security

Software security is a topic with a high relevance as already described in the Introduction. Therefore it is important, that software is developed and deployed with security in mind. But this is not trivial in anyway because the complexity and different influences in the process. Therefore this chapter describes first the terminology and definitions, which will provide an overview of the important terms. Next the security goals are discussed which a system/software should try to achieve. Because of this, the next part focusses on how these goals can be achieved in the development process and relevant practices with guidelines. In addition, this section also elaborates on the methods which can be employed to test the security of software. Lastly, warranty and liability aspects in licenses are discussed with two examples of the most popular operating systems in comparison to the most used open source licences.

2.1 Terminology Definitions

This seminar paper uses technical terminology. Thus, in order to ensure clarity and a common understanding of the terms, this part will define the important terms. The following definitions will be understood as general as possible and will have the below defined meaning in this paper.

Source Code: Describes the instructions written in a programming language, which can be compiled and modified.

Repository: Describes where the source code, documentation or other relevant data for a software project is stored. Mostly this is on a platform with a version control software like "Git".

Software License: This is a legally binding document that outlines the terms of use for the source code. The original author applies the software license to the source code.

IT Security: This consists of several aspects: The first aspect is Safety. This means that the software doesn't get into unauthorised states. The next aspect is Security, which means that the software or the user doesn't get unauthorised permissions to get or change information. The last two aspects are Protection (no unauthorised access to system resources, such as data) and Privacy (security measures to comply with applicable laws, such as GDPR). (C. Eckert, 2018, p.6)

Software Security: Application focused discovery of risk and vulnerabilities (Khan et al., 2022)

Dependability: The software is so stable and reliable that it doesn't get into an unacceptable state due to errors, such as programming errors. (C. Eckert, 2018, p. 6-7)

.

Vulnerability: A weak point of the software or system which can be exploited with the objective to cause damage (Sommerville & Sommerville, 2012, p. 350). Additionally a Zero Day Vulnerability describes a weak point which is unknown for the involved parties, which own the software or are responsible for mitigating and patching the problem.

Attack: Exploitation of weakness in a system with the goal to cause damage. (Sommerville & Sommerville, 2012, p. 350)

2.2 Security Goals

C. Eckert (2018) describes the various security objectives for the relationship between a subject and the information/data (object). Since she offers such a comprehensive overview, this paper will primarily reference his described objectives.

The first objective is "authenticity" and describes the correctness of subject and object. Both of them should be correct and therefore should be able to authenticate themself. In this context, the subject can authenticate through i.e. a password and the object through a proof of origin.

Secondly, the author mentions "integrity" which means that the subject should not be able to access or manipulate an object without the permission that has been granted. This also includes that any manipulation should be recognized by the system and that the possibility of manipulation should be prevented. The prevention tactic could be through cryptographic algorithms and hashing.

The author continuos with the objective of "confidentiality" which can be described as no possibility of unauthorized access and collection of information by any subject. Therefore, a software/system has to control the information flows in a way that only the correct subject gets the information it is supposed to get. This can be achieved through encrypting and decrypting the information. In addition to encryption, labelling of sensitive objects can also be applied to reach this objective.

The next goal is "availability", which means authorized subjects can interact in their role without any restriction. This includes sanctioning and limiting the resources of not authorized subjects. Actions to reach this objective would be to prioritize subjects and resources in a system.

The author describes the next security goals as "non repudiation" which can summarized as Assign-ability and traceability of user actions. With this goal, accountability should be provided which is significant in any electronic business. For this the subject actions should be logged and be auditable.

The last security objective, which the author describes is "anonymisation" and "pseudo anonymization" - in short "privacy". This can be summarized in one sentence as the alteration of personal data in such a way that the original personal data is no longer recoverable (anonymization) or the data is in no way assignable to an individual (pseudo anonymization). To achieve this goal, one possible approach would be through anonymization service providers, which shields the data from unauthorized third parties and provides it to authorized entities. There is one problematic aspect to this, which is that the subject mostly doesn't have any influence on the process.

The above security objectives should be considered in the design and development of secure software. Not all of these objectives can be achieved to the same degree and there are possible conflicts when achieving those security goals. To illustrate an example what a possible conflict could be, let's compare the goals of "privacy" and "non repudiation". For "privacy", one goal is to anonymize the subjects data and even reduce data. In contrast to that goal stands the "non repudiation" goal, where the information about the subject should be collected for audit reasons. As can be seen, these two goals are conflicting with each other regarding how each objective should be achieved and to what extent, inevitably leading to some sort of trade-off between those two goals.

2.3 Software Security in Development

There are no universal solutions that apply to all projects, but there are various practices that are deemed "good practice" and can enhance security. Furthermore, software development is in the most cases not a linear undertaking from start to a pre-defined goal, but much rather an iterative and agile process where the software evolves over time (Akbar et al., 2018).

According to Sommerville and Sommerville (2012) there are three phases for the process of considering security in development: A rough risk analysis, risk analysis in the development process itself, and analysis of risks during operation. The first phase is a broad perspective to define requirements for the system as a whole. The second phase develops concepts for security implementations and how to achieve the security goals in the software. The final phase assesses the risks after the deployment of the software. It is also important where the software or system will be deployed because the requirements will differ in terms of security and availability. For example, a consumer software for text editing has lower security requirements than a software for medical devices. Thus, there should always be an individual assessment of the context software will be used in and the risks of failure due to a lack of security or other issues.

The next two subsections take a closer look at which processes can be implemented for reliable and secure software. In addition to that, different concepts and tools which can be applied to prevent security pitfalls will be discussed. It should be already pointed out that regardless of effort and precautions taken, no software is perfectly secure and it is always a process to achieve secure software.

2.3.1 Processes in Development

Sommerville and Sommerville (2012, p. 392-394) describes that it is important for the development processes that it is standardised and documented. As a result, the development process can be audited and repeatedly verified. The result is a robust and verifiable process. Through standardization and documentation of a development process, software errors and security problems are less likely to occur. Furthermore, the following two methods increase the overall quality, reliability and security of software:

- Peer reviews: At least one other person checks the code for errors or possible security problems.
- Software testing: These are pre-defined tests to check the functionality of the software regarding its correct behaviour.

Next, the software development life cycle (SDLC) will be introduced. The typical phases in the SDLC include the assessment of requirements, design, coding, testing, deployment, and maintenance. All of these phases are important and security aspects should be taken into consideration during every single one (Khan et al., 2022).

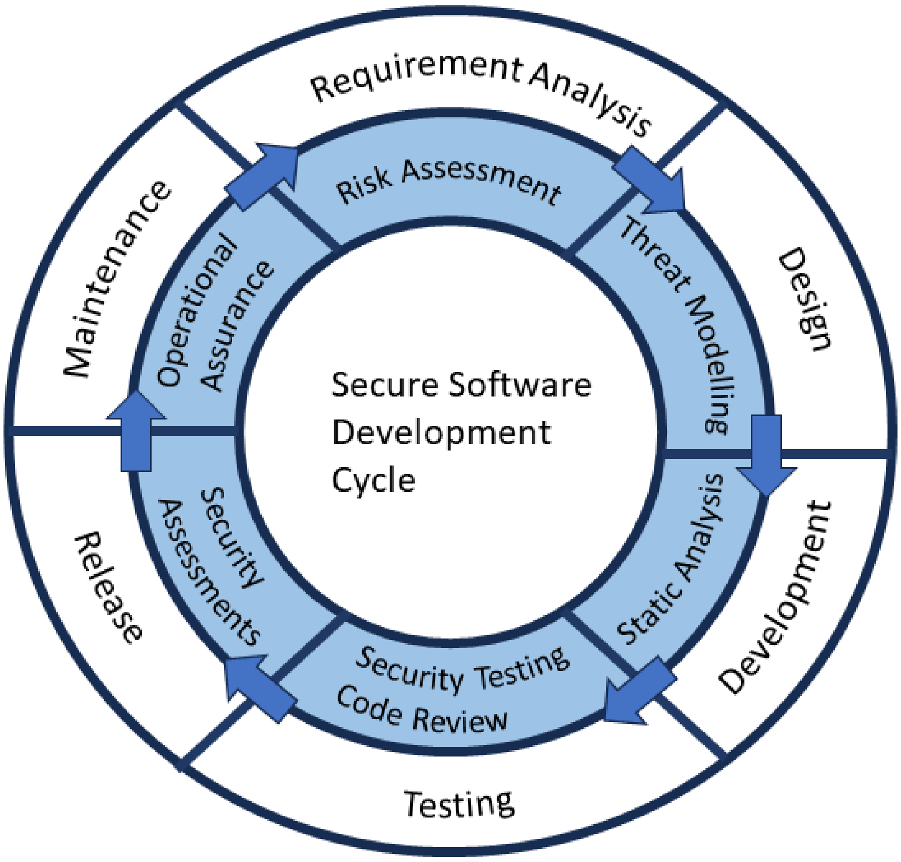


Figure 1: Secure Software Development Life Cycle (own representation)

Let's have a closer look at the SSDLC figure above and what the implications are in each of these phases. According to Khan et al. (2022), the requirement phase is crucial and stands with 97,5% on top of identified security risks in literature. The author argues that the reasons behind this are insufficient identification, specification, and documentation of security requirements. In addition to that, non-functional security requirements should be more focussed on. The design phase uses threat modelling and design reviews (Digital Maelstrom, n.d.). The author describes in this phase, that pitfalls lie in a lack of developing threat models. This risk can be mitigated through threat prioritization based on the potential impact. In the development phase, the author describes, that mostly a lack of awareness by the developer can lead to security risks. In this phase, possible actions like the already mentioned code reviews, security training and the implementation of static code analyses have proven to be beneficial. In the testing phase, the author argues that the detection of security risks by employing security testing techniques is of high importance. The risk in this phase is the lack of employing these testing techniques and test cases. The author identifies in the deployment phase the risks of setting the software up in a new environment and therefore resulting in new security risks. These can be mitigated by identifying vulnerabilities and creating updates for fixing these problems. The last phase, maintenance, is similar to the previous stage. Therefore the author describes, that in this phase, the software should be monitored for new security risks in order to react accordingly.

Another aspect is how the development is done and which models are used. Therefore, Akbar et al. (2018) describes the most used and well known models in software development which implement the SDLC phases differently. The first one is the waterfall model, which has a more static structure. In this model, the requirements are collected and the software project gets completely planned. It finishes when the project is delivered to the customer and only then the next iteration of the software starts. As an agile approach, the author describes the "Extreme Programming Model" (XP). This model can be described by short development cycles, constant feedback and incremental arranging. There are more models such as the V-Model, Scrum or RAD, which the author describes, but these are out of scope for this paper.

Another process, which can be implemented are automated vulnerability scanners. Such tools examine the source code to identify security weaknesses which can result in security issues later (NIST, 2021). Such code scanners can also be integrated in the development platform. For example, GitLab offers multiple security analysis tools on their platform where source code of the project is hosted (GitLab B.V., n.d.).

Another important issue in this context is compliance, where applicable, as not all software products need to be compliant in the same way. Nevertheless, it should be considered to carry out process and software security audits to avoid financial risks and security threats Herath and Herath, 2014. In general, software security audits are a crucial part in the software lifecycle.

Furthermore, there are different standards and processes which can be implemented like ISO-27k series or COBIT. These tools help security compliance and governance (Prapenan & Pamuji, 2020). These standardized processes are not further elaborated in this seminar paper but are nevertheless important to mention.

To conclude this section, it is important that the right processes are implemented in the whole software life cycle. If this is not the case, it adds more complexity and costs to the software project (Khan et al., 2022).

2.3.2 Security Testing Methods

In this chapter, the broadly used security testing methods are described. These methods help to evaluate and increase security in software. The following security testing methods are described according to Cruz et al. (2023).

The first category are the Static Code Analysis (SAST), tools. With this approach, the source code is analysed and possible security risks are reported. Depending on the tool, these methods can range from syntax analysis to the detection of included packages that do not meet the necessary security requirements. The second approach is through the perspective of the attacker, which is called "Dynamic Application Security Testing" (DAST). The author distinguishes this approach from SAST by stating that the DAST approach is a black box approach, whereas SAST is a white box testing method. If those two approaches are used together, this hybrid approach is referred ot as "Interactive Application Security Testing" (IAST). The author describes two more important approaches, which are "Mobile Application Security Testing" (MAST) and "Software Composition Analysis" (SCA). With the MAST approach, mobile devices are tested utilizing SAST and DAST methods. Lastly with the SCA approach, third party dependencies are scanned. This means, that this approach tries to analyse the complete supply chain of third party packages and software for vulnerabilities.

In addition to the methods described above, there are also other security improving methods which are widely used. The first one is "Pen testing" or "penetration testing". This is a method, where an attack is simulated to find and exploit vulnerabilities. (Cloudflare, Inc, n.d.). However, this is mostly offered by specialized security companies. The second method to mention is "Fuzz Testing" or "Fuzzing". With this method, invalid or unexpected inputs are sent in order to test not only security, but also reliability (Synopsys, Inc., 2017). Fuzzing is a practical method, which belongs to the tools of Dynamic Application Security Testing (DAST) (Cruz et al., 2023).

2.3.3 Guidelines for Security in Software Development

Security guidelines help to improve software. They also help to create a more standardised approach to software development. The security aspects should be already considered at the stage of planning the software architecture. For this, Sommerville and Sommerville (2012) suggest ten security guidelines in the design stage of a software:

1. Established security-policies: Already applied security policies should be recognized and implemented.
2. Fault tolerance: The system should not have a total breakdown if an error occurs.
3. Failure avoidance: The system should be stable and if it fails, there should not be any data leakages.
4. Balance of security and usability: Too many security functions will have a negative effect on the usability.
5. User-action protocols: The system should create protocols of the user action to restore a state if the system failed, to find errors and to prevent insider attacks.
6. Redundancy and diversity: To have multiple versions of the software and implement different technologies for various platforms.
7. Validate input: The system should only accept expected inputs, therefore the input must be limited.
8. Diversify risks: There should not be access rights that grant the user to access "everything" or "nothing. User should only have the possibility to access functions or data in their defined scope.
9. Simple setup: Risks of errors when setting up the system should be reduced.
10. Recoverability: The system should be recoverable to a stable state.

To achieve these goals, "Secure Software Coding" (SSC) can be used. The objective of SSC itself is to minimize risks of security vulnerabilities in the code. Therefore, key elements of SSC are that security features need to be adapted depending on the project, and that different programming languages pose different risks (Humayun et al., 2023). To exemplify this aspect, a closer look at the programming languages C and Rust will be taken. With code written in C, memory safety has to be considered and therefore the code must be written accordingly to ensure correct memory handling (Kalin, n.d.). In contrast, Rust is a memory safe programming language by default (Rina Diane Caballar, 2023)). There are more SSC guidelines which need to be considered and are among others defined by the OWASP Foundation[1]. In this seminar paper, further elaborations of the guidelines are out of scope.

Although it is important to mention, that applying secure software coding, it should be implemented based on three important factors which are people, processes and tooling. In summary, this means that all stakeholders should be included and that it should be ensured that the tools support the new processes, which can be for example the Secure Software Development Cycle (Synk Ltd., n.d.).

2.4 Warranty and Liability

Warranty and liability are complicated topics due to the complexity of how software is developed, distributed and maintained. This section is divided into three subsections in which the aspects of warranty and liability are discussed. The first subsection takes a closer look at the license terms of two widely used operating systems. The second section takes a look the licensing terms of the most popular open-source licenses. The last subsection discusses the EU regulations in the field of software.

2.4.1 Popular Operating Systems License Terms

The most popular OS at writing this, are with around 68% Windows and around 20% OS X (MacOS)(StatCounter, n.d.-a) Therefore a comparison of the licensing terms between the newest versions of each OS Microsoft with Windows 11 and Apple with MacOS Sonoma. Note that these two are both proprietary operating systems and used by individuals and businesses.

The Windows 11 license terms state, that the software is provided without any warranty, except if locally the law allows that, the device manufacturer, software installer or Microsoft can be held accountable. As for the liability part, the notice states, that there will be no compensation for any damages (Microsoft Inc., 2021).

Similar to the Windows 11 license terms, the MacOS Sonoma states that they don't provide any warranty and the user uses the software on their entire own risk (APPLE INC., n.d.).

These two examples of the most popular operating systems show that both have a limited warranty and they are not liable if damages occur through the systems. This is an interesting finding, because for Windows 11, which is also a paid software, there exists

[1]https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/01-introduction/05-introduction

warranty for damages. However, it seems like a very complicated endeavour to make any claims regarding liability due to the many factors, like installed software.

2.4.2 Open Source License Terms

For the open source licenses, the three most relevant were selected for this paper which are the Apache 2.0 with 30% market share, MIT with 26% market share and GPLv3 with 9% ("Open Source Licenses in 2022," n.d.). Figure 2 shows the most popular open source licenses world wide. This is only an overview of the used licenses and they will get more discussed in the section Open Source Licensing.

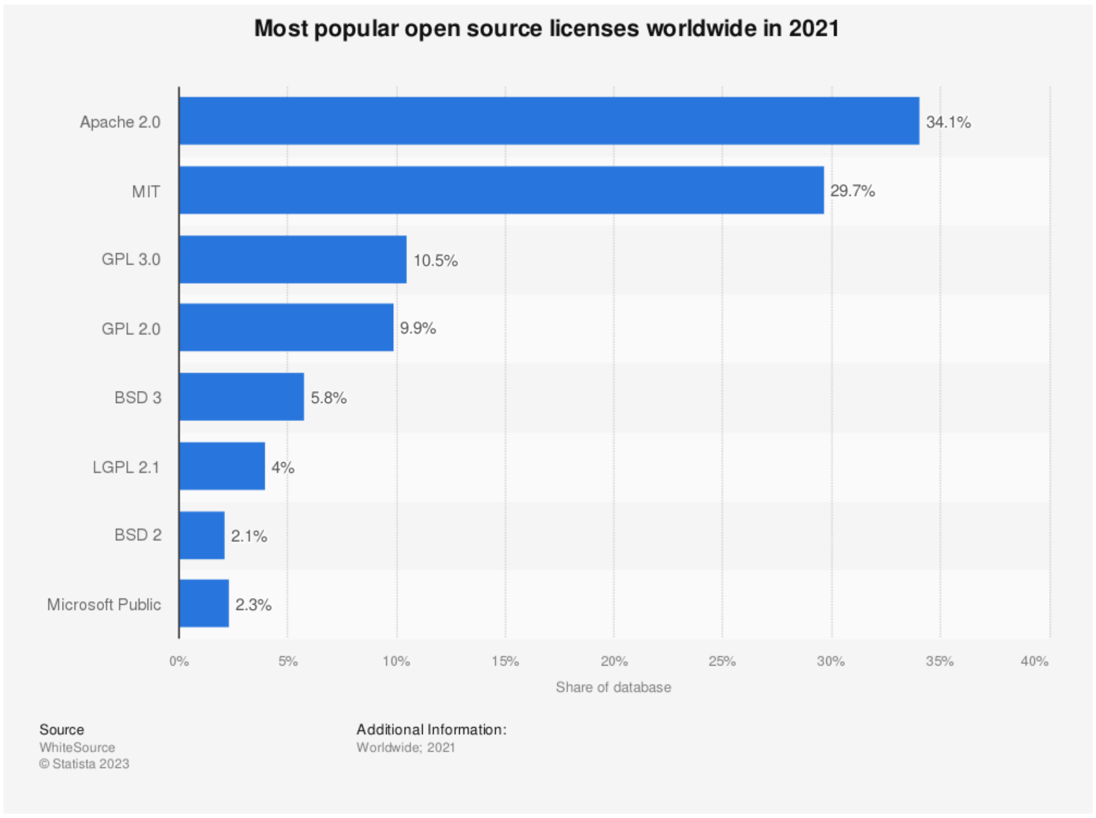


Figure 2: Market share of open source licenses WhiteSource, 2022

In the section, the warranty and liability parts of the Apache 2.0 license[2] are described. These sections state that the software which is distributed with this license doesn't provide any warranty or liability ("Apache License 2.0," n.d.).

The MIT license [3] is the shortest in of these three licenses. It also briefly states that no warranty or liability is provided (MIT, n.d.).

[2]https://www.apache.org/licenses/LICENSE-2.0.html
[3]https://mit-license.org

Lastly, the GPLv3 license [4] is more comprehensive than the other two licenses. At section 15 and 16 it states that no warranty or liability is provided.

To summarize the warranty and liability provisions of these three widely used open source licences, none of them gives the user any warranty or liability rights from the software developer or distributor. This is also important since many open source projects are created by individuals or groups. This means that these licences protect the community driven projects by excluding any warranty or liability. In a comparison by GitHub Inc. (n.d.) of 45 open source licences, it was found that they do not grant or explicitly state any liability or warranty rights.

2.4.3 EU Regulations

According to European Commission. Directorate General for Communications Networks, Content and Technology. (2021) for a product the liable party is the producer, which means the manufacturer of the finished product. By this definition, the problem arises that it might not be possible to identify a liable party in the EU when software was downloaded. Also, the producer must meet safety expectations when bringing the product into circulation. The producers are not liable if they can prove that the software was not deficient when it was brought to the market. If a customer wants to report a defect on their purchased software and make product liability claims, this can be difficult since it can be challenging for a non-expert in this field to assess the situation correctly. Another point to add is that the software life cycle patches vulnerabilities and updates the software itself. This can also lead to a new risk profile of the software.
Developments in regulations brought the new PLD (Product Liability Directive), which holds the producer more accountable for their produced software and therefore helps the consumers (European Parliament, Council of the European Union, 2019)

3 Open Source Software

Open source software is a crucial part of the digital world and nearly everyone who browses the web has already encountered open source software. One reason for this is that the most used web servers are Ngnix with 33,9% and Apache with 32,8% in January 2023(Q-Success, n.d.). Also, this is not only limited to web servers. The most used mobile OS Android with a market share of 73,2% (StatCounter, n.d.-b) is open source as well. To clarify, open source does not mean that it cannot be commercial, it has an essential role in our digital world and economy.

[4] https://www.gnu.org/licenses/gpl-3.0.en.html

At this point, the question arises how open source software can be accessed. This depends on the project since there a different ways to share the source code. One of the most used methods is to share it on a version control platform like GitHub [5]. Other methods are downloading the source code directly from the author website, possibly from another medium or upon request. Therefore open source provides transparency which can lead to a increased trust in the software which is in proprietary software not in the same way possible.

This chapter should give a compact overview of the complex topic "open source". It contains definitions, history, licensing, maintenance, organization, funding and security. The last subchapter gives examples of two security vulnerabilities of open source software which, had a global impact.

3.1 Definitions & Criteria, History and Licensing

When it comes to open source software, it is important to note that it has developed over time and that it has various different aspects. Therefore, this chapter is split up into three sub chapters to take a closer look at different definitions and criteria of open source software. The next subchapter focuses on the history, which describes how open source evolved over the time starting in the 1950s. The last subchapter explores licensing, which is one of the most important topics when it comes to software. Here, the key components of the top 5 most used licenses are listed and categorized into three levels of copyleft.

3.1.1 Open Source Definitions and Criteria

The definition of Open Source according to open source initiative (2006) consists of the access to the source code and the distribution terms of the licenses.
First, lets declare what open source in the context of software is and what criteria it has. There are different definitions and the criteria were subject to change over time and therefore, this seminar paper focusses on one of a general declaration by the "open source initiative"[6] which is affiliated with well established open source organizations like the "Linux Foundation" or the "Apache Software Foundation".

The open source initiative (2006) ten features an open source software should have are:

1. Free Redistribution - No restriction and fees in distribution of the software
2. Source Code - The source code must be accessible and modifiable

[5]https://github.com/
[6]https://opensource.org/

3. Derived Works - It should be allowed to modify the code and distribute it under the same license
4. Integrity of The Author's Source Code - It must be clear from the license how modified versions are distributed
5. No Discrimination Against Persons or Groups
6. No Discrimination Against Fields of Endeavour - The license should not restrict a field, for example in a business or university setting, where the software is used
7. Distribution of License - The license applies to the software and no additional licenses should be needed for redistribution
8. License Must Not Be Specific to a Product - If open source code is incorporated in a software, all parties must have equal rights as specified in the license according to the used code
9. License Must Not Restrict Other Software - It should not restrict other licensed software when being distributed alongside it
10. License Must Be Technology-Neutral - The licenses should not be restricted to one technology.

It is also important to emphasize again that different foundations and initiatives may vary in the criteria what open source is. For comparison, the Free Software Foundation (2023c) defines four freedoms which open source software should have:

1. Run the program for any purpose
2. To access and change the source code
3. Redistribute freely copies of the software
4. To redistribute modifies versions of the software.

Additionally, in terms of licensing and copyleft, the FSF takes a different stance by allowing for a stronger copyleft. This means that an author can restrict the redistribution of their software, unless the same copyleft terms are applied. (Free Software Foundation, 2023a).

To summarize, the main difference between these two definitions and criteria is that the Open Source Initiative allows licenses which the Free Software Foundations considers as too restrictive.

These criteria show that the license is an essential part of any open source or proprietary software and is not easily to define. Therefore, this important topic is described in more detail at the open source part Licensing and at the proprietary part Open Source Components in Proprietary Software.

3.1.2 History

Open source has a long history which Schrape (2016) groups into four main phases.

The first phase started in the 1950's. This era is referred to as the "collective inventions" where software and hardware were seen as completely separate products. In this time, the development and exchange of knowledge was mainly established between universities.

The second phase was around mid 1960's and can be described as "commodification and emerging subcultures". In this time, software was seen as an independent product and was sold independently from the hardware. During this era, different software licensing models were introduced.

The third phase started in the 1980's and can be described as "institutionalization". In this phase, independent developer groups got support by the Free Software Foundation institutions. Therefore, besides research and companies, foundations supported the development of free software. Also in this time, the first Copyleft licenses got introduced.

The last phase, according to the author, started in 1998 and can be described with the two words "growth" and "diversification". The Open Source Initiative was founded and companies like NetScape, which was at the time one of the most well recognized internet browsers, or RedHat started to make software open source. In this phase, the new type of commercialized open source software was introduced.

Due to the rapidly changing software landscape, a new chapter in the history of Open Source Software can be included that lasts until the present day. Its start can be traced back to around 2015 when there was a surge in large-scale acquisitions of open source companies by established ones. One of the largest acquisition was when IBM bought RedHat for $ 34 billion in 2018 which can be seen as a remarkably large investment in open source development (IBM, 2018). In the same year, Microsoft bought one of the largest development platforms, namely GitHub, for $ 7.5 billion (Center, 2018). GitHub is one of the most popular development platforms used to collaborate and host open source software, thus this investment also shows that the established tech companies increase their presence in the open source market.

Looking at these developments and especially the significant purchases by large global corporations it can be concluded, that open source software development will have an increasing significance and as a consequence more investments in the field of security can be expected.

3.1.3 Licensing

As already mentioned in the criteria section, licensing is an important aspect especially with regards to how the software can be used. Therefore, this section will focus on the key aspects of the most commonly used open source licences, as shown in the figure about the "market share of open source licenses". This should give a brief overview of the different licenses and where they are applied.

Firstly, it has to be clarified how the different copyleft levels can be described. According to Sen et al. (2008), they can be categorized in three groups: "Strong-Copyleft License", "Weak-Copyleft License" and "Non-Copyleft License". In the first group are licences where any derived or based software must apply a licence very similar to that of the original software. For the second group, it is also allowed to apply a different license, but only under certain conditions. The last one is the most permissive group, which allows to use other software without license restrictions. These groups can be exemplified the following way:

Apache 2.0: This is in the category of a Non-Copyleft license. It also permits to distribute work under a different license if large changes are made to the source code, but the original license must be included. A large open source project which uses this licenses is Tenserflow[7]

MIT: This is in the category of a Non-Copyleft license. It is a short licenses and the only condition is, that the original license and copyright must be preserved. The [8] by Microsoft uses this license.

GPLv3.0 and GPLv2.0: Both licenses belong to the category of Strong-Copyleft Licences, which requires additional conditions. Generally, this implies that the source code of the modified version must be released as open source, and the same licenses must be applied. For example the [9] uses this license.

[7]https://www.tensorflow.org/
[8]https://dotnet.microsoft.com/en-us
[9]https://www.r-project.org/

LGPL 2.1: This belongs to the Weak-Copyleft Licenses. This means, that other licenses can be applied, but only conditionally. One example of this would be if this is a software library and is isolated, it can be used without restriction. (Free Software Foundation, 2023b). An example for a project which uses this license is PHP Mailer[10].

BSD 3 and BSD 2: This belongs also to the category of Non-Copyleft License and it requires to add the license and copyright notice to the source modified work. An example of a well known project is Flutter[11].

All the licenses which are mentioned above have the limitation that no liability or warranty is provided. Another thing all these licenses have in common is that they are allowed to be used in commercial and private contexts. This brief overview already illustrates that there is a lot of potential for conflicts or incorrect use of these licenses in an open source project. A software adviser found in 2409 scanned codebases, that 53% had licensing conflicts (SYNOPSIS Inc., 2022).

3.2 Maintenance and Organization

Maintenance and organizational structure represent key components for successful open source projects. Unfortunately, many software projects cease to develop and updates slow down, often resulting in their abandonment. The process of collaboration in these projects is based on the availability of the source code, thus enabling anyone to propose changes and improvements to the software. Due to this, significant projects are managed by corporations, collectives, foundations, public institutions or other groups. However, smaller projects are overseen by one or more independent individuals. The process of collaboration in these projects is based on the availability of the source code, thus enabling anyone to propose changes to the software. Depending on the project's organizational structure, the proposed change will be managed accordingly.
The next two subsection will describe the topics of maintenance and organization in more detail.

3.2.1 Maintenance

It is important to take maintenance in consideration when deciding on an open source software or packages because it is a crucial part to ensure stability and security but it is not always clear whether the software or package is still maintained.

Before further elaboration of this topic, it is important to distinguish between two

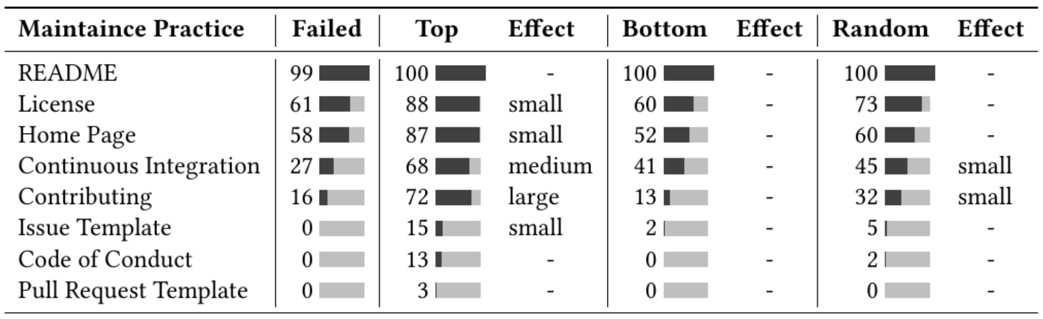
[10]https://github.com/PHPMailer/PHPMailer
[11]https://flutter.dev/

groups of individuals in this context: The maintainer and the contributor. The role and responsibilities of a maintainer are more, but not exclusively, administrative. For example, they have to manage how the contribution should work, e.g. where the source code is hosted and which development platforms are used. This group also makes the decisions what contributions are accepted and which are rejected. The other group are the contributors. This group of individuals contribute through code changes and general improvements by adding features or fixing bugs and security problems. This means that this group can only propose changes to the maintainers mostly through so called pull requests which the maintainer can approve or decline. Note that these two categories are only a broad view and in this paper, going into further details like distinguishing between core participants, regular contributors, etc. would go beyond the defined scope.

Regardless of the above described role, the individuals who contribute to FOSS, are mostly doing this in their free time. A survey of open source project maintainers and contributors by The Linux Foundation written by Frank Nagle et al. (2020), shows that nearly all of the interviewed individuals work full time and that nearly half of them contribute to open source project also during their work time. Apart from contributing in a work context, the report suggests that individuals are motivated to contribute for non-financial reasons. The top three motivations to contribute are adding a new feature or code fixes, enjoying learning and fulfilling the need for creative and enjoyable work. In contrast to that, the least motivating reasons are financial gains through developing, career boosts and peer recognition. Therefore, it is interesting that the report suggests that financial help is one of the most beneficial supports for FOSS projects. The reason for this is that the money is mainly allocated for security and infrastructure (i.e. CI) investments.

Since open source projects rely on their contributors, it is clear that projects have a risk of stopped development and as already mentioned at the start of this section, it is not trivial whether an open source project is maintained or not. In order to tackle this problem, one study about GitHub projects Coelho et al. (2020) created a machine learning model to classify GitHub repositories regarding their maintenance state. The reason for choosing GitHub in particular is that it is one of the most used open source project platforms. For this, the study used indicators like the commits, forks, owner of the repository (whether it is an individual or an organization) and other GitHub related indicators. The study also suggest that the owner of the repository is not significant, but rather if practices like a CI or contribution guidelines are important.

The following table shows what impact the recommended measurements for open source repositories have in terms of further development.



Maintaince Practice	Failed	Top	Effect	Bottom	Effect	Random	Effect
README	99	100	-	100	-	100	-
License	61	88	small	60	-	73	-
Home Page	58	87	small	52	-	60	-
Continuous Integration	27	68	medium	41	-	45	small
Contributing	16	72	large	13	-	32	small
Issue Template	0	15	small	2	-	5	-
Code of Conduct	0	13	-	0	-	2	-
Pull Request Template	0	3	-	0	-	0	-

Figure 3: Recommended practices for managing open source repositories (Coelho & Valente, 2017)

According to Coelho and Valente (2017), the reasons for discontinued development on an open source software are due to project characteristics such as bad code design choices, team reasons like time constraints and due to reasons in the environment, e.g. when a competitor did something similar and thus the project got obsolete. The author also mentions the strategies to mitigate these problems, which are moving the project to an organization, transferring the account to a new maintainer or adding new developers to the core team.

3.2.2 Organization

When the project scales up and gets more contributors and moreover a broader interest, new organizational challenges arise. These can be from managing code contributions, handling donations to legal aspects, like intellectual property rights and liability. Therefore, the organizational structure is also a key element for handling security related issues. Because of this, the evolution of a successful open source project can be described according to De Laat (2007) in three stages: Spontaneous governance, internal governance and increasing institutionalization.

The first phase is characterized by non formal coordination or control and more self directing of the few core participants. In the second phase, more individuals work together on the project and therefore the next phase is called the "internal governance". Key characteristics of this phase are modularization, division of roles, delegation of decision-making, member management, formalization of decision making like autocracy and democracy. The last phase the author describes is governance towards outside parties. In this phase, the project and community has tendencies towards increasing institutionalization because through a legal entity, the community has benefits like to claim copyright licenses and trademarks.

In the last two phases, the open source community matures and the question arises how to proceed with the organizational structure of the community. In this case, non profit or charitable foundations are a widespread choice for these projects. Although the community has to decide how they want to implement it, e.g. joining a foundation or to create an own, the key factor according to R. Eckert et al. (2019) is independence. The author describes three possible ways how a community can transition to a foundation. The first option is the autonomous approach, where the open source community creates their own foundation and therefore keeps the complete independence from others. However, the downside of this approach is the administrative workload and possible financial risks. The second approach is to create a foundation but to outsource tasks to a "service organisation" which can also be another foundation. These tasks include handling donation payments or other legal matters. This approach has the benefit that the community keeps their independence and also reduces the administrative workload. However, the drawback is that creating an own legal entity could be a financial risk. Lastly, there is the integrated approach, where the open source community joins an umbrella foundation. In this foundation, they are their own working group with some degree of independence and they can profit from the reach and other benefits from the foundation. For this approach, the drawbacks are that the community looses some independence and that the licensing could oppose a conflict of interest. The following figure illustrates "different approaches in terms of foundations"

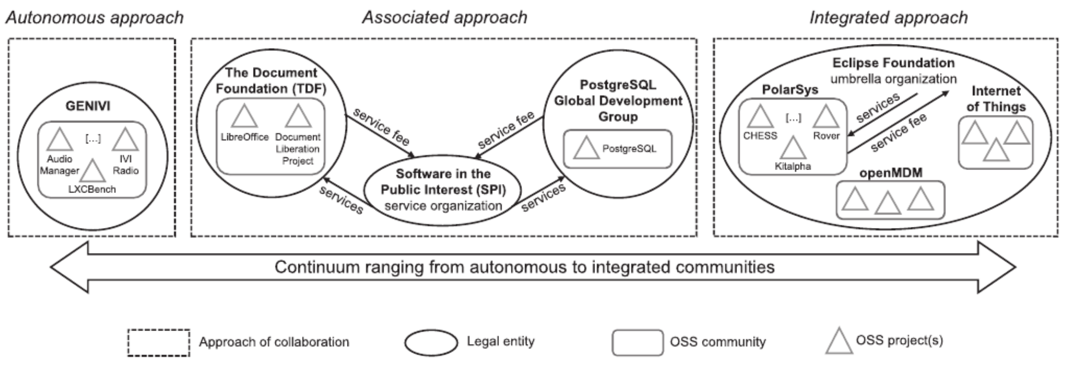


Figure 4: Different approaches for open source communities (R. Eckert et al., 2019)

Nevertheless, also profit oriented companies can organize open source projects and communities. There are also different approaches by the companies to organize and sustain the community. For example, a company has made their main product open source and sells extra features as proprietary. In this case, individuals can also improve the main product by adding some features or fixing bugs, but the approval is controlled by the company (Duparc et al., 2022). There are some more organization models which differ mainly in their revenue model and because of this, these are elaborated further in the section Funding.

When it comes to participation in an open source project as already mentioned in the section Maintenance, it is important that the organization is neutral and contributions to an open source project with for-profit organization was avoided (Frank Nagle et al., 2020). Therefore, the organizational structure is a key element to for a sustainable and secure open source project.

3.3 Funding

In the field of open source, funding is an important topic which helps to further develop, improve and maintain the project. Due to this reason, there are besides the foundations also for-profit organizations which have a business model built with open source. Besides that, also smaller projects have the ability to get funding through platforms like "open collective[12]", "librepay[13]" or other direct sponsoring like on GitHub[14]. Furthermore, when it comes to funding, there is also a difference between direct financial and indirect funding, e.g. when a developer is contributing to an open source project at work (Frank Nagle et al., 2020). In this paper, the term funding always refers to direct financial funding.

According to Duparc et al. (2022), there are seven open source business model archetypes. Firstly, there are the profit oriented business models. The first model is the "Infrastructure", which offers an open source software with a copy left licence and sells a web-based infrastructure to their product. Secondly, there is the "Open-Core Platform", which has an open source version and offers another extended version with proprietary features. The organization is clearly done by the company and it uses open source and proprietary licenses. The third model which the author describes is the "Proprietary-like" which only has some software open source, but the main product is proprietary. Also in this model, the company builds upon open source but does not return much back to the open source community. The next model is the "Open Innovation", which offers different products with permissive and proprietary licenses but builds on co-creation, knowledge transfer, and collaboration. In this business model it is common that they offer their service tailored to businesses. These business models described before differ also in the interest for building communities for their open source projects, but the author also describes the non profit oriented business models which are described in the following part.

Firstly, the author describes the "Open Source Platforms", which are primarily non-profit but are mostly subsidized by a company. The interest of these companies is to sell this project as a finished solution, so it is more indirect. In comparison to the "Infrastructure model", this also allows besides the copyleft also permissive licenses. Furthermore,

[12]https://opencollective.com/
[13]https://liberapay.com/
[14]https://github.com/sponsors

the model tries to utilize a network effect and attract a large number of user and contributors. The next archetype is the "Funding Based model", which is non commercial and is based on voluntary work and funding. Most of the foundations have this business model and therefore the licensing according to them can be permissive and copyleft. The last business model which the author describes is the "Traditional". In this case it is hard to say whether it is really a business model because it has no clear structure and is created by a community or individual. Financial help is mostly obtained through donations and the licensing part.

With the described business models above it is clear for the profit seeking ones that investments in security concerns should be allocated by the companies. In contrast, the non profit business models rely on their community and donations and therefore it is not as straight forward how to address this issue.

Let's take one of the largest non-profit foundation, namely the Apache Software Foundation as an example to illustrate how they deal with security problems. This foundation has it's own teams assigned to these topics which deals professionally with the reported security problems (Cox, 2023). A large and well recognized foundation has the capacity and the financial resources to have such teams but this is not the case for smaller projects. A study by Zhou et al. (2022) about Open Collective shows, that 54% of the donations are spent on non technical investments like travel and the other 46% are spent on technical matters. 18% of the technical spendings are spent on bug bounty programs. Furthermore, a survey of the Linux Foundations by Frank Nagle et al. (2020) of open source project developers shows that financial donations are likely to be allocated for security related issues. Therefore, the donations could be invested in software security specialists which can create for example an audit report.

To end the topic about funding, the two types of contributions will be elaborated which apply to the for-profit and non-profit business models. These models are funded through financial support or with development contributions. When it comes to the financial part, it is straight forward for the for-profit organizations since they are funded through their business model. For the non-profit organizations, funding is realized mostly through donations from individuals and companies, but it can be also through services like sponsoring infrastructure. For example the navigation app "Organic Maps" gets their IT Infrastructure sponsored by an internet service provider ("Improving the world bit by expensive bit - Mythic Beasts," 2021). For the development contributions, which can include code improvements, bug fixes, adding features or documenting, the resource is basically the time of a developer. This time can be spent in the developer's free time or at work which is also broadly accepted by employers. If such contributions were made, the added value to the project by the company should also be clearly indicated (Frank Nagle et al., 2020). This is important because a company has their own agenda and can influence the open source project in its own interests (Munir et al., 2018).

3.4 Security

One of the main problems is, that many open source libraries are used in the supply chain and therefore can create unexpected security vulnerabilities. The following pictures illustrates this drawing on the example of the Heart vulnerability, which is described in the example section Heartbleed which occurred in OpenSSL.

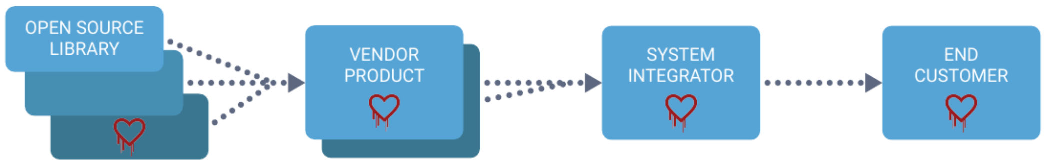


Figure 5: supply chain of software packages (Synopsys, Inc., 2017)

Therefore it is crucial to have secure open source packages and software. As already discussed in the section Software Security in Development, there are different approaches and tools to improve open source software. One of the problems with employing tools is, that especially monetary resources are limited to use professional security tools for the code. To address this issue, Stephen Hendrick and Martin Mckeay (2022) suggests, that companies, which use open source software, should use automated tools, like supply chain analysers, to improve their open source components. Another issue addressed in a survey by Frank Nagle et al. (2020) is, that only 2,27% is spent by open source contributors on security related issues. To address this issue, the survey showed, that free security audits and contributions in vulnerability and bug fixes, would help the most. Another point to add is, that there are efforts to train and create awareness for contributors in security related topics. One of these projects is OpenSSF [15] by the Linux Foundation, which offers free training and certifications.

Besides the problems mentioned above, a report by Gordon Haff (n.d.) about enterprise open source found that 89% perceived open source a more secure than proprietary software. Furthermore the report shows, that these interviewed companies see the benefits mostly in the well tested projects for their in house applications and that the security patches are well documented and can be scanned for.

[15]https://openssf.org/

The most important point in terms of security in open source is, that it is auditable by everyone. This means, that any expert, developer or individual can verify what the software does and therefore it can build trust. Furthermore, through these practices it is transparent and documented when a security vulnerability occurs (Bursell, 2020). Among other reasons, this could be a strong reason why many security and cryptography projects are open source. Software like OpenSSL[16], which is used for secure networking protocols and encryption (OpenSSL, 2022) are open source. These tools are essential to secure the digital world and if there is a security issue. Furthermore some companies, which provide a security oriented service, are employing open source as measurement for transparency and trust (Bursell, 2020). To illustrate that, e-mail provider like the German based service Tutanoto[17] or the Swiss based Proton mail [18] employ open source for their secure e-mail software.

To conclude this topic, security and open source are not excluding each other rather it is a synergy.

3.5 Examples Security Issues

Each year many new vulnerabilities in open source software are discovered and reported. Only in 2020, there were around 9.658 new vulnerabilities made public which was an increase of 50% from 2019 (Adam Murray, 2021a). To put this number in perspective, this chapter shows two examples of well known security vulnerabilities in open source software which had a global impact.

The Heartbleed vulnerability (CVE-2014-0160[19]) was found in the widely used library OpenSSL[20] which is a toolkit cryptography and secure communication (TLS - Transport Layer Security and SSL - Secure Socket Layer). The Heartbleed was found on the 3rd of April 2014 and was made public on the 7th of April 2014 with a patched version on the same day (Synopsys, Inc, 2020). The vulnerability was found in the part of OpenSSL, where the library checks the connection by sending and receiving a message and with exploiting the vulnerability, the attacker could have access and read the memory of the attacked machine (Synopsys, Inc., 2017), hence allowing the attacker to read sensible information of the attacked machine. (Ronald Eikenberg, 2014). This vulnerability existed for two years in OpenSSL before it was discovered and it was estimated that around 600 thousand servers were affected (Synopsys, Inc., 2017).

[16]https://www.openssl.org/
[17]https://tuta.com
[18]https://proton.me/
[19]https://www.cve.org/CVERecord?id=CVE-2014-0160
[20]https://www.openssl.org/

The Log4Shell vulnerability (CVE-2021-44228[21])) was found in a widely used logging package, namely Log4j[22] for the programming language Java. According to IBM (n.d.), the vulnerability was found on the 24th of November 2021 and got reported to the Apache Software Foundation. It was shared publicly on the same day as the first patch was released on the 10th of December 2021. In the first patch was another security vulnerability which was then finally patched on the 28th of December 2021. The code which could be exploited for the Log4Shell vulnerability was implemented on the 18th of July 2013. This means, that this severe vulnerability had been in the Log4J library for around 9 years.

With the Log4Shell vulnerability, the attacker could easily get admin rights through the Log4J logging objects and execute code with them. This attack can be described as a "Remote Code Execution" (PwC, 2021). The big issue with this was, as already mentioned in the introduction, that the library Log4J can be found in many applications. Therefore, the vulnerability Log4Shell could be exploited in many applications such as critical web server infrastructure, smart homes and many more Java based applications (Fabian A. Scherschel, 2022). The outreach of Log4Shell was so large that governments issued warnings about this vulnerability. The Austrian government set the alarm level to code red and listed mitigations as well as recommended next steps (A-SIT Zentrum fÃ¼r sichere Informationstechnologie â€“ Austria, 2021).

4 Proprietary Software

Proprietary software is the standard for commercial software (Gartner, Inc., n.d.) and is owned by a company or other legal entity. This is best illustrated by the operating systems for personal computers, which are dominated by Windows with 68% market share according to StatCounter (n.d.-a). Furthermore, proprietary has software a market share of 45% in enterprise software according to a report by Gordon Haff (n.d.). Especially businesses rely on these closed source technologies, which shows the high significance of proprietary software in our world.

This chapter starts with definitions and licensing, which are key aspects of securing ownership for a software. Next I will describe different maintenance aspect which are fundamental for a software company in a legal and customer aspect. Additionally, the next chapter describes open source components in proprietary software, which is also a topic of high relevance due the coherent structure of software. Lastly I will describe two examples of security issues, which occurred in proprietary software and had a high impact.

[21]https://www.cve.org/CVERecord?id=CVE-2021-44228
[22]https://logging.apache.org/log4j/2.x/

4.1 Definition and Licensing

The following two sub chapters elaborate on the definitions of important concepts with proprietary software and different licencing models. When it comes to licenses, it should be highlighted that there are many different models and specific license agreements between the vendor and customer. Therefore, only a short and rough overview is provided.

4.1.1 Definitions

According to Gartner, Inc. (n.d.), proprietary software is defined by the ownership of the software which could be a company or an individual. Another aspect is, that proprietary software it is the de facto standard for commercial software because it is based on proprietary protocols or standards which can be an obstacle for the usage. In addition to the definition, proprietary software is mostly closed source. This means the source code can not be publicly accessed.

Furthermore a risk which can occur when using proprietary software is vendor lock-in. According to Opara-Martins et al. (2016) this problem occurs when the customer is dependant on a single technology provider and can not easily move to a different vendor. Therefore, the author suggest that in case of a lock-in situation, interoperability and portability are challenging. This can result in higher costs and complexity when switching to a different vendor.

Additionally, a new security threat is also important to define. This is the security issue of reverse engineering, which is according to Chikofsky and Cross (1990) the process of analysing a system to identify the system's components how the software is build. Therefore, the attacker can have different motivations like extracting secret informations from the software or to alter the software's behaviour (Schrittwieser et al., 2017).

4.1.2 Licenses

When it comes to licencing, there are different possibilities how proprietary software can be licenced. This is a complex topic and therefore I will try to give a simple overview without much detail. According to Ivanti (2020), it can be distinguished between subscription and perpetual licenses. The difference between these two are, that the subscription is limited in time whereas software with perpetual license can be used indefinitely. Therefore, the subscription model is mostly also referred to as a "software as a service" licence. Another licencing strategy which the author describes, is the device licencing. With this type, the software can only be installed on a specific computer or data centre device. Another possibility how software can be licenced is through consumption based licencing. The author describes this method as flexible because the customer pays only when a user accesses the software or feature. This licencing approach is used for example in software products which are infrastructure as a service.

Security considerations should be taken into account with these different licensing models in mind. It has to be considered that the availability of updates and patches differ for perpetual licences and subscription models (Choudhary, 2007).

4.2 Maintenance

In the software lifecycle, maintenance is one of the most important parts of a company which sells software or services. The paper of Zelkowitz (1978) suggests, that only 25% - 33% of the development efforts go into creating the product and this underlines the importance of maintenance. Therefore, it is an expensive part and the authors suggest, that companies under-evaluate the effort and spending on this topic. Therefore, the question arises how these processes can be optimized. Bhatt et al. (2006) describes the possibility of outsourcing activities like maintenance to other companies and focussing on the core competencies.

Generally for any maintenance activity, the company, which built the software is responsible. Therefore, the companies should act when a vulnerability is detected in their software. But there can be a risk of increased complexity when dealing with vulnerabilities. According to Reis et al. (2021), of all analysed patches, 38.29% increased software complexity. The author further argues that through the increased complexity, new bugs and vulnerabilities are more likely occur which results in a decrease of maintainability. This should be considered when tasks are outsourced since this could lead to financial risks for the company.

Furthermore, due to the dependencies of open source software in commercial software, maintenance also applies to the dependencies. According to the recent report of (SYNOPSIS Inc., 2023), over 89% of the scanned commercial code repositories had open source elements, which were out of date for more than 4 years. Another finding of this report is, that 91% of these codebases had components which hadn't any new development in over two years. This topic gets further elaborated in the section Open Source Components in Proprietary Software

Nevertheless, maintenance is a topic where the companies have different approaches and there is no one solution which can be applied to every product. Therefore, it is important that the companies address security issues fast and deliver patches to the affected software. Also, it is important to avoid fixes which increase complexity because new problems and financial risks could arise. Lastly, it is highly advisable that companies keep their dependencies updated.

4.3 Open Source Components in Proprietary Software

Open source packages help to improve development time and therefore costs. According to the Open Source Security and Risk Analysis Report by SYNOPSIS Inc. (2023), around 96% of the scanned commercial software, had open source packages installed in 2022. The authors collected the data, through their "Software Composition Analysis" (SCA) tool on 1.703 codebases in 17 different industries. The report highlights that out of all these codebases, 54% had licence conflicts. The largest share of conflicts, which account for 35%, were found with the Creative Commons Attribution ShareAlike 3.0 and 4.0. The second largest occurrence of issues which add up to 15% were found with the GNU Lesser General Public Licence. This licence was used in 43% in the scanned codebases. Besides the licencing issues, the report shows that maintenance of the used packages is often problematic. As already mentioned, in the part Maintenance, the report shows, that 89% of the codebases are using open source software which was outdated by more than 4 years. Furthermore, 91% had packages in usage, which weren't further developed in over two years. These findings are problematic in terms of security, because the risk of so called supply chain attacks rise. Therefore, this security issue is further explained in the section Security.

The high usage of open source software in commercial software means that there is also a contribution towards the open source components taking place. A survey conducted by the Linux Foundation written by Frank Nagle et al. (2020), showed, that 48,7% of the developers are getting paid by the employers to contribute to open source projects. Regarding the contribution policies of companies to open source projects, the survey concludes that it is necessary to have clear policies in this regard.

To conclude this subchapter, it is important that trusted open source packages are also verified (SYNOPSIS Inc., 2023). Furthermore, proprietary software vendors should consider how they manage and contribute to open source components.

4.4 Security

Security in proprietary software has in general a different approach because the source code is not available. Therefore, it can be refereed to as a black box since no third party can audit the software without permission of the software producer. Due to this circumstance, it can be derived, that security matters like independent code audits, software testing and secure coding practices are the responsibility of the software producer. This also means that security problems can be present in the software for long and not be recognized until it is exploited (Adam Murray, 2021b).

Therefore, new security threats arise like the so called "Software Reverse Engineering". With this method, a third party tries to analyse the structure and functions of a compiled software. According to Schrittwieser et al. (2017), the motivations can be to extract information of a commercial software such as cryptographic keys or program parts, which are considered a trade secret. Therefore, the results can be among others intellectual property theft or that other parts of the software get compromised (OWASP Foundation, Inc., n.d.). It is difficult to defend against this threat, but possible security measurements could be through code obfuscation, which can be effective in restricted scenarios (Schrittwieser et al., 2017).

Also, supply chain attacks which follow a similar principle in regards of dependencies were already discussed in the open source security chapter. However, these are also very relevant in proprietary software. According to Ohm et al. (2020), the goal of this attack is to change the software by a trusted vendor with a malicious dependency. This can be achieved through tempering existing packages or in the numerous transitive dependencies in the packages and therefore exploit the developers trust in these package hosts. The importance of this issue is shown by “Gartner Top Security and Risk Trends in 2022” (2022) with a prediction, that 45% of companies will have experienced a supply chain attack until 2025.

4.5 Examples Security Issues

EternalBlue (CVE-2017-0144) was a vulnerability which led to the global distribution of the ransomware WannaCry on computers with the Windows operting system. This vulnerability did not exist by accident, rather it was a backdoor held open in the SMB (Windows Server Message Block protocol). Therefore, the backdoor could be used to execute code remotely. This vulnerability was leaked in April 2017 and finally patched on 14. March 2017 by Microsoft (Newman, 2018).

The second example shows the impact of a supply chain attack with the example of the popular freemium software CCleaner[23]. The original version of the software was replaced through a malicious version which had a backdoor for the attackers. This modified version of the program was undetected downloadable on the official website from the 2. August 2017 until 13. of September 2017. In this time period, the software was downloaded by over 2.2 million end users (Swati Khandelwal, 2018).

[23]https://www.ccleaner.com/

5 Comparison

In this section I will compare the open source and proprietary software concept on security aspects. Therefore, this part will not include the relevant but indirect factors of the different organizational structure, legal issues or licencing.
The key difference between open source and proprietary software is whether or not the underlying source code is accessible. Therefore, in open source software, anyone can check and find security vulnerabilities directly in the source code. As a result, these security problems can be reported and fixed by anyone. In contrast, proprietary software, where the source code is not available, requires the company producing the software to check the source code for security problems. This means that the software is like a black box, and vulnerabilities can hide undetected for a long time until they are exploited. This difference boils down to transparency and trust. In open source, anyone can audit the security aspects of the software, which results in complete transparency and therefore can build trust, while this is not possible in the same way with proprietary software.

Next, the examples shows, the differences and similarities of security issues. In both examples, the vulnerability was present in the software for a long time. But the EternalBlue example shows a key difference: the vulnerability was a built-in backdoor, whereas in open source software it would be much more difficult to have a security vulnerability hidden for such a long time. Also a similarity of both concepts are the dependability on other packages which can lead to security vulnerabilities. Furthermore, proprietary software depends on open source projects, and is therefore dependent on the security problems of open source projects. It is not the other way around - open source is not dependent on proprietary software. Also in this regard, the reporting of security issues is in open source software is well documented which is not always the case with proprietary software.

6 Conclusion and Discussion

In summary, there are many different factors involved in software security for proprietary and open source software. The factors of secure programming, security checks, organisation, funding and maintenance are also approached differently in these approaches and are therefore of different importance. In addition, open source and proprietary software security issues cannot be considered as completely separate issues because of the dependencies of proprietary software on open source. Therefore, there is no clear answer as to which concept is more secure than the other, as many factors must be considered.

For discussion, there may be a bias in this seminar paper due to the use of reports from commercial security and open source companies. This may be to the detriment of the proprietary software section, as it was difficult to find other reports or studies on the subject. Another point for discussion is that this paper has a broad perspective on software. Therefore, security issues are different for each application area, programming language and other technical topics.

Further research could look at how the open source security landscape is changing and which practices are proving effective. Another interesting topic would be how commercial software is responding to the growing security risk of supply chain attacks.

References

Adam Murray. (2021a, April 14). *All about mendâ€™s 2021 open source security vulnerabilities report*. Retrieved November 19, 2023, from https://www.mend.io/blog/2021-state-of-open-source-security-vulnerabilities-cheat-sheet/

Adam Murray. (2021b, March 14). *Open source vs proprietary software security* [Mend]. Retrieved December 13, 2023, from https://www.mend.io/blog/open-source-vs-proprietary-code-security/

Akbar, M. A., Sang, J., Khan, A. A., Fazal-E-Amin, Nasrullah, Shafiq, M., Hussain, S., Hu, H., Elahi, M., & Xiang, H. (2018). Improving the quality of software development process by introducing a new methodologyâ€“AZ-model. *IEEE Access*, *6*, 4811–4823. https://doi.org/10.1109/ACCESS.2017.2787981

Apache license 2.0. (n.d.). https://www.apache.org/licenses/LICENSE-2.0.html

APPLE INC. (n.d.). SOFTWARE LICENSE AGREEMENT FOR macOS sonoma. Retrieved October 30, 2023, from https://www.apple.com/legal/sla/docs/macOSSonoma.pdf

A-SIT Zentrum fÃ¼r sichere Informationstechnologie â€“ Austria. (2021, December 21). *Die sicherheitslÃ¼cke log4shell: Gefahrenpotenzial und gegenmaÃŸnahmen*. Retrieved November 19, 2023, from https://www.onlinesicherheit.gv.at/Services/News/Die-Sicherheitsluecke-Log4Shell.html

Bhatt, P., Shroff, G., Anantaram, C., & Misra, A. K. (2006). An influence model for factors in outsourced software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, *18*(6), 385–423. https://doi.org/10.1002/smr.339

Bursell, M. (2020, August 25). *Why we open sourced our security project | opensource.com*. Retrieved December 12, 2023, from https://opensource.com/article/20/8/why-open-source

Center, M. N. (2018, June 4). *Microsoft to acquire GitHub for $7.5 billion* [Stories]. Retrieved December 6, 2023, from https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/

Chikofsky, E., & Cross, J. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, *7*(1), 13–17. https://doi.org/10.1109/52.43044

Choudhary, V. (2007). Comparison of software quality under perpetual licensing and software as a service. *Journal of Management Information Systems*, *24*(2), 141–165. https://doi.org/10.2753/MIS0742-1222240206

Cloudflare, Inc. (n.d.). *What is penetration testing? | what is pen testing?* [Cloudflare]. Retrieved December 10, 2023, from https://www.cloudflare.com/learning/security/glossary/what-is-penetration-testing/

Coelho, J., & Valente, M. T. (2017). Why modern open source projects fail. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 186–196. https://doi.org/10.1145/3106237.3106246

Coelho, J., Valente, M. T., Milen, L., & Silva, L. L. (2020). Is this GitHub project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, *122*, 106274. https://doi.org/10.1016/j.infsof.2020.106274

Cox, M. (2023, January 31). *ASF security report: 2022* [Section: blog]. Retrieved December 9, 2023, from https://security.apache.org/blog/asf-security-report-2022/

Cruz, D. B., Almeida, J. R., & Oliveira, J. L. (2023). Open source solutions for vulnerability assessment: A comparative analysis. *IEEE Access*, *11*, 100234–100255. https://doi.org/10.1109/ACCESS.2023.3315595

De Laat, P. B. (2007). Governance of open source software: State of the art. *Journal of Management & Governance*, *11*(2), 165–177. https://doi.org/10.1007/s10997-007-9022-9

Digital Maelstrom. (n.d.). *The secure software development lifecycle explained.* Retrieved November 19, 2023, from https://www.digitalmaelstrom.net/it-security-services/secure-software-development-lifecycle-ssdlc/

Duparc, E., MÃ¶ller, F., Jussen, I., Stachon, M., Algac, S., & Otto, B. (2022). Archetypes of open-source business models. *Electronic Markets*, *32*(2), 727–745. https://doi.org/10.1007/s12525-022-00557-9

Eckert, C. (2018). *IT-sicherheit: Konzepte, verfahren, protokolle* (10. Auflage). De Gruyter Oldenburg.

Eckert, R., Stuermer, M., & Myrach, T. (2019). Alone or together? inter-organizational affiliations of open source communities. *Journal of Systems and Software*, *149*, 250–262. https://doi.org/10.1016/j.jss.2018.12.007

European Commission. Directorate General for Communications Networks, Content and Technology. (2021). *Safety and liability related aspects of software.* Publications Office. Retrieved December 10, 2023, from https://data.europa.eu/doi/10.2759/760023

European Parliament, Council of the European Union. (2019, May 20). DIRECTIVE (EU) 2019/770 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL. https://eur-lex.europa.eu/eli/dir/2019/770/oj

Fabian A. Scherschel. (2022). *Log4shell: Eine bestandsaufnahme.* Retrieved November 19, 2023, from https://www.heise.de/hintergrund/Log4Shell-Eine-Bestandsaufnahme-6342536.html

Frank Nagle, David A. Wheeler, Hila Lifshitz-Assaf, Haylee Ham, & Jennifer L. Hoffman. (2020, December 10). *Report on the 2020 FOSS contributor survey, the linux* *foundation & the laboratory for innovation science at harvard.* https://www.linuxfoundation.org/resources/publications/foss-contributor-2020

Free Software Foundation. (2023a, January 31). *Categories of free and nonfree software - GNU project - free software foundation.* Retrieved November 19, 2023, from https://www.gnu.org/philosophy/categories.html.en

Free Software Foundation. (2023b, June 11). *GNU lesser general public license v2.1 - GNU project - free software foundation.* Retrieved November 20, 2023, from https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html

Free Software Foundation. (2023c, September 8). *What is free software? - GNU project - free software foundation.* Retrieved November 19, 2023, from https://www.gnu.org/philosophy/free-sw.html.en#selling

Gartner, Inc. (n.d.). *Definition of proprietary software - gartner information technology glossary* [Gartner]. Retrieved December 12, 2023, from https://www.gartner.com/en/information-technology/glossary/proprietary-software

Gartner top security and risk trends in 2022 [Gartner]. (2022, April 13). Retrieved December 13, 2023, from https://www.gartner.com/en/articles/7-top-trends-in-cybersecurity-for-2022

GitHub Inc. (n.d.). *Appendix | choose a license.* Retrieved December 10, 2023, from https://choosealicense.com/appendix/

GitLab B.V. (n.d.). *Integrating security into your DevOps lifecycle.* Retrieved December 10, 2023, from https://about.gitlab.com/solutions/security-compliance

Gordon Haff. (n.d.). *The state of enterprise open source 2022.* Red Hat Inc. https://www.redhat.com/en/enterprise-open-source-report/2022

Herath, H. S., & Herath, T. C. (2014). IT security auditing: A performance evaluation decision model. *Decision Support Systems*, *57*, 54–63. https://doi.org/10.1016/j.dss.2013.07.010

Humayun, M., Niazi, M., Jhanjhi, N. Z., Mahmood, S., & Alshayeb, M. (2023). Toward a readiness model for secure software coding. *Software: Practice and Experience*, *53*(4), 1013–1035. https://doi.org/10.1002/spe.3175

IBM. (n.d.). *What is the log4j vulnerability? | IBM.* Retrieved December 9, 2023, from https://www.ibm.com/topics/log4j

IBM. (2018, October 28). *IBM TO ACQUIRE RED HAT, COMPLETELY CHANGING THE CLOUD LANDSCAPE AND BECOMING WORLDâ€™S #1 HYBRID CLOUD PROVIDER* [Press releases]. Retrieved December 6, 2023, from https://www.redhat.com/en/about/press-releases/ibm-acquire-red-hat-completely-changing-cloud-landscape-and-becoming-worlds-1-hybrid-cloud-provider

Improving the world bit by expensive bit - mythic beasts. (2021, October 6). Retrieved December 9, 2023, from https://www.mythic-beasts.com/blog/2021/10/06/improving-the-world-bit-by-expensive-bit/

Ivanti. (2020, July 23). *Software license types explained: What you need to know*. Retrieved December 12, 2023, from https://www.ivanti.com/blog/software-license-types

Kalin, M. (n.d.). *Code memory safety and efficiency by example | opensource.com*. Retrieved December 10, 2023, from https://opensource.com/article/21/8/memory-programming-c

Khan, R. A., Khan, S. U., Khan, H. U., & Ilyas, M. (2022). Systematic literature review on security risks and its practices in secure software development. *IEEE Access*, *10*, 5456–5481. https://doi.org/10.1109/ACCESS.2022.3140181

Laurence Goasduff. (n.d.). *Gartner forecasts global security and risk management spending to grow 14% in 2024* [Gartner]. Retrieved December 11, 2023, from https://www.gartner.com/en/newsroom/press-releases/2023-09-28-gartner-forecasts-global-security-and-risk-management-spending-to-grow-14-percent-in-2024

Microsoft Inc. (2021, June). *MICROSOFT SOFTWARE LICENSE TERMS WINDOWS OPERATING SYSTEM*. Retrieved October 30, 2023, from https://www.microsoft.com/en-us/Useterms/OEM/Windows/11/Useterms_OEM_Windows_11_English.htm

MIT. (n.d.). *The MIT license*. https://mit-license.org/

Munir, H., LinÃ¥ker, J., Wnuk, K., Runeson, P., & Regnell, B. (2018). Open innovation using open source tools: A case study at sony mobile. *Empirical Software Engineering*, *23*(1), 186–223. https://doi.org/10.1007/s10664-017-9511-7

Newman, L. H. (2018). The leaked NSA spy tool that hacked the world [Section: tags]. *Wired*. Retrieved December 13, 2023, from https://www.wired.com/story/eternalblue-leaked-nsa-spy-tool-hacked-world/

NIST. (2021). Source code security analyzers [Last Modified: 2023-11-20T12:16-05:00]. *NIST*. Retrieved December 10, 2023, from https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers

Ohm, M., Plate, H., Sykosch, A., & Meier, M. (2020). Backstabberâ€™s knife collection: A review of open source software supply chain attacks [Series Title: Lecture Notes in Computer Science]. In C. Maurice, L. Bilge, G. Stringhini, & N. Neves (Eds.), *Detection of intrusions and malware, and vulnerability assessment* (pp. 23–43). Springer International Publishing. https://doi.org/10.1007/978-3-030-52683-2_2

Opara-Martins, J., Sahandi, R., & Tian, F. (2016). Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective. *Journal of Cloud Computing*, *5*(1), 4. https://doi.org/10.1186/s13677-016-0054-z

open source initiative. (2006, July 7). *The open source definition* [Open source initiative]. Retrieved November 16, 2023, from https://opensource.org/osd/

Open source licenses in 2022: Trends and predictions [Mend]. (n.d.). Retrieved October 30, 2023, from https://www.mend.io/blog/open-source-licenses-trends-and-predictions/

OpenSSL. (2022, June 24). *Command line utilities - OpenSSLWiki* [OpenSSL wiki]. Retrieved December 12, 2023, from https://wiki.openssl.org/index.php/Command_Line_Utilities

OWASP Foundation, Inc. (n.d.). *M9: Reverse engineering | OWASP foundation.* Retrieved December 13, 2023, from https://owasp.org/www-project-mobile-top-10/2016-risks/m9-reverse-engineering.html

Prapenan, G. G., & Pamuji, G. C. (2020). Information system security analysis of XYZ company using COBIT 5 framework and ISO 27001:2013. *IOP Conference Series: Materials Science and Engineering*, *879*(1), 012047. https://doi.org/10.1088/1757-899X/879/1/012047

PwC. (2021, December 17). *Log4j-schwachstelle: Log4shell.* https://www.pwc.de/de/im-fokus/cyber-security/log4j-schwachstelle-log4shell.html

Q-Success. (n.d.). *Usage statistics of web servers* [Usage statistics of web servers]. Retrieved December 1, 2023, from https://w3techs.com/technologies/overview/web_server

Reis, S., Abreu, R., & Cruz, L. (2021). Fixing vulnerabilities potentially hinders maintainability. *Empirical Software Engineering*, *26*(6), 127. https://doi.org/10.1007/s10664-021-10019-z

Rina Diane Caballar. (2023, March 20). *The move to memory-safe programming - IEEE spectrum.* Retrieved December 10, 2023, from https://spectrum.ieee.org/memory-safe-programming-languages

Ronald Eikenberg. (2014, April 9). *Passwort-zugriff: Heartbleed-lÃ¼cke mit katastrophalen folgen.* Retrieved November 19, 2023, from https://www.heise.de/news/Passwort-Zugriff-Heartbleed-Luecke-mit-katastrophalen-Folgen-2166861.html

Schrape, J.-F. (2016). *Open-source-projekte als utopie, methode und innovationsstrategie: Historische entwicklung - sozioÃ¶konomische kontexte - typologie.* vwh, Verlag Werner HÃ¼lsbusch, Fachverlag fÃ¼r Medientechnik und -wirtschaft.

Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., & Weippl, E. (2017). Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys*, *49*(1), 1–37. https://doi.org/10.1145/2886012

SecurityScorecard. (2023, December 9). *Security vulnerabilities published in 2022.* https://www.cvedetails.com/vulnerability-list/year-2022/vulnerabilities.html

Sen, R., Subramaniam, C., & Nelson, M. L. (2008). Determinants of the choice of open source software license. *Journal of Management Information Systems*, *25*(3), 207–240. https://doi.org/10.2753/MIS0742-1222250306

Sommerville, I., & Sommerville, I. (2012). *Software engineering* (9., aktualisierte Auflage). Pearson.

StatCounter. (n.d.-a). *Desktop operating system market share worldwide* [StatCounter global stats]. Retrieved December 1, 2023, from https://gs.statcounter.com/os-market-share/desktop/worldwide/2022

StatCounter. (n.d.-b). *Mobile operating system market share worldwide* [StatCounter global stats]. Retrieved December 1, 2023, from https://gs.statcounter.com/os-market-share/mobile/worldwide/2022

Stephen Hendrick, & Martin Mckeay. (2022). *Addressing cybersecurity challenges in open source software*. https://resources.snyk.io/state-of-open-source-security-report-2022/open-source-security-report-2022

Swati Khandelwal. (2018, April 18). *CCleaner attack timelineâ€"here's how hackers infected 2.3 million PCs* [The hacker news] [Section: Article]. Retrieved December 13, 2023, from https://thehackernews.com/2018/04/ccleaner-malware-attack.html

Synk Ltd. (n.d.). *Sicherer SDLC | Sicherer Softwareentwicklungs-Lebenszyklus* [Snyk]. Retrieved November 29, 2023, from https://snyk.io/de/learn/secure-sdlc/

SYNOPSIS Inc. (2022). *2022 OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT* (No. 7).

SYNOPSIS Inc. (2023). *2023 OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT* (No. 8). https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

Synopsys, Inc. (2017). *Diary of a heartbleed*. https://www.synopsys.com/content/dam/synopsys/sig-assets/whitepapers/diary-of-heartbleed.pdf

Synopsys, Inc. (2020, June 3). *The heartbleed bug*. https://heartbleed.com/

WhiteSource. (2022, January 27). Most popular open source licenses worldwide in 2021 [graph]. in statista. Retrieved October 27, 2023, from https://www.statista.com/statistics/1245643/worldwide-leading-open-source-licenses/

Zelkowitz, M. V. (1978). Perspectives in software engineering. *ACM Computing Surveys*, *10*(2), 197–216. https://doi.org/10.1145/356725.356731

Zhou, J., Wang, S., Kamei, Y., Hassan, A. E., & Ubayashi, N. (2022). Studying donations and their expenses in open source projects: A case study of GitHub projects collecting donations through open collectives. *Empirical Software Engineering*, *27*(1), 24. https://doi.org/10.1007/s10664-021-10060-y