FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Shell Activity Logging and Auditing in Exercise Environments of Security Lectures using OSS

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Florian Pritz
Matrikelnummer 01125452

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Thomas Grechenig
Mitwirkung: Florian Fankhauser

Wien, 2019-05-02

(Unterschrift Verfasser) (Unterschrift Betreuer)

Shell Activity Logging and Auditing in Exercise Environments of Security Lectures using OSS

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Florian Pritz
Registration Number 01125452

elaborated at the
Institute of Information Systems Engineering
Research Group for Industrial Software
to the Faculty of Informatics
at TU Wien

Advisor: Thomas Grechenig
Assistance: Florian Fankhauser

Vienna, 2019-05-02

Statement by Author

Florian Pritz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Place, Date)	(Signature of Author)

Acknowledgements

I wish to thank Thomas Grechenig for supporting my work as my advisor and making it possible in the first place.

I thank Martin Moutran and Alexander Nawratil for their time in various meetings, expert interviews and their feedback on the proof of concept implementation.

Furthermore, I thank Brigitte Brem for her sincere guidance and kind help regarding the organizational aspects of writing a thesis, even if I may not have always emailed or called at the most convenient time.

Last, but by no means least, I thank Florian Fankhauser for his valuable feedback on various versions of this thesis. Especially during the early phases, his feedback has been very helpful in improving the structure of the work as a whole and thus helping to prevent time-intensive rewrites.

Kurzfassung

Ein Aktivitätsaudit ist ein Untersuchungsverfahren, bei dem Aktivitäten auf einem System aufgezeichnet und später hinsichtlich Missbrauch des Systems oder unautorisierte Aktivitäten untersucht werden. Solche Audits werden teilweise von Zertifizierungen und gesetzlichen Verordnungen verlangt, wobei durch diese oft eingeschränkt wird, wie Systeme einer Organisation bzw. deren Daten genutzt werden dürfen. Ein Auditor kann Audit-Logs nutzen, um zu verifizieren, dass die Systeme einer Organisation bzw. deren Daten den Anforderungen der jeweiligen Zertifizierungen oder Verordnungen genügen. Durch solch eine Überprüfung können Aktivitätsaudits nicht nur helfen, Missbrauch zu erkennen, sondern zusätzlich kann dadurch belegt werden, dass eine Organisation verantwortungsvoll agiert und sich an strikte Richtlinien hält.

Ein Beispiel, in dem Systemnutzung mit Hilfe von Aktivitätsaudits überwacht werden kann, findet sich in Übungsumgebungen von Universitäten. Unterschiedliche IT-Sicherheitslehrveranstaltungen bieten Studierenden Übungsumgebungen an, in denen diese sicherheitsrelevante Aufgaben auf (virtuellen) Maschinen lösen müssen und dabei mit Sicherheitswerkzeugen in einer kontrollierten Umgebung experimentieren können. Studierende erreichen diese Übungsumgebung über das Internet mit Hilfe von Secure Shell (SSH). Die Übungsumgebung enthält für Lehrzwecke absichtlich angreifbare Dienste und Programme, allerdings dürfen Studierende die Umgebung nicht missbrauchen, indem sie zum Beispiel die Umgebung selbst oder das Internet darüber angreifen. Das Ziel dieser Diplomarbeit ist die Erstellung eines Aktivitätsauditkonzeptes, mit dessen Hilfe die Lehrveranstaltungsleitung Missbrauch erkennen und zu einem Angreifer zurückverfolgen kann. Um dieses Ziel zu erreichen, werden die Anforderungen an eine Aktivitätsauditlösung mit Hilfe von Expertenbefragungen sowie einer Bedrohungsanalyse und einem darauf aufbauendem Risikomanagement ermittelt. Weiters wird eine Literaturrecherche durchgeführt, um das Anforderungsprofil zu ergänzen.

Das identifizierte Anforderungsprofil wird mit publizierten Lösungen abgeglichen und auf Basis des gewonnenen Gesamtbildes ein adäquates Lösungskonzept erstellt. Dieses Konzept wird als Prototyp implementiert und diese Implementierung wird anschließend evaluiert und getestet, um zu prüfen, ob die Anforderungen tatsächlich erfüllt werden. Kernelement des Konzepts ist die ausnahmslose Aufzeichnung aller Aktivitäten durch Protokollierung sämtlicher Eingabe- und Ausgabedaten, die mittels SSH übertragen wurden. Die daraus entstehenden Aktivitätslogs können anschließend forensisch untersucht und zum Erkenntnisgewinn zusätzlich auch neuerlich abgespielt werden. Schlußendlich wird in der Arbeit dargestellt, ob und inwieweit die konzipierte Lösung für den Einsatz in anderen Umgebungen geeignet ist.

Schlüsselwörter

IT-Sicherheit, Aktivitätsaudit, Log Management, Logging, Open Source Software

Abstract

Activity auditing is the practice of recording activities on a system and later analysing them regarding abuse of the system or for unauthorized activity. Being able to audit a system is also necessary to comply with certain regulations and certifications that restrict system and information usage. An auditor can use the audit log data to verify that the organisation's systems, and the information that is stored on them, were used in accordance with the requirements of the relevant regulations. By proving such compliance, auditing not only allows detection of abuse, but also allows the organisation to prove their accountability by showing that they adhere to strict standards.

An example where auditing system usage is useful can be found in exercise environments at universities. Various security courses provide exercises where students can try security related tasks on (virtual) machines and experiment with security tools in a controlled environment. Students reach this environment from the internet by using Secure Shell (SSH). This environment may deliberately contain vulnerable services or software for teaching purposes, but students are not allowed to misuse the environment by attacking it or other hosts on the internet. The purpose of this thesis is to develop an activity auditing concept that allows the course administration to track abuse of the environment back to an attacker. To achieve this goal, in this thesis expert interviews are used, threat modelling techniques and risk management methods to determine the requirements for an activity auditing solution. Further, a literature review is performed to supplement the requirements profile.

The identified requirements profile is compared with published solutions and, based on the obtained overall picture, an adequate solution concept is created. This concept is then implemented as a proof of concept implementation. The implementation is evaluated and tested to show that the identified requirements are fulfilled. A central element of the concept is the recording of all activities without exception by logging all in- and output data that is being transfered via Secure Shell (SSH). The concept records all student activity by recording all in- and output data sent over the encrypted SSH connection. The resulting activity audit logs can then be forensically examined and they can be replayed for additional insights. Finally, the work shows if and to what extent the solution concept is fit for use in different environments.

Keywords

IT Security, Activity Auditing, Log Management, Logging, Open Source Software

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Many systems are used by multiple users or trusted to handle important data. In such systems it is often important or even necessary to establish trust in the correct operation of the systems and control what is happening on them. Chuvakin and Schmidt[22] argue that activity auditing is a great way to gain such desired insight into a system.

1.1 Problem Description

ISO/IEC[50] explain that auditing refers to a process that uses audit evidence and evaluates it objectively to determine if and to which extend predefined audit criteria are satisfied. Chuvakin and Schmidt[22] explain that, on one hand, auditing is required by certain certifications and regulations, such as Payment Card Industry Data Security Standard (PCI-DSS), Health Insurance Portability and Accountability Act (HIPPA), Sarbanes-Oxley Act (SOX), the ISO 27000 standards family [50], and ITIL. These regulations put restrictions on how information may be used and how it must be protected from misuse. Verifying that an organisation complies with such regulations is done by conducting an audit of the organisation [22, 50].

On the other hand, Chuvakin and Schmidt[22] explain that audit evidence can be used for other purposes as well. For example, audit logs, which are a form of audit evidence, are very important for forensic analysis of system abuse [22]. Without a well-developed audit solution, there may not be any information as to what a user did on a system, especially if that user manages to gain administrative privileges. Thus, the problem of auditing is not only of concern to executive management that need to satisfy compliance requirements, but it is also important for system administrators that need to be able to protect their systems. Without sufficient auditing capabilities they may be unable to determine what happened during an attack on their systems and neither can prevent the attack from happening again [22].

An example where auditing system usage is useful can be found in exercise environments at universities. Various security courses provide exercises where students can try security related tasks on (virtual) machines. There they can experiment with various tools and explore their capabilities in a controlled environment. This also includes performing attacks against certain services and trying to exploit them in some way as part of their course assignments. Students can reach this exercise environment via Secure Shell (SSH) over the internet. This allows them to work on their exercises from home or from inside the university network. However, students are only allowed to attack the services in the exercise environment in specific ways. They are not allowed to attack the exercise environment itself or other hosts on the internet and doing so is considered abuse of the exercise environment.

1.2 Expected Results

The purpose of this thesis is to develop an activity auditing concept that allows the course administration to address abuse of the exercise environment. In case of abuse, the course administration needs proof which student performed the attack. Such proof can be obtained by using an activity auditing solution that records the activities of users on a system. If an incident occurs, these audit logs can be analysed to determine how it happened and who is responsible.

While the first step towards activity auditing is logging of actions, simply logging user's actions to a log file is not sufficient to satisfy the requirements of the course administration. Rather, one has to build a comprehensive log management concept that ensures that, for example, students are not able to tamper with the log files. Thus, auditing of actions in a system also touches multiple areas of IT security such as confidentiality, integrity, availability, dependability, and authenticity of the audit log data.

This thesis develops a concept that shows how activity logging and auditing can be implemented in a security exercise context by using open source software. The concept focuses on open source solutions since those allow for customization by the course administration and reduce dependencies on third parties.

The objective of this thesis is to answer the following research questions:

- **Research Question 1**: What are the requirements for and the threat model of an auditing solution in the context of a security lecture exercise environment?
- **Research Question 2**: Is it possible to define a technical concept that satisfies all requirements and can be implemented efficiently?
- **Research Question 3**: How can the audit data be evaluated to detect misuse of the exercise environment and is it possible to integrate automatic auditing solutions into this concept considering the requirements?

1.3 Methodological Approach

To find an answer to the before mentioned research questions, the thesis makes use of software engineering methodologies:

1. The first step is the collection of requirements. To determine the requirements that the solution needs to satisfy, multiple methodological approaches are employed. These include conducting expert interviews with key stakeholders, the creation and analysis of a threat model, performing a risk analysis, and performing a literature review to determine if others have discovered additional requirements.
2. Once the requirements haven been determined, a concept that satisfies them is created. It can not only be used in the example context, but with small modifications it can also be used to comply with regulation and certification requirements. The concept combines existing solutions, discovered by literature review, and describes new ones if they are necessary to reach the goal. First, it looks at the basics of the environment in which the system shall be used. Next, it investigates how audit log data can be collected, and finally, it describes how the collected data can be analysed.
3. This concept is then implemented to provide a proof of concept implementation. In accordance with the determined requirements, the implementation reuses existing software solutions such as the SSH daemon and the system log service. Integration code between the different parts of the solution is written in a scripting language, such as bash or python.
4. The prototype is tested to prove that it does indeed satisfy the requirements. Tests include unit tests of newly created components where necessary, and system tests. System tests include an evaluation of the performance overhead of the solution, verification of fail-safety, and verification of log content.

Towards this goal, Chapter 2 describes the foundations of IT security, threat modelling and risk management, as well as security aspects of open source software and SSH. Afterwards, Chapter 3 discusses the motivation, goals, requirements, types and security considerations of logging in general. Chapter 4 investigates activity auditing and its specialized nuances in more detail. Chapter 5 describes the example exercise environment for which Chapter 6 provides an activity auditing concept and discussion about its implementation. Finally, Chapter 7 draws a conclusion based on the research questions specified in this section.

2 Foundations of IT Security

An important issue when dealing with IT systems, especially in the context of logs and audit trails which may include sensitive data in general, is that of ensuring IT security. Anderson[2] explains that IT security is a term that is rather difficult to define. The most common definitions describe Confidentiality, Integrity and Availability (CIA), but security can also be defined as the balance between risk and controls that reduce that risk [2]. Anderson[2] defines it as „A well-informed sense of assurance that information risks and controls are in balance“. He claims that this is far better suited for the corporate world because a CEO knows when he is well-informed and has sufficient assurance, but it is rather difficult to determine good metrics for a CIA approach.

2.1 IT Security

The CIA triple as defined by Guttman and Roback[42] in the National Institute of Standards and Technology (NIST) handbook on computer security is one of the first, if not the first, definitions of Computer Security (or IT security). It defines IT security as a process that requires continuous monitoring and management and ensures confidentiality, integrity and availability of information system resources. System resources include hardware, software and data. More recent work, like that of Parker[78] and Mellado and Rosado[65], lists additional elements or even uses an entirely different definition, while a newer version of the NIST handbook, now by Nieles, Dempsey, and Pillitteri[70], still uses the original CIA definition without extensions. Parker[78] defines IT security via availability, utility, integrity, authenticity, confidentiality, and possession. He believes that the CIA definition's parts would have to be redefined to include a broader range of threats and doing so would make them more difficult to understand. Therefore, he chose to use different terms to avoid confusion. Mellado and Rosado[65] describe Information Systems Security (here called IT security) as a process that tries to establish security policies, and the resulting procedures and control elements, over information assets. The goal of this process is to ensure the confidentiality, integrity, availability, and authenticity of the protected information.

For the purpose of this thesis, the original CIA definition is used and extended to include dependability and authenticity. Sections 2.1.1 to 2.1.5 define each of these terms in detail. Dependability and authenticity are used because, as explained by Oram and Viega[77], audit logs aim at providing traceable, reliable information to determine who performed certain actions on a system. Without ensuring authenticity of the log data, it is impossible to guarantee this since audit logs may be faked and thus result in incorrect conclusions. Similarly, if the logging system is not dependable, it may not work correctly and thus audit logs for some actions may be missing and, therefore, those actions cannot be audited.

Furthermore, an approach to IT security based upon risk is also described in Section 2.3 because it applies better to a real world scenario. The CIA definition helps to find possible attack vectors, but does not associate them with any kind of weight. It considers all vectors equally important and strives to prevent all of them even though this is hardly always possible because in a real world setting they sometimes contradict themselves. Barker[8] notes that, for example, using encryption reduces availability because the key needs to be available to read the data, yet it raises confidentiality. If encryption is used it, therefore, becomes necessary to ensure the continued availability of the key. Additionally, it is also necessary to ensure the confidentiality and integrity of the key. He further argues that in certain situations non-cryptographic protection may be more

suitable. Since confidentiality and availability both are security goals, either decision to use or not to use encryption potentially harms the other goal. With the addition of a risk based approach it is possible to determine the significance of each attack vector and figure out which ones threaten the system in question the most, thereby allowing to clarify which and how many attack vectors need to be addressed to gain sufficient security [70].

2.1.1 Confidentiality

Confidentiality means that only certain users are allowed to access specific information [65]. For example, in a sales environment only billing staff may access the billing history of a particular customer while shipping staff may only access the shipping address and the list of ordered goods. Garfinkel, Schwartz, and Spafford[34] note that this not only means that the complete set of information should be protected, but also individual pieces that may seem harmless on their own. These pieces could become harmful if combined with information from other sources. With information like a telephone number, a postal address, the birthday or credit card number it might be easily possible to impersonate someone and convince support staff to reveal additional information or perform restricted actions like resetting a password.

Confidentiality can be ensured by restricting access to information to certain authenticated users and by using cryptography to protect data when it is stored or sent elsewhere [70]. For example, Dierks and Rescorla[24] explain how cryptography can be used to protect the confidentiality of credentials when authenticating users.

Examples

Data Confidentiality Encryption like Advanced Encryption Standard (AES) can be used to ensure confidentiality for stored or transmitted data. Cryptographic algorithms like AES work by using substitution, transposition and mathematical functions on the message so that the message content will read like gibberish after it has passed through the algorithm. It can then only be read again if the symmetric/private key is known and the operation is reversed [34].

Password Confidentiality Passwords should not be stored or sent in plain text because doing so would allow an attacker to either attack the server and extract the stored data or intercept the password during transit [34]. Dierks and Rescorla[24] explain that nowadays network traffic can be readily encrypted by using a library that implements the Transport Layer Security (TLS) protocol which is a collection of cryptographic algorithms that „prevent eavesdropping, tampering, or message forgery“ and thus it is easy to protect password information in transit. Additionally, it is important to protect stored credentials against leaks since an attacker that gains knowledge of a password might be able to impersonate a user, either on the system where they obtained the password, or on another system where the user reused the password [34]. One way to protect passwords is by only storing hashes of passwords rather than the plain text password [34]. However, hashes do not protect against brute force attacks, they just make attacks more difficult and time consuming. Some algorithms also support deliberately slowing down the calculation of the hash. Currently algorithms that allow this kind of behaviour include PBKDF2, Bcrypt and Scrypt. Chang et al.[18] explain that those algorithms use lots of CPU time or memory to make it more difficult to compute many hashes in parallel or generally in a short amount of time. By using such algorithms, brute force attacks can be made more costly and thus they pose less of a threat. On the other hand, hashes alone cannot protect users that use the same password since a hash function always outputs the same hash for the same input.

This determinism of a hash functions obviously leads to the problem that an attacker could look at the database dump and decide whether they want to attack more common or less common

passwords first. To prevent an attacker from seeing identical hashes for identical passwords a salt can be added to the password before hashing it. The salt does not have to be kept secret but it should be unique for each stored password because the goal is to make every password and salt combination unique. Even if many users share the same password the hashes saved in the database will all be different and an attacker cannot prioritise hashes that appear more often when performing a brute force attack. Using a salt also protects against rainbow tables that list every possible password with a certain length and its hash. These tables can be used for fast lookups even of expensive hashes. When a salt is used they can still be calculated, but since the salt is different per password the table is only valid for one specific salt value. If the salt is long enough and thus the search space is large enough this ensures that it is infeasible to create and store a rainbow table for all possible password and salt combinations even with restricted length and character sets [34].

Physical Security of Storage Media Physical security of the storage media also needs to be considered, especially when the media are transported and might be lost or stolen. The likely most secure way of protecting data's confidentiality is to encrypt it before it is stored on the media. Some devices provide encryption support in hardware, but nowadays it is easy and fast enough for most workloads to perform encryption in software. Thus, it is possible to encrypt data independently of the storage media's own encryption support. However, it is important that the decryption key is stored securely because if it is damaged or becomes unavailable, the encrypted data cannot be restored. If data is not encrypted, then the only protection of the data on the media is the media's physical security since file system permissions or user passwords are only validated by the running system [34].

2.1.2 Integrity

According to Mellado and Rosado[65], the principle of integrity says that information will not be „modified by third parties“ and in general that „correctness and completeness“ is ensured. However, they do not define „third parties“ and, therefore, it is unclear if this definition includes factors such as damaged hardware, which may result in incorrect data. Incorrect data has clearly lost its correctness property, which means that this data no longer possesses integrity. Therefore, it should arguably be included in this definition.

Barker[8] provides a broader and clearer definition, saying that integrity means that data is not „modified in an unauthorized manner since it was created, transmitted or stored“. He further explains that such modification can be accidental or deliberate. Accidental modification includes transmissions errors and storage hardware failure, while deliberate modification is performed by an adversary.

However, in general the term integrity is not clearly defined. Garfinkel, Schwartz, and Spafford[34] use integrity in a broader sense that includes the fact that a system should protect its data and programs from being deleted or altered without permission of the owner. Parker[78] considers it a loss of integrity if a distributor sells a DVD with software from a publisher, but removes the name of that publisher from the DVD even though he does not alter the software itself.

Guttman and Roback[42] explain that integrity means that information has to be timely, complete, accurate and consistent. However, they also note that this is difficult or impossible to achieve in computing and thus they further define integrity via „data integrity“ and „system integrity“. Data integrity means that „information and programs are changed only in a specified and authorized manner“. System integrity means that a system „performs its intended function in an unimpaired manner, free from deliberate or inadvertent unauthorized manipulation of the system“. In the revised version of this document, Nieles, Dempsey, and Pillitteri[70] provide a new, combined definition for integrity stating that it guards against „improper information modification or de-

struction“ and ensures „information non-repudiation and authenticity“. They also continue to use the terms „data integrity“ and „system integrity“ with a similar meaning as before.

For the purpose of this thesis, data is considered to possess integrity if it has not been altered without authorisation, if it is complete and if it matches the data that was saved and is thus accurate. The same definition also applies to programs since these are just a special form of data.

Examples

Data Storage Integrity Verification Chang et al.[19] explain how NASA's Jet Propulsion Laboratory (JPL) created a backup concept using cloud storage to store encrypted data backups for the Mars Exploration Rover and be able to restore them quickly when necessary. They use the AES algorithm to ensure data confidentiality, but they do not report on whether they take measures to ensure data integrity apart from an audit procedure that compares the data between the primary and secondary backup storage. Their system keeps one local backup copy and one copy in the Amazon S3 cloud storage service.

To audit their remote backup, Chang et al.[19] use MD5 to create a list of hashes of the files stored on S3 and compare that list with one created in parallel from the locally stored backup [19]. They do not provide an explanation as to why they create two hashes in parallel rather than simply comparing, bit by bit, the data used to create the hashes. They claim that this audit process ensures data integrity of the remote backup, but they also note that it simply compares the created hashes of files from both storage locations. Thus if the data is modified by an attacker in both locations in the same way, the hashes will still match, but the integrity of the file will still be lost because integrity means that the data is explicitly not modified by an unauthorized third party.

Digital Signatures Nieles, Dempsey, and Pillitteri[70] explain that digital signatures can be used to ensure the integrity of a piece of data, such as a document or a log file. Symmetric approaches use a shared secret key which allows anyone with knowledge of that key to create and verify signatures. They note that this assumes that all parties involved must trust each other since any party can perform any function. This problem is further discussed in Section 2.1.5.

Asymmetric approaches use private and public keys which do not require this level of trust and allow to either create or verify a signature, but not both. The private key can create a signature, while the public key can verify that a signature is valid and has been created by the corresponding private key [70]. According to Schneier[92], digital signatures allow to ensure that the private key is only available to the entity that shall be able to sign a message, while the public key can be distributed safely to anyone who needs to verify the signatures.

Collision Resistance Rogaway and Shrimpton[87] explain that collision resistance means that it is computationally infeasible to find two inputs that produce the same hash as an output. They also explain a related concept called second-preimage resistance, which means that it is infeasible to find a second item with the same hash as a given item, but with different content. Schneier[92] clarifies that this means that an attacker cannot easily change the content of a file and ensure that the hash of the changed file matches that of the original one. This is important because a hash function, like MD5, may be used in a digital signature algorithm. According to Turner and Chen[106], MD5 is known to be vulnerable to various collision attacks and should, therefore, in general not be used when collision resistance is desired, especially in new applications. However, MD5 is good enough if the goal is to detect unintentional changes of data caused by e.g. transmission errors, but to determine if this is the case, the purpose of use has to be stated clearly [106].

Sasaki and Aoki[91] describe an attack on MD5 that can generate a preimage with a complexity of $2^{123.4}$. A preimage attack is an attack that finds any input which hashes to a preselected output hash [87]. A brute force attack that tries every possible input has an average complexity of 2^{128} since the output of the MD5 function is a 128bit hash [91]. Turner and Chen[106] believe that even this reduced preimage complexity of $2^{123.4}$ is still sufficiently high to consider MD5 preimage resistant.

Access Control Nieles, Dempsey, and Pillitteri[70] explain that access control is the process of granting or denying access requests. Physical access control is used for buildings and restricts who may enter a building or room. Logical access control is a function of an operating system or application and performs similar control in the electronic world. An operating system may allow to limit which users can read, write or delete a file. This is referred to as „file system permissions“ in this thesis. Databases and networked applications may provide similar functionality and only allow certain users to access or add data. These controls may be implemented either internally in the application itself, or they may be provided by external devices [70].

Access control allows isolating parts of a system from each other and preventing unauthorised actions from taking place [70]. Since the definition of integrity includes unauthorized modification as a threat, access controls can also be used to prevent these and, consequently, they can help ensuring integrity. Similarly, access controls can also help ensuring confidentiality by restricting access to authorized users only.

2.1.3 Availability

As the name suggests, availability means that information is readily available when it is needed [65]. This means that, for example, the backup data stored on a storage system will still be readable when it is necessary to restore the system months or years after the data has been stored.

Garfinkel, Schwartz, and Spafford[34] state that unavailability of information can be just as bad as if the information was deleted. Reasons for unavailability may include services being degraded or made unavailable without authorisation. Parker[78] gives a similar example of a rejected programmer who removed the file name of an important file. Doing so the programmer rendered bank staff unable to find and use the file's content even though only the name was removed, but the content was still there. Parker[78] also highlights the importance of having backups and multiple ways to access the same content by using programs that can search for the content directly without requiring a file name.

Example: Multiple Copies Kent and Souppaya[53] note that a solution towards increasing the availability of logs is maintaining multiple copies of the log data. These should be stored in separate locations to ensure that data is still available if one copy is damaged or destroyed. Section 3.4.4 further discusses secure log data storage and Redundant Array of Independent Disks (RAID) storage solutions.

2.1.4 Dependability

Dependability is generally seen as a combination of multiple concepts. Avizienis et al.[4] define it as including the following attributes:

- Availability: Being able to provide service. See Section 2.1.3.
- Reliability: Providing correct service and continuing to do so.
- Safety: Not incurring catastrophic consequences for the user or the environment.

- Integrity: Absence of improper modifications. See Section 2.1.2.
- Maintainability: Ability to modify and repair the system.

Avizienis et al.[4] note that confidentiality is prominently absent in this definition. This is because they consider confidentiality to be an attribute of „security“, which further contains availability and integrity. Since dependability and security both contain availability and integrity, they differentiate as follows: Their definition of „dependability“ focuses on providing a system that is able to stay within acceptable limits regarding service failures. In the context of dependability, integrity considers all improper modifications equally [4]. In this case „improper“ includes unauthorized modifications, but also hardware and software failures that either change the system state or prevent an authorized user from (correctly) changing the system state. On the other hand, their definition of „security“ focuses on preventing only unauthorised access to information of the system [4]. Security also contains the integrity attribute, but the focus is put on the unauthorized access portion of integrity. They note that this distinction between dependability and security is drawn because in practice it helps to clarify how dependability and security are balanced in a system [4].

Examples

Trust Kochs[56] explains that dependability is important because it helps to avoid creating unreliable systems that result in economic disaster or huge penalties. He notes that poor dependability can cause loss of trust in the system and the organisation behind it.

This notion of trust can also be found in Avizienis et al.[4]. However, they start with the notion of dependence between two systems A and B. Dependence is the extent to which the dependability of system A is affected by system B's dependability. This dependence can then be used to define an „accepted dependence“ which expresses the trust placed in a specific system.

Avizienis et al.[4] further note that trust in a system requires correct service provided by the system. Correct service is provided when the system correctly implements the system function. If this is not the case, a service failure occurs. A failure is the „loss of ability to perform as required“ [56]. This means that the external state of a system is incorrect. The external state is the state that is visible to the user of the system via the user interface [4]. In this case, a user can be another hardware or software system, a human, or any other entity that interacts with the system. For example, a failure of a web service would be an incorrect reply to a request, with that incorrect reply being the external state of the web service and whoever sent the request being the user of the system.

Faults, Errors, Failures The source of a failure is called an error, which in turn is caused by a fault. A fault is the basic inability to perform a function correctly. However, just because a fault exists in a system that does not automatically mean that this fault results in an error or even a failure. The fault only leads to an error if that particular functionality is used in a way that triggers the error. If that happens, an error occurred. Similarly, an error does not yet lead to a failure. A failure only occurs if the error is activated and the incorrect, internal state reaches the external state of the system. Thus, fault tolerance and error detection capabilities of a system help to prevent a fault from becoming an error and an error from becoming a failure [4]. Kochs[56] notes that a fault is the result of another failure either of the system itself or of a failure in an earlier stage of that system's life cycle such as specification, design, implementation or maintenance.

2.1.5 Authenticity

Mellado and Rosado[65] define authenticity as allowing „trustful operations by guaranteeing that the handler of information is whoever s/he claims to be“. Avizienis et al.[4] explain that authenticity is a secondary attribute, which means it is a specialized or refined version of a primary attribute, such as integrity. They define authenticity as the integrity of the data itself, as well as the origin of the data and optionally the time of emission of the data. Thus, data is authentic when the creator can be verified and the data possesses integrity [4].

Nieles, Dempsey, and Pillitteri[70] refer to authenticity in their definition of integrity and note that integrity is used to ensure „information non-repudiation and authenticity“. However, they do not provide a definition of authenticity itself even though they also discuss user and message authentication approaches. These approaches are also discussed later in this section.

Authentication is often confused with authorisation, however, while these two concepts are related, they are not the same. Authenticity is ensured by performing authentication. Lowe[61] explains that authentication means verifying the validity of an identification and thus being sure of a user's identity. An identification is a claim provided by a user to a system about their identity [42].

Examples

User Authentication A user or an automated system presents some form of credentials which allow another system to confirm it as a legitimate user and authenticate the presented identity. This information can be used further, to permit, or authorise, specific actions. Thus, using a confirmed identity and tying it to certain permissions is called authorisation [42].

User authentication is often performed with a, potentially public, user name and secret password. In this case, the user name is the identification. Authentication of this identification can only be performed by a system that has the capability to verify that the user possesses the password. If the user initially provides the password to the authenticator system, the system can store it. Later it can verify that the user knows the private information and thus validate the identification claim. A drawback of this solution is that the authenticator also knows the passwords of all users, but this can by resolved by using more advanced methods to provide password confidentiality as discussed in Section 2.1.1. The system may also use more complex methods than a simple password, such as public key cryptography which is discussed in Section 2.5.3 . The general idea of user authentication is thus to prove access to information that is either secret or very difficult to obtain [42].

Message Authentication However, in the case of authenticating messages, the situation is more difficult. There, cryptography can be used to calculate a Message Authentication Code (MAC). This MAC takes as input, a secret key and the message, and outputs a hash code. This hash code is then sent, along with the message, to the recipient. To verify a message, the verifier also needs the secret key and then simply needs to perform the same operation again. If the calculated hash is identical to the code sent with the message, the verifier can assume that the message has not been modified by a third party [42]. However, requiring that both, the sender and the recipient, have the same key presents multiple problems. Shostack[96] explains that, nobody can verify who created a MAC because the recipient can calculate the same MAC. Therefore, the MAC only asserts that someone, who had the secret key, created the MAC for a given input. Furthermore, all parties that want to verify the MAC require the secret key. This presents a problem when a message is supposed to be verified by a large group that is not supposed to be able to create valid MACs themselves [96]. Luckily, digital signatures solve this problem by using public key cryptography as discussed in Section 2.1.2.

2.2 Threat Modelling

Myagmar, Lee, and Yurcik[68] explain that creating complex yet secure systems is difficult because the bigger a system is, the more difficult it is to instinctively discover possible threats towards it. Threat modelling provides methods and a structured way to approach this problem effectively and thus allows the authors to deal with problems of any size. However, threat modelling itself does not address the discovered problems or prioritise them. Rather, it aims to find as many threats as possible, regardless of their probability of occurrence [68]. Addressing threats is done by performing risk management, which is described in Section 2.3.

2.2.1 Definition of Threat Modelling

According to Myagmar, Lee, and Yurcik[68] threat modelling describes the process of enumerating the threats to a system. Using the threat model, system architects can develop meaningful and realistic security requirements that mitigate these threats. Myagmar, Lee, and Yurcik[68] state that systems can be attacked in a variety of different ways and system designers often try to find possible vectors by performing brainstorming. Such an approach is not reproducible and, therefore, „likely to leave large portions of the attack space uninvestigated“ [68]. A much better solution is to use a systematic approach to ensure that as many threats as possible are discovered by the system developers.

Shostack[96] defines threat modelling as „the use of abstractions to aid in thinking about risks“. For him a threat model consists of a model of what is being built and a model of the threats. A good model should help the user in looking at the big picture and see groups or classes of attacks rather than individual attacks. For example, one should think about operator errors or confidentiality threats in general instead of looking at specific instances of these problems and ignoring very similar ones in other places in the project.

Shostack[96] uses a four step threat modelling framework. First he creates a model of the system that needs to be protected. Then he looks for threats, tries to address these threats and validates the result. In this thesis, threat modelling is only concerned with modelling the system and finding threats. How these threats are addressed is part of risk management which is discussed in Section 2.3.

Myagmar, Lee, and Yurcik[68] describe threat modelling as a process that determines all assets that need to be protected, identifies their access points and then finds threats that target those access points. How these steps are executed depends on the specific type of system. They suggest to model an application by using data flow diagrams, while they suggest a network model for networked systems. Refer to Section 2.2.3 for a description of these modelling methods as well as identification and threat finding methods. Myagmar, Lee, and Yurcik[68] further stress the fact that to identify assets one has to understand the exact system that is being protected. It is thus impossible to provide a general exhaustive list of threats because no two systems are equal.

2.2.2 Reasons for Threat Modelling

Shostack[96] notes that there are various reasons for threat modelling depending on the goal. Threat models can be created even before the project is being implemented. Thus, any issues discovered with the model can be used to improve the project in the planning stage where changes are relatively cheap. In later stages it may not be practical to properly address an issue and instead workarounds might be implemented. While they may work to some degree, Shostack[96] argues that it is more appropriate to deal with them at the beginning, since later changes become more expensive.

Another benefit of threat modelling is a more consistent set of requirements and features that address these requirements. A good threat model allows an engineer to determine if a security requirement fits together with the rest of the security features of the project. Without a threat model, mitigations for some threats may be missing, but customers may not know about this and assume that the product offers a certain level of general protection or they may extrapolate from those mitigations they know about. Shostack[96] provides the example of a house with varying protection levels of the windows and walls. The walls may be made of wood or brick and there can be various levels of protection for the windows. Windows can be made of normal or reinforced glass, and they may be protected by an alarm. This alarm may have a heartbeat to detect when an intruder cuts the wire to the alarm. It may further use cryptographic methods to protect the heartbeat against a fake signal sent by an intruder to convince the system that the alarm has not been cut. However all these mitigations fail to consider that there is a key for the front door under the mattress. A customer that knows about such comprehensive mitigations applied to the windows, may assume that other parts of the system are protected similarly.

This example shows how many potential mitigations can be applied to a single initial issue and how many more may be needed to address limitations of the applied mitigations. Shostack[96] notes that in a real world situation, implementing mitigations is time-consuming and costly. Addressing all threats is not possible since resources are limited. With a threat model it is possible to obtain a comprehensive overview over most threats towards the system. This can be used to provide a consistent level of protection that addresses all threats equally well instead of focusing too much on a small subset and ignoring the big picture [96]. This prioritization can be performed with a risk based approach, which is discussed in Section 2.3.

2.2.3 Threat Modelling Methods

System Modelling Myagmar, Lee, and Yurcik[68] explain that threat modelling first starts by modelling the system that has to be protected. For an application, they suggest data flow diagrams. These diagrams characterize systems by showing how information can get into the system and how it is processed [68]. Shostack[96] also uses a data flow diagram to model the system.

For networked systems, Myagmar, Lee, and Yurcik[68] suggest a network model. This network model allows them to examine communication between machines in a network. It starts by identifying the roles and functions of each class of computers in a network. Afterwards, it maps the communication patterns between the different roles. This mapping describes the protocols, ports, and traffic patterns for each communication pattern [68].

Identifying Assets and Access Points The model of the system is then used to identify assets and access points. Assets are any abstract or concrete resources of the system that must be protected from misuse. Examples for assets include processes and data, but also abstract concepts like data consistency. To gain access to an asset, and thus to attack it, an attacker uses an access point. For example, network sockets, Remote Procedure Call (RPC) interfaces, configuration files, hardware ports, and files can all be access points [68].

While Myagmar, Lee, and Yurcik[68] also mention trust boundaries, they do not use them nor do they explain why they are important. Shostack[96] explains that parts of the model are likely to be controlled by different entities. The parts that are controlled by the same entity are said to be within the same trust boundary. For example, Figure 2.1 shows a diagram for a simple web application which includes a web browser, web server and a database. There is a trust boundary around the web browser and around the web server and database. The web browser is controlled by a user or an attacker, while the web server is controller by the organisation providing the web application. The database may also be inside this organisation's trust boundary as shown in the

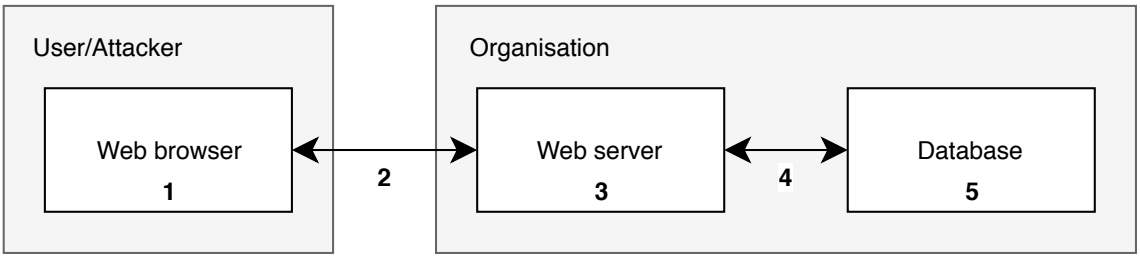

Figure 2.1: Simple diagram of a web application showing trust boundaries and data flow between different parts of the web application. Based on: Shostack[96]

diagram, or it may be outsourced to a different organisation and thus be in it's own trust boundary. Trust boundaries are important because the threats discovered that cross these boundaries are likely to be important threats. Thus, trust boundaries provide a good place to start at when it is time to find threats [96].

Shostack[96] also suggest to number each process, data flow, and data store as shown in Figure 2.1. He explains that in larger and more complex diagrams it may become easy to miss parts of the diagram or be confused by labels, especially if labels occur multiple times. Trust boundaries are not numbered because they receive descriptive, unique names.

Furthermore, Shostack[96] notes that the diagram may be incomplete and that further use may reveal that it is missing key parts, such as additional databases or data flows. If such issues arise, the diagram should be extended. It may also be necessary to move parts of the system to dedicated diagrams to prevent each one from becoming overcrowded and unreadable. Finally, the goal of a diagram is to help analysts understand the system and help them reason and think about it [96].

Finding Threats Finding threats is „thinking of things that might go wrong“ [96]. Myagmar, Lee, and Yurcik[68] explain that the basic process for this is to step through the created model and look for possible threats towards each of the desired security properties of the system. In the case of this thesis, Section 2.1 defines IT security as ensuring confidentiality, integrity, availability, dependability, and authenticity. Threats towards these properties can then be identified by creating a hypothesis that violates any one of the properties for a given asset [68]. For example, a threat towards integrity might be that an attacker can manipulate a log file.

It may prove useful to start with a list of known, common threats and look for instances of these in the system. However, this typically only results in these common threats being found and system-specific threats may be missed without further analysis [68].

Myagmar, Lee, and Yurcik[68] note that it is helpful to look for threats based on their effect. Effects of potential threats are provided by the STRIDE mnemonic, which is described later in this section [68, 96]. Shostack[96] clarifies that other methods can also be used. The goal of STRIDE is only to help analysts simplify finding threats by focusing their brainstorming process [96].

STRIDE

To find threats, Shostack[96] and Torr[105] use an approach which is called STRIDE. STRIDE is a mnemonic for the terms Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privileges [96, 105]:

- **Spoofing**: Pretending to be someone or something else.
- **Tampering**: Modifying something one is not supposed to (be able to) modify.

- **Repudiation**: Performing an action that can not be traced back to the actor that performed it.
- **Information Disclosure**: Exposing information where it should not be exposed or to someone who should not be able to see it.
- **Denial of Service**: Preventing service to users by various means such as crashing a system or making it unusably slow.
- **Elevation of Privilege**: Obtaining more permissions than one is authorised to have, respectively performing actions one is not authorised to perform.

The application of STRIDE in the threat modelling process is simple. For each part of the model, „consider each of the STRIDE categories and how they might apply to the object and accompanying data flow“ [105].

2.3 Risk

Once threats have been identified using threat modelling, is becomes necessary to deal with them. However, it is rarely possible to apply countermeasure for every identified threat and in some cases, countermeasures work against each other just as requirements sometimes contradict themselves. Additionally, requirements may also interact with potential countermeasures for threats, which results in a complex combination of requirements and countermeasures that must be carefully balanced [96]. Garfinkel, Schwartz, and Spafford[34] warn that perfect security can never be achieved because only the risk of threats can be reduced, but it is impossible to reduce that risk to zero. One example for this dilemma has already been mentioned in Section 2.1 regarding encryption and key availability. Risk management provides the means to identify risks towards a project, business venture or even a software system, and analyse and manage them.

It is important to note that since perfect security is impossible there are always trade-offs to be made. How those look in practice is something that everyone has to figure out for themselves, but the definition by Anderson[2] given at the beginning of Section 2, which calls for a well-informed sense of assurance that risks and controls are in balance, provides a good guideline for such decisions.

Shostack[96] also argues that threats and countermeasures should not be viewed as static. Instead, they dynamically change as the environment changes. For example, an attacker may change their strategy in response to a countermeasure, which may prompt the organisation to implement additional countermeasures. Thus, risk management is a process and the threat situation must be monitored and reevaluated constantly [96].

2.3.1 Definition of Risk

The term „risk“ has multiple meanings depending on context. According to Heckmann, Comes, and Nickel[47] it originates from the Greek word „rhizikon“ which means roughly avoiding „difficulties at sea“. Later, when trade over the seas became common-place the meaning changed to the threat of losing a ship for reasons such as storms, pirates or illness. These considerations are closely related to modern „scenarios“ which are used in risk analysis to gain a better understanding of potential threats.

Dionne[26] defines a „pure risk“ as „a combination of the probability or frequency of an event and its consequences“. He further notes that the consequences of such a risk are generally negative.

Heckmann, Comes, and Nickel[47] also agree on the importance of probability. They note that in the 16th century Blaise Pascal and Pierre de Fermat tried to mathematically capture the uncertainty or „risk“ in gambling. This mathematical description was founded on probability and can still be found in modern definitions.

The ONR[1] 49000 - „Risikomanagement für Organisationen und Systeme“ (risk management for organisations and systems) standard by Austrian Standards[3] defines risk as the combination of probability of occurrence and the potential consequences, which may be positive or negative. Probability in the context of risk and risk management is a statement about the relative frequency of future events. The authors further note that it is sometimes difficult to quantify the probability of occurrence of a risk in practice. Therefore, since it is not always possible to obtain objective data, this definition also includes subjective estimation as a source for probability values. Probability can either be provided as the number of events per time period, such as „once every 100 years“, or as a ratio between desired and undesired cases, such as „10 percent“.

Bernstein[13] explains the „Grenzrisiko“, which roughly translates to „border risk“, as the highest, still acceptable risk. It is defined in the context of machines, but Fankhauser, Schanes, and Brem[32] also apply it in the context of IT systems. Threats that pose a bigger risk than allowed by this border risk must be reduced by applying suitable risk management methods and may not be accepted. After reduction, the risk of an addressed threat is reduced to the „Restrisiko“ (remaining risk). This remaining risk is not required to match exactly with the border risk value. Instead, it may be below the border risk [13].

2.3.2 Risk Identification

The first step in dealing with risks is to identify them, which can be achieved by using a variety of different methods. ONR 49000 [3] groups them into 5 areas as shown below. For readability, the methods are described later in this section.

- Creative methods: Brainstorming, Delphi method
- Scenario analysis: Fault Tree Analysis (FTA), Scenario analysis
- Indicator analysis: Critical Incident Reporting
- Function analysis: Failure Mode Effects Analysis (FMEA), Hazard and Operability Study (HAZOP)
- Statistical methods: standard deviation, confidence interval

Becker et al.[10] differentiate only two categories of methods, which are collection procedures and search methods. Although, they further divide search methods into creative methods and analytical search methods. Collection procedures include risk checklists, worker and expert consultation, while creative methods include brain storming and the Delphi method. Analytical search methods include FMEA and FTA.

Risk Checklists, Worker and Expert Consultation Risk checklists collect already known risks from previous projects and present them in a structured questionnaire. Worker and expert consultation means that knowledge carriers are asked what they believe to be risks in their domain. This allows risk analysts to also leverage subjective experience in risk identification [10].

[1] Österreichisches Normungsinstitut-Regel (Austrian Standards Institute Rule). Source: https://www.help.gv.at/Portal.Node/hlpd/public/content/256/Seite.2560005.html (visited on 2019-05-02)

Brainstorming and Delphi Method Brainstorming is a method where multiple experts discuss a predefined topic and discover risks by simply thinking about the topic and being creative. A moderator coordinates the discussion and it is important that it occurs in a supportive environment to promote creativity [10]. The Delphi method is similar, but it uses a much more rigid and systematic approach. First, experts are questioned individually and identified risks are written down. Next, the collected data is anonymised and presented to all participants for further iterative rounds of reviews and improvements [10].

Critical Incident Reporting Smith and Mahajan[97] explain that a Critical Incident Reporting system is an operational system that enables workers to report critical situations to management. Güntert[41] explains that a distinguishing feature of such a system is that it aims to reduce anxiety in workers by reducing or eliminating negative repercussions for the workers involved in the incident [41]. The goal is to let the organisation learn from incidents and near misses to prevent similar problems in the future [97].

HAZOP McDermid et al.[64] describe Hazard and Operability Study (HAZOP) as a method that analyses a system's expected behaviour. A team of analysts uses guide words to prompt their imagination to discover potential hazards if the system deviates from the expected behaviour.

FMEA and FTA Failure Mode Effects Analysis (FMEA) analyses the behaviour of a system or subsystems in case of a failure of a single component of that system. This works by starting at the cause of a risk and analysing which consequences could happen as a result. Contrary, Fault Tree Analysis (FTA) starts with a system in a faulty state and analyses which causes could lead to this specific outcome. Following this idea, Ruijters and Stoelinga[89] explain that FTA creates a tree like diagram, called „fault tree“, which shows the dependencies between components of a system. The fault tree contains events, which represent failures, and uses boolean operators to connect those events with each other to model how failures propagate through the system. If the system does not have sufficient redundancy to tolerate certain failures, the failure propagates towards the root node of the tree, which indicates a system failure [89].

2.3.3 Risk Analysis

Risk Identification alone treats all risks equally, but dealing with risks requires that their probabilities are quantified. According to Anderson[2] risk is often difficult to measure even though Garfinkel, Schwartz, and Spafford[34] argue that numbers can be obtained from industry organisations or insurance companies, but they also note that doing so is difficult work. If numbers are available, for example, because the event happens sufficiently often and occurrences are recorded by the organisation, then probabilities can be calculated. Anderson[2] explains that the probability multiplied by the expected loss gives the annual loss expectancy (ALE). The ALE can be compared to the cost of the protection to determine whether the benefit is worth the cost [2]. However, such values should be used with care since the uncertainty for some estimations can be pretty high and, therefore, predictions may be incorrect [2].

ONR 49000 [3] claims that nearly all methods that were used to identify risks can also be applied to analyse their probability. However, they note that statistical methods work best to determine the probability, while other methods, such as FMEA and scenario analysis, are better suited to determine the potential consequences of the risk.

Brauweiler[17] explains that it is necessary to classify discovered risks to allow for systematic handling, management and prioritisation. Such categorisation is often done by multiplying the probability of the risk with its potential damage or loss to calculate a „risk value“ [10, 17]. This results in a two-dimensional graph, which is called a „Risk Map“. When this graph is segmented

into a matrix, for example, a 3x3 matrix, it is called a „Risk Matrix“. Fields in such a matrix can then be assigned severity labels [10], or, as shown in Table 2.1, they can be assigned directly to actions that shall be taken [17].

	<30% probability	30-85% probability	>85% probability
Low loss	Accept	Accept and monitor	Accept and control
Medium loss	Accept and monitor	Accept and control	Control and manage
High loss	Accept and control	Control and manage	Extended management

Table 2.1: Example Risk Matrix showing which risk management methods should be applied to each risk category. Based on: Brauweiler[17]

However, simple multiplication is not the only solution to calculate a risk value. Han et al.[44] note that the ISO 27000 standard explains that the risk value can be determined using the asset, threat and vulnerability indicators. While there are multiple ways to combine these values to calculate a single risk value, they use the multiplicative method from the Chinese version of the ISO 27000 standard, which is called GB/T20984-2007. This method calculates the risk value R as $R = \sqrt{A * T * V}$. The asset indicator A describes the importance or value of a resource. Similarly, the threat indicator T rates the potential damage that a threat can cause, and the vulnerability indicator V rates the likelihood of occurrence. They assign a value from one to five to each indicator. They note that this method intentionally increases the contribution of the vulnerability indicator to the risk value because this may be preferable in certain environments.

2.3.4 Risk Management

Austrian Standards[3] defines „risk management“ as a set of processes and behaviours that aim at controlling an organisation relative to risks. Implementing risk management in an organisation leads to a „risk culture“ in the organisation [3]. Brauweiler[17] defines „risk management“ as the general idea of performing risk management, which means the use of methods for identifying, analysing and managing risks. Rather than focusing on changing the organisation, he focuses on helping an organisation to discover and deal with risks. While he also lists „creating a risk culture“ as one goal, it is only one of many goals and not the primary one. His definition thus aims more towards providing a set of tools that can be used for various purposes, rather than forcing a culture change in an organisation. However, in general, both definitions refer to dealing with risks in a project, or an organisation and its business.

Brauweiler[17] notes that risk management deals with critical situations. He states that the goal is to discover these situations in a timely manner and deal with them before they can negatively affect the business or project. ONR 49000 [3] casts a wider net and also includes emergency, crisis and continuity management into the definition of risk management. This definition is based on the belief that preventing all risks from occurring is impossible and it is thus important to be prepared for any possibility. This preparation includes planning how an organisation can recover from an event to quickly resume normal operation. Crisis management in this case deals with a coordinated handling of ongoing, but also future crises and ensures that the organisation can react quickly. Continuity management deals with the restoration of normal operation in response to an event [3].

Risk Management System A risk management system combines all the steps of risk management, namely risk identification, analysis, and management, into a continuous process across an organisation. This allows reacting to new or changed risks by implementing a Plan-Do-Check-Act (PDCA) cycle. In the „planning“ stage, the executive management of the organisation allocates

resources and assigns roles to the staff. The goal of the „do“ stage is to perform the risk management process by identifying potential risks, analysing and managing them. In the „check“ stage the organisation performs internal or external audits to determine the effectiveness of the previously performed risk management. Finally, the system is adjusted in the „act“ stage so that it can perform better in the next cycle [3].

Risk Management Methods Risk management methods aim to control either the probability of occurrence of the risk or the potential damage caused by the risk. Risks can be resolved by using multiple strategies such as avoiding the risk, reducing or controlling the effect. Methods can be grouped into risk avoidance, risk reduction, risk diversification, risk acceptance, and risk transfer [10, 41].

- Risk Avoidance: Güntert[41] defines risk avoidance as preventing a risk from occurring by avoiding the trigger. This method tries to neutralize either the probability of the risk or the negative effect. This is generally done by not conducting the action that can lead to the risk. Obviously this also means that the possible positive outcome of that action cannot happen [41]. For example, the risk of a computer system being attacked via the network can be avoided by not connecting this system to the network. However, while this neutralizes the risk by reducing the probability of occurrence to zero, it also removes the potential benefits of the network connection since there is no network connection to begin with.
- Risk Reduction: A less drastic measure is called risk reduction. It aims to reduce the probability of a risk or the potential damage, but it does not completely neutralize the risk [41]. Ways to reduce risks include technical and operational measures. For example, to reduce the risk of an attacker breaking into a network, firewalls can be used to limit the potential entry points. However, since often not all entry points can be closed and the network connection still exists, the risk is only reduced and not avoided completely.
- Risk Diversification: Risks can be diversified by splitting a risk into multiple, independent partial risks. These risks still have the same probability of occurrence, but the potential damage is reduced. For example, Chuvakin and Schmidt[22] explain that the risk of hardware failure impacting operation of a service, such as a log storage system, can be reduced by storing log data on multiple, potentially smaller systems. Each system is still vulnerable to hardware failure, but only the data stored on that particular system is affected rather than all data.
- Risk Acceptance and Transfer: Unless risks are avoided, some part of the risk will remain even after risk management measures are applied. Other risks might already be so small that mitigating them by the measures discussed above is not worth the effort. These remaining risks can either be accepted by the organisation or they can be transferred to someone else, for example, by getting insurance [68].

Example Han et al.[44] assess the information security risks of a typical digital library using the Chinese GB/T20984-2007 standard. While they do not call their work „risk management“, they essentially follow the process described in this section. First they work on identifying risks, which are then analysed to classify their potential damage. While their work mainly focuses on identification and analysis, which together are also called „risk assessment“, they also finish with some general suggestions to manage the discovered risks.

To assess risks, Han et al.[44] use multiple methods to improve their coverage as well as the objectivity of the gathered data. They conduct questionnaires in all departments of the library, inspect the systems themselves, and use software to automatically scan for potential vulnerabilities.

They discovered that using questionnaires to obtain data can prove difficult when done across a whole organisation because people are only familiar with their own department. Especially non-technical staff had difficulties understanding technical questions which prompted them to create different questionnaires for different departments. Afterwards, they catalog and consolidate the collected data and quantify the impact as well as the probability of each risk and combine these values into a „risk value“.

Using these risk values, Han et al.[44] provide some suggestions how the most important risks can be addressed. These include using secure passwords and changing them regularly, updating all software regularly, storing backup data outside the library, physically protecting important devices like servers, and managing removal storage media usage. They also suggest that staff be trained in security awareness, especially regarding usage and maintenance of the used devices. Finally, they note that while the results have been well-received, it is important to establish an ongoing process that continually monitors the library and addresses new or changing risks.

2.4 Security Aspects of Open Source

Open Source Software (OSS) describes software for which the source code is available for anyone to read and probably the most prominent example of OSS is the Linux operating system kernel. Linux is commonly called „Open Source“, although it is important to note that „Open Source“ on its own only really means that the source code is available to be read, not that it may be modified or even redistributed. To avoid such ambiguity the term Free Libre Open Source Software (FLOSS) can be used.

The value of FLOSS lies in its unique security properties. One argument against open source may be that it is easy for attackers to find weaknesses since the source code can be read by anyone, but according to Hoepman and Jacobs[48] this is put into perspective by the fact that at the same time it is more difficult for developers of the software to get away with bad project management or quality control than it is in closed source projects. They continue to argue that, if issues are found in open source software, they can be fixed by anyone with the necessary programming knowledge. In closed source software only the producer has access to the code and is, therefore, the only one who can change it. This problem is further intensified by the issue that many bugs, but also security concerns, are either fixed after weeks or months, or worse yet, not at all [48]. In open source projects, such issues are often visible to the users.

While it is certainly possible to perform large scale review of source code of open source projects there is no guarantee that this is being done by other people, yet it is easy for anyone to assume that someone else does it and thus move the responsibility away from oneself. Hoepman and Jacobs[48] often talk about the possibility to view the code, but they do not discuss how often this actually happens.

Having code available to be read by anyone is similar to how cryptographic algorithms' security usually only relies on some form of secret, often called „key“, rather than the algorithm itself being secret. Former military systems also worked by restricting information to as few people as possible and thus used the „security through obscurity“ principle. Hoepman and Jacobs[48] note that their ciphers were „not particularly difficult to decipher“ [48]. Schneier[92] explains that this is also known as Kerckhoff's principle which states that a good cryptography system shall not rely on the algorithm being kept secret from an attacker.

When using FLOSS, users can, if they so desire, evaluate the security of the software themselves or hire someone they trust to evaluate it for them. Such evaluation is crucial to reduce the risk of backdoors in the software and for example in November 2003 an attempt to insert a backdoor into the Linux kernel was thwarted due to code review. In a closed source project it would be uncertain

if the software developer is really trustworthy enough to not put in backdoors, and even if they are to be trusted, the user cannot view the code and thus cannot verify the claim [48].

Users that do not know enough about software development, or that do not have the time or money to review software in detail, can also benefit from open source software. Due to the nature of open source software, it provides a perfect opportunity to study the relationship between software characteristics and discovered security vulnerabilities. Gkortzis, Mitropoulos, and Spinellis[38] use the National Vulnerability Database (NVD) to correlate vulnerability metrics of open source software with the respective source code of the project. The NVD is closely related to the Common Vulnerabilities and Exposures (CVE) project, which is run by The MITRE Corporation[102], and which can track publicly known software vulnerabilities. Additional information, such as technical details and affected versions of the software, can be tracked by the NVD, which is also run by The MITRE Corporation[103]. While CVE can also track vulnerabilities in closed source software, there are often much more details available about those in open source software. Additionally, both projects only track vulnerabilities that are submitted to them. It is possible, especially in closed source software, that a vulnerability is fixed silently and such a vulnerability may thus not be tracked.

Gkortzis, Mitropoulos, and Spinellis[38] mainly focused on creating a dataset with various metrics. These metrics include the testing ratio, which describes the ratio between program source code and testing code. They also record whether the project uses Continuous Integration (CI) technology to automatically run tests on each new change. To show how this dataset can be used, they compare the vulnerability density to the testing ratio. The vulnerability density is the number of vulnerabilities per 1000 lines of code. They discover that projects which include more tests generally have a lower vulnerability density than those with fewer tests. Projects that use Continuous Integration are more likely to have fewer vulnerabilities. Additionally, projects that have a lower testing ratio appear to contain more severe vulnerabilities. Finally, projects that are written in a non-bounds checking language, such as C or C++, have slightly more severe vulnerabilities than other projects as well.

The quality of FLOSS code is not perfect since it is also written by people, just like code for a closed source project. This is made worse by the fact that many open source projects happily take any help they can get and quality is often not controlled. However, since the code is available to be read, the user can take a look themselves and it is generally accepted that sloppy code is untrustworthy and should be avoided. Of course the user is free to take over the project and make it better or pay someone else to do so. Similarly, if the original author decides to stop working on it, the user can take matters into their own hands and, even if a project is not properly maintained, critical patches can still be applied. All of this is generally not possible for closed source projects without support by the owner [48].

Thompson and Wagner[104] point out that some projects perform small scale code review when merging new changes. During such a review, the reviewer and author both work together and try to find the best solution to fix an issue or solve an architectural problem. Once the participants are satisfied with the solution, the change is merged into the main code base of the project [104]. Perl et al.[81] discovered that new contributors to a project are five times as likely (0.49% compared to 0.10%) to introduce a security vulnerability with a change than seasoned contributors. While this may not be a big surprise, they hope that quantifying the risk leads projects to perform more thorough code review.

Thompson and Wagner[104] performed a study on the effects of per-change code review and discovered small, but significant relationships. Most notably they discovered that there is a negative relationship between the amount of code review and the number of security bugs reported in a project. If the average number of review comments for each change is doubled, they expect the

project to have 5.5% fewer issues. Similarly, projects that merge changes without performing any code review at all tend to have a higher number of bugs and security vulnerabilities. Reducing the number of such unreviewed changes by half, results in an expected 6% fewer security issues. Finally, they note that their results may not be representative for all open source projects since their data came only from projects hosted on GitHub. Some projects, including big ones such as Chromium, Firefox and Apache projects, choose to use other services or host their own ones. These projects are not covered by this study and the authors are also careful to note that the study only shows correlation and not necessarily causation between code review and security.

There are no direct metrics to assess the security of software projects [104], but some of the relationships discussed above can be used as an indirect measure of the security of a project. This can be used to help compare multiple projects and choose between otherwise similar solutions. In general, projects are more likely to have better security properties if the following points apply:

- The project has automated tests for their code. A higher test code to application code ratio is better [38].
- Test are run automatically on each new change [38].
- Code review is performed for new changes. More reviews per change are better [104].

2.5 Secure Shell

Koniaris, Papadimitriou, and Nicopolitidis[57] describe Secure Shell (SSH) as a protocol for logging into remote machines and executing commands on those machines. It uses encrypted connections as well as authentication to create a secure connection and is commonly used in Linux and other Unix-based operating systems.

2.5.1 Protocol Description

The SSH protocol has a client/server architecture and has been defined by Ylonen and Lonvick[114]. A client/server architecture means that a client on machine A connects to a server on machine B. The protocol is split into three major components that are layered on top of each other [114].

- The Transport Layer Protocol: First, both machines negotiate which encryption algorithms they want to use for further transmissions. Afterwards, they establish a session key and the client authenticates the server to prevent Man-in-the-middle (MITM) attacks. Ylonen et al.[115] explain that a MITM attack means that a third party intercepts the connection between the client and the server. When the client then tries to establish an encrypted connection with the server, it actually does so with the attacker. The attacker then simulates a client of their own and connects onwards to the server. If the client now sends data via the encrypted connection, it first arrives at the attacker which can decrypt, read and reencrypt it to send it to the server. If they wish, they can also change the content before they send it to the server. The same holds true for data that is sent back from the server to the client.
- The User Authentication Protocol: Once the client has authenticated the server, it sends its own authentication credentials to the server over the encrypted connection provided by the first layer. If the server is able to validate those credentials, the connection is established and ready for use.

- The Connection Protocol: This protocol supports multiplexing multiple logical channels over a single, established connection. Such logical channels can be used to support, for example, interactive shells, file transfers or network tunnels.

2.5.2 Use Cases

One of the most prominent use cases of SSH is interactive shell usage over the internet. This means that a user connects to a remote machine and executes shell commands similar to how they would in a locally running terminal. This can be used to manage, update, configure, and generally maintain servers, workstations, and even other devices such as networking equipment. It also allows running terminal-based applications remotely [115]. This interactive usage over ssh simply connects the input and output channels of a program running on the remote machine to the respective channels on the local machine. Thus, to the user, the existence of the SSH connection is mostly transparent, apart from differences like network delays.

A related use case is transferring files between machines. For this, SSH is used as the basis for the Secure Copy Protocol (SCP) and the Secure File Transfer Protocol (SFTP) . Both of these protocols use the connection provided by SSH and thus inherit the security properties of this connection [115]. This also means that they do not require their own login credentials, but they can use whatever the underlying SSH connection supports.

The third common use case is point-to-point network tunneling. SSH can be used to tunnel arbitrary network traffic across the secure channel. This can be used to tunnel traffic on a single port over the SSH connection, or even to implement a Virtual Private Network (VPN) tunnel. A VPN creates a virtual network in which both machines can send arbitrary network traffic to each other. They can also be configured to act as a gateway or router, and route traffic to other machines [115]. Essentially the VPN can be thought of as a virtual network cable that connects both machines, but in reality the traffic over the VPN connection is sent, in this case, via the SSH connection.

All of these use cases, can either be performed manually by a user using an SSH client, or in an automated fashion. Automated access means that a machine is configured such that it automatically performs an action without direct user interaction [115]. This is often the case in large IT environments where SSH is used to integrate applications, set up virtual machines in a cloud environment, or for secure transfer of backup data.

2.5.3 Authentication Methods

The SSH protocol provides mutual authentication of both endpoints of the connection. That means that it supports server authentication and client-side user authentication. For a general description of the meaning of authentication, refer to Section 2.1.5.

Server authentication in the context of SSH allows the client to ensure that they are connecting to the correct server and prevent a MITM attack as discussed in Section 2.5.1. This works by using public key cryptography or by using certificates issued by a certificate authority, but public keys are used more often in practice. When connecting to a server for the first time, the SSH client connects to the server and receives the server's public key from the server. Since this is the first connection, the client does not know if the key can be trusted. Consequently, the client asks the user to confirm the key via a secondary communication channel, such as talking to the server operator. Once the user confirms that the key is correct and belongs to the server to which the client is supposed to connect, the key is stored in a „known hosts“ file. Future connections look in this file and discover the existing key. Thus, the client knows that it can trust this key and does not need to ask for repeated confirmation. With the trusted key, the client can now encrypt data to the server using this public key and the server can decrypt the data with its own private key [115].

User authentication refers to authenticating a user towards the server. The protocol supports multiple different authentication methods, however, all of these rely on some form of secret information that is either presented to the server or of which knowledge can be proven to the server. The most obvious method is password authentication, followed by more tricky ones such as host-based, kerberos, and public key authentication. Any data sent during the user authentication is sent over the encrypted, authenticated connection as discussed in Section 2.5.1.

Password Authentication Password authentication can be performed via the basic password authentication mechanism, which is an old, legacy solution, or via the keyboard-interactive mechanism, which is used in most modern environments. Keyboard-interactive supports challenge-response style authentication to provide traditional password authentication or also one-time password authentication. The secret can be compared to various databases such as the local system password store or a centralized Lightweight Directory Access Protocol (LDAP) server. Password authentication is often used when accounts are used interactively by a user. While it is possible to use it for automated access, for example, by hardcoding the password in a script, this is less common [115].

When using passwords, it is very important that they are difficult to guess. In the context of SSH this is especially important because the service is often available to a wide range of machines, sometimes even the entire internet. Access is often not restricted by a firewall, and SSH can provide a high level of access to a machine, since it is often used for system administration. This makes SSH services a valuable target for attackers and roughly half of the automated attacks [57] try to use a dictionary attack to guess the password for the „root“ user. „root“ is the user name of the administrator user of a Linux or UNIX system. Dictionary attack means that the attacker has a list of common passwords, or rules how common passwords are constructed, and tries each word in this list, or dictionary. Ramsbrock, Berthier, and Cukier[84] note that common attempts are the user name, either with or without some additions such as „123“ at the end, and common passwords such as „password“, „123456“, „test“, „passwd“, „123“. Koniaris, Papadimitriou, and Nicopolitidis[57] also discovered other commonly tried passwords such as „changeme“, „111111“, „abc123“, and keyboard patterns such as „q1w2e3“ or „1qaz2wsx3edc“. They also note that attackers not only try attacking the „root“ account, but also temporary accounts such as „test“ and accounts belonging to commonly used services named „oracle“, „nagios“, „postgres“, and „tomcat“. Both papers emphasise that it is important to carefully choose secure passwords.

Host-Based Authentication With host-based authentication, the server has a list of hosts that are allowed to access certain accounts. These hosts can then log into the SSH server without further authentication. Ylonen et al.[115] note that this solution is not recommended, neither for interactive nor automated access, and instead other authentication methods should be used to provide some form of interactive login.

Kerberos Authentication Kerberos is a single-sign-on system which allows storing user accounts, authorisations, and credentials in a centralized directory. Once a user logs in to the central Kerberos Key Distribution Center, they receive a „ticket“ that can be sent to services, such as the SSH server, and allows them to authenticate the user. Ylonen et al.[115] explain that Kerberos authentication is best suited for environments that need to manage many interactive users. It is not recommended for automated access because automated processes should have restricted permissions and not be able to access hosts that they do not need to have access to.

Public Key Authentication Similar to the server authentication, public key authentication uses public key cryptography or certificates. The common case is again the simple form, using public keys [115]. Ylonen and Lonvick[113] explain that these keys are generated on a user machine

and that the private key remains only on this machine or other machines of the user. The public key can be distributed to all servers to which the user needs access and is stored in an „authorised keys“ file. When the user connects to a server and authenticates, the SSH client sends a signed message to the server. The message contains, amongst other things, the session identifier, the user name and the public key that the clients wants to use for authentication. The server verifies that the message content, such as the session identifier and user name, is correct. Next it verifies that the public key is listed in the „authorised keys“ file of the user. If the key is authorised, the server finally uses the public key to verify that the signature on the message is valid. This ensures that the client has the corresponding private key. If the signature is successfully verified, the server accepts the authentication.

Ylonen et al.[115] note that many SSH implementations support configuring restrictions for authorised keys. Such restrictions can include limiting the usage of the key by restricting the commands that can be run, or limiting the client IP addresses from which the key can be used. They also note that an important advantage of this method is that all allowed access is recorded in the „authorised keys“ file on the server. This allows administrators to easily verify who is allowed to access a system and revoke access for certain keys, when necessary, without impacting other keys.

The private key can be stored in various ways, such as on smartcards or in password-protected files. It can also be stored in plain text files for automated access, in which case the key is protected by file system permissions similar to other sensitive data. Multiple keys can be listed in the „authorised keys“ file and each of these can individually be stored using any of the supported storage solutions. Given this flexibility, public key authentication is the most frequently used and also recommended method for automated SSH access. It is also recommended for interactive use, especially in combination with smartcards. In the case of file-based storage, it is important to enforce strong passwords to protect the private key [115].

2.5.4 Access and Key Management

SSH provides secure access to remote machines and it is used in a lot of organisations for administration and automated processes, even in critical systems such as routers, firewalls and security appliances. These tasks obviously require administrative privileges and if the accounts and the credentials are not properly protected, various security vulnerabilities arise. Ylonen et al.[115] identify 7 major vulnerability categories:

1. Vulnerable SSH implementations: Just like any other piece of software, an SSH server or client may contain bugs that present security vulnerabilities or it may be configured incorrectly. To prevent such issues, SSH server functionality should only be enabled on systems when it is required, and disabled when it is not needed. Additionally, server and client software must be kept up to date on all systems. Both of these tasks can be supported by tracking which systems have either software deployed, and by automating the update and configuration tasks. Finally, the software must also be configured correctly. This means that, for example, unused access methods (password, kerberos, host-based), unused features, such as port forwarding and the SSH version 1 protocol, should be disabled. The list of allowed encryption ciphers should be configured to include only the necessary ciphers to reduce the risk of using a vulnerable cipher [115].

2. Improperly configured access controls: While the SSH service itself may be configured correctly, it often uses other high privilege services such as operating system access controls, pluggable authentication modules (PAM), or Kerberos. For example, the operating system may be configured to allow direct root access which may present a problem, or Kerberos may be configured to allow implicit access to other user accounts which then causes im-

plicit SSH access to other accounts. It is also important to base access configuration on the least privilege principle. If public key authentication is used, this can be achieved by limiting which commands may be executed and from which source system the account may be accessed [115].

3. Stolen, leaked, derived, and unterminated credentials: Third parties may obtain account credentials, including private keys, by copying them from a host, fetching them from a backup, having malware harvest them, or deriving the private key from a public key by factoring. These credentials can then be used to gain unauthorised access to a system. The risk of these problems can be reduced by specifying minimum key lengths, selecting approved algorithms, changing keys after a „cryptoperiod“ to ensure that keys have a maximum lifetime, and preventing or at least monitoring key duplication to other systems. Additionally, usage of keys should be monitored, for example, by logging the key fingerprints. This allows detecting keys that are 1) not used, 2) used from unauthorised locations, or 3) used from outside the organisation and that are thus unlikely to be subject to the key rotation controls, meaning that they need to be rotated manually [115]. Barker[8] suggests replacing authentication keys after one to two years.

4. Backdoor keys: Many organisations require that systems are accessed via a bastion host, which is an access management system that records all performed actions. However, users can configure access to the systems behind the bastion host directly by adding their key to the „authorised keys“ file of the respective system. These files are often not monitored and thus the direct connection may not be discovered for years. To prevent such issues from occurring, the „authorised keys“ file may be configured to be only writable by the root user. The SSH server then also has to be configured such that it only accesses the protected file and not the user-writable version [115].

5. Unintended usage: User may use keys for purposes for which they were not intended to be used. For example, a key that was intended to be used only for automated file transfers, may be used to tunnel traffic, thus concealing that traffic from network security controls. This can be prevented by enforcing strict command and source restrictions for automated keys, as well as rotating all keys when an administrator that had access to the private keys, leaves the organisation. In such a case, the administrator's personal key may be removed from all „authorised keys“ files, but they may have kept a copy of the private key of an automated service. Thus, it is important that all such keys that could have possibly been copied, are also rotated [115].

6. Pivoting: Pivoting describes an attack that spreads and jumps between various machines in the network. This can be a big problem in large networks with many automated access keys. An attacker may gain access to one machine and then find a key on that machine that is intended for automated access to another machine. Thus, the attacker can now connect to the next machine. If many such keys exist, the attacker may gain access to an entire network. The key may be found, for example directly in the file system, or also in a backup. Preventing pivoting can be achieved by only configuring either inbound access or outbound keys on each account. If a machine needs to be reachable and also reach other machines, then each direction should be configured on a dedicated account such that if the account with incoming access is compromised, the account handling outgoing access is safe [115].

7. Lack of knowledge and human errors: Finally, a big problem, especially in large networks, is human error during access management. Ylonen et al.[115] note that there are organisations that spend multiple person-years annually on manual key provisioning. In such cases, it is only a matter of time until a key is deployed to an incorrect host or incorrect account. They

suggest that access keys should be managed automatically and all access should be tracked. This can, for example, be done by using a ticket system with predefined templates for the provisioning requests. Once a request has been accepted, the system can automatically deploy the change and thus reduce labor and the potential for mistakes [115].

3 Basics of Logging

Chuvakin and Schmidt[22] explain that logging provides a basic tool to understand what happens in an IT system. Without logs, the system is an opaque box that may or may not perform the desired function. Only with logs, it becomes possible to determine why something happened the way it did and to retrace the steps that the system has taken to arrive in a specific state.

3.1 Motivations and Goals of Logging

The desire to create logs can come from a wide variety of reasons and there can be different goals associated with it. For example, someone might be interested in obtaining data for research purposes, or in showing compliance with government regulations [22].

3.1.1 Motivations for Logging

Oram and Viega[77] explain that logs may reveal weaknesses in a system and allow administrators to see that something is not quite right. They can help to look into the future and tackle potential problems even before they occur. Furthermore, logs show that an organisation is accountable. The authors note that accountability is an important characteristic for an organisation because it sends the message that the organisation acts responsibly and that they take their business seriously. The accountability of the organisation is directly influenced by the accountability of the organisation's IT systems. An important building block to achieve accountability in IT is record keeping in the form of logging.

Doubleday, Maglaras, and Janicke[27] explain that certain regulations, such as the Payment Card Industry Data Security Standard (PCI-DSS) and the Health Insurance Portability and Accountability Act (HIPPA), require data transfers to be protected by encryption. While it is possible to simply encrypt a transfer by using suitable tools, it may also be necessary to show that these tools are and were being used. One way to show such usage is by keeping logs that can be inspected and thus an auditor can gain confidence that an organisation does indeed comply with the regulation. Furthermore, Oram and Viega[77] note that some regulations explicitly state requirements for audit logging. For example, HIPPA contains a section that covers audit, logging and monitoring controls for systems that handle protected patient health information. The Sarbanes-Oxley Act (SOX) also indirectly addresses log collection and reviews of audit logs.

Logging may also be prompted by the desire to gain additional knowledge by looking at what really happens on a system. This also includes academic research, since research often requires data and logs are a great provider for data about events in an IT system. Without logs, it would be very difficult for researchers like Doubleday, Maglaras, and Janicke[27] to analyse attacks on SSH servers.

3.1.2 Goals

Log data may be used to achieve a variety of different goals. In many cases, the same data can even be used to extract different kinds of information to satisfy several different goals at once. For example, logs can be used to gather data about attacks on a system and record which users and passwords the attackers try most often [27]. The same log can be used to show compliance with

a regulation or it can be used to analyse which actions an attacker performs on a machine once they have successfully logged in. However, attackers are quite aware that their actions may be monitored. Some of them regularly check if they are being watched and stop using the machine once they discover that they may have walked into a trap [84].

Logs can also be used to determine what happened after an incident occurred. Oram and Viega[77] provide an example where a file server in a protected company network had crashed. While the company did not know what had happened when they noticed that their server was not working any more, they had kept centralized logs for just such an occasion. They launched an investigation and discovered that while the server was not supposed to be reachable from the internet, at some point the administrators created a publicly writeable directory so that customers could upload files for troubleshooting. The firewall logs showed that an attacker probed the network for weaknesses and at some point discovered this public File Transfer Protocol (FTP) account. While remote logging was not enabled on the FTP server due to an omission, the firewall still recorded the most important events. In this case, it contained evidence that showed that the attacker uploaded a file to the server. This file was later confirmed to be a rootkit that took over control of the server.

Another goal may be analysing system usage and improving system performance. For example, web server logs can be configured to record each requested URL and sometimes also how much data has been sent back to the user. This information can then be analysed to determine which parts of a website are visited more often than others. If the response size is available, it can be used to find unnecessarily large responses. In some cases, for example, it may be an image that has not been scaled down properly or it may be an automatically generated list of items that is not limited in size. The log data can also be analysed to detect old files that were previously accessed, but which have lost interest and are no longer used.

However, performance issues are not the only problems that can benefit from logs. Simple bugs, weird behaviour, and other operational issues that need to be investigated, can also be analysed much easier with log data. Xu et al.[110] note that various logging techniques were used since the dawn of programming. Log content varies depending on the program, but often things like variable values, exception traces, or runtime statistics are logged. Some programs even log full sentences that are designed to be read by people, usually the developers of the software themselves. Beschastnikh et al.[14] note that even if no clear error message is logged, log data can help to understand what happens inside a software. In this case, the software is treated as a black box and a user tries to gain understanding of the internal processes by reading the various logs created by the system.

Troubleshooting and performance optimization can also be combined. Lim, Singh, and Yajnik[58] describe how they utilize log data in a Voice-over-IP (VoIP) call center. They work towards multiple goals, such as improving the availability and reliability of the call center, debugging problems in the call center network, and predicting anomalous behaviour. They explain that system trace and debug logs allow system administrators to identify the causes of failures. In the context of a call center, identifying failure causes may be very important, especially if the call center handles, for example, emergency services. System developers often include trace and debug logs, which contain a lot of data for in-field debugging. The authors state that they believe these logs depict the state of a system fairly accurately. Therefore, they can be used to detect patterns of behaviour that can predict failures so that they can be addressed before they cause loss of business, loss of revenue, or even loss of life.

Logs might even be used to troubleshoot problems that are not local and instead are happening outside the control of a company. Kiciman et al.[55] explain that a content service provider only has limited visibility into the state of the internet at large. This presents a problem because their business highly depends on their ability to receive and respond to requests from clients from all

over the internet. The authors note that users are quick to label a content provider as „unreliable“ when they notice problems, however, problems often only affect a subset of users. Users that experience problems may share the same network provider, live in the same country or their providers may use the same upstream provider to reach the content provider. Content providers can try to contact unreliable network providers and ask them to investigate the problems, but some problems only occur infrequently and network providers may not be motivated to try to find the problem. Instead, they may ask if the problem still exists and in many cases it might not at the time of the communication, but it may resurface again later only to vanish again before the network provider can investigate it. If the content provider has more information about the problem, they can relay that to the network provider. The authors hope that the additional information helps the network provider during their investigation and thus they resolve the problem instead of waiting for it to happen again. However, the content provider generally does not have direct access to the network provider's system which is a problem when they want to gather more information. A solution to this issue is to infer information about failures from their available information sources instead of trying to find new ones. One such source are log files generated by various services in the content provider's own network. By analysing this data, Kiciman et al.[55] can detect reliability and reachability problems that end users are facing when they try to access resources of the content provider. They can also infer which of the many network providers between them and the end user is likely to be the cause of the issue. This allows them to contact the correct provider and thus increases the chances that the problem will be fixed quickly and in the correct network.

3.2 Requirements for Logging

Similarly to the various goals an organisation may want to achieve by implementing logging, there can be various different requirements towards the logging system itself. When more than one application is used in an organisation, a common solution is to use software that implements the syslog protocol. Gerhards[36] explains that the syslog protocol provides a relatively simple framework to generate, aggregate and store log entries. In this document, the requirements towards a logging system are grouped by which parts of the system they affect. This follows the idea of threat modelling, discussed in Section 2.2.3, by following the path of a single log entry from beginning to end to determine all requirements.

3.2.1 Data Generation and Collection

The first step in a logging system is the generation and collection of the log data. While it may sometimes be possible to reconstruct pieces of data by cross-referencing entries from various other logs, data that is not collected is gone. Therefore, it is important that log levels are set to an appropriate value in applications to make it possible to link connected entries later on. Kent and Souppaya[53] note that log content is often inconsistent. For example, one log source may only log the source IP address of a request, while a second source, which may be a different application, logs only the user name and not the IP address. Without a third source, it may prove difficult to link these two entries together since they do not have any common values. This happens because logs are often optimized for efficiency and logging the same value many times obviously increases the resource requirements related to generation, transmission, and storage of each log entry. However, many applications provide configuration options that allow an administrator to set a log level that tells the application how much information it should log. To address this problem of badly connectible information it is important that this level is set to include sufficient detail while also taking transmission and storage requirements, which are discussed in Sections 3.2.2 and 3.2.3, into account. At the same time, it is also important to note that some regulations, such as the General Data Protection Regulation (GDPR) created by the European Union[31], may

impose restrictions on which information may be recorded, how it may be used, and how it must be protected. Thus, the chosen level of detail must also consider if such regulations can be met by the log data generation itself, and by the transmission and storage solutions that follow it. Log sources where badly connectible log entries can happen quite easily are email systems [77]. Email systems are often composed of multiple applications that provide parts of the services that the full system provides. For example, the Simple Mail Transfer Protocol (SMTP) service, which allows sending mails to or receive mails from other systems, may be one application. Another application could be the Internet Message Access Protocol (IMAP) service, which allows users to access mails that are stored in their mailbox. Furthermore, the spam and virus filters are often also dedicated applications. All of these applications may be provided by different vendors, run on different machines, and record received and sent messages in different log entries or even different files. They may even support different log data storage or processing solutions and it may be necessary to configure the log level for each of them separately.

Having too much detail in a single message may also be counterproductive. Depending on where log data is sent to later, there may be limits on the length of each message. For example, the syslog protocol specification by Gerhards[36] only requires implementations to support a minimum message size of 480 characters. It notes that they should support lengths of 2048 characters and may support longer messages, however that is optional. Messages that are too long should be truncated, but implementation may also choose to simply discard them. This obviously leads to a problem, when an attacker controls a part of the log messages. If the attacker is able to inject a sufficiently long string, the outcome depends on the implementations that later handle the message. If the message is dropped, it is obviously lost, but even if it is truncated, it may be missing important information. Messages should be created in such a way that the vital parts of the message are at the beginning, while less important parts are at the end. They should also try to be sufficiently short, which can for example be achieved by splitting a message into multiple consecutive, smaller messages [36].

Collecting log data may not always be as simple as one would hope. Many systems and applications support writing their log data to a file or sending it to collection services, or databases, but the Simple Network Management Protocol (SNMP) does not. Rather, SNMP logs are supposed to be submitted to a different system for processing. SNMP can, for example, be used to deliver alerts from an Intrusion Detection System (IDS). Such alerts may include details about a detected information leak in the network, which is obviously highly important data that should not simply get lost. SNMP alerts are delivered to SNMP traps which then process the data [53]. Since the data may not be stored locally on the system, it is important to configure the SNMP trap destination and thus ensure that it is collected like other logs.

3.2.2 Data Aggregation

Aggregating log data means that log data from various machines, log files, and services is combined in a central log. This allows for easier usage and better correlation between log entries. In theory, correlating log entries should be quite easy since nearly all logs include time stamps in each individual entry. These time stamps then provide a chronological sequence of events showing when an entry was created and which entries came before and after it. However, correlating log entries can still be quite difficult, especially if there is a difference between time stamps of log entries [77]. This can happen for multiple reasons, including clocks that are not synchronized correctly and different time zones. Clocks can be synchronized automatically by using Network Time Protocol (NTP) software, while time zone handling may depend on the specific application. Some applications may use the local time zone and either include the offset in the log entry or omit it, while others may default to UTC and again include or omit the offset. Kent and Souppaya[53] also point out that log sources may represent the same value differently. For example, applications

may use different formats to record time and dates, such as DDMMYYYY, YYYYDDMM, or YYYYMMDD, with YYYY denoting a year, MM a month, and DD a day. Obviously, depending on the exact values, this can result in ambiguities during later analysis. Log aggregation reduces these problems because the aggregation point can augment the data with its own time stamp. Additionally, a central aggregator can record logs in a strictly ordered fashion and thus further improve correlation possibilities.

Furthermore, central aggregation does not only help against unintentional correlation problems, but also against attackers. If the goal of the logging system is to allow for forensic analysis of the data, having a central aggregator ensures that the log data can be trusted. Assume that an attacker obtains root privileges on a machine. Using these privileges, they are able to change the configuration of the logging system on the machine and also access the files that the system has been logging to. If they want to hide their tracks, they can just remove the log files. However, missing logs files may trigger errors or alerts and administrators may wonder why parts of the log are missing when they want to resolve an unrelated issue. More careful attackers may consequently not remove the entire log, but instead remove or change parts of the log that show their activities. They may even fabricate new entries that look as if something else had happened. Thus, the logs on a compromised host are not a source of reliable information. If log data is continually forwarded to a central logging aggregator, the logs of this aggregator may still be considered trustworthy, even if the same cannot be said for the source of the data [53, 77].

However, attackers may not only be able to inject log entries into a local log, but other logs as well. If logs are transferred via a network connection, it is important that this connection is authenticated. Gerhards[36] notes that the early syslog implementations performed plain text transfers via the User Datagram Protocol (UDP). The standard recommends against using UDP because it is inherently unreliable, however, it is supported for compatibility reasons. The standard further explains that with a simple UDP transfer, an attacker can easily send their own data to the central log server since often no authentication is performed. This also allows an attacker to flood the log aggregator with useless messages, which may prevent other machines' log data from arriving due to network congestion. Additionally, UDP provides no assurances about the order of data delivery and not even the assurance that data is delivered successfully to the destination. A data packet that is sent, may be dropped by the network if the network is overloaded, or it may be dropped by a malicious attacker that intercepts the traffic. Postel[82] explains that the UDP protocol is deliberately simple and protection against such problems is missing on purpose. If reliability is necessary, other protocols, such as the Transmission Control Protocol (TCP), should be used [82]. Gerhards[36] states that the syslog protocol does not include any safeguards to work around these limitations of UDP because the standard only specifies the format of the messages. It does not specify a transport protocol and thus any transport protocol may be used, as long as the message content is not altered between a sender and a receiver. However he warns that if an unreliable transport, such as UDP, is used, some messages may be lost. Therefore, it is recommended to use more reliable protocols, such as TLS to avoid these problems [36].

There is also no replay protection in the syslog protocol. This means that an attacker can capture network traffic and resend it at a later time to simulate regular log activity of the machine. Newer implementations support TCP and also TLS to address these problems. Implementations that conform to the standard are even required to support TLS and its usage is recommended by the standard. TCP and TLS allow the log aggregator to authenticate the source of messages and thus limit handling of data to authorized senders. TLS also ensures that network traffic is protected against various threats such as replay, MITM, or sniffing attacks. Without TLS an attacker might be able to redirect or sniff syslog network traffic and thus be able to read log traffic even if they cannot directly read the log files themselves.

Another reason for using log aggregators may be availability. Forwarding log data to an additional system obviously increases the number of systems that have a copy of the data and thus raises the availability of that data. More redundancy can be added by using multiple aggregators that work together, or by sending log data to multiple independent aggregators. If any of the independent aggregators becomes unavailable, the others can still be used to retrieve the log data [53].

3.2.3 Log Data Formats and Storage of Log Data

When log data arrives in a log aggregator, it is still only stored in memory and not yet written to a file. Moving log data to more permanent storage presents some challenges depending on how data is stored. One possibility to store log data is by using text files that simply record each log entry on one line. Simple logs that record only data from one application may include a time stamp showing when the entry was logged and the message itself, however some also only record the message without additional information.

With multiple applications, syslog software is often used to aggregate and store the log entries. The syslog protocol supports combining entries from multiple applications and hosts and each entry may include a time stamp, the host name of the machine and the name of the application that created the log entry, and finally the message content. Applications are given a lot of choice regarding the log message content since the syslog protocol treats it as a free-form message. The syslog standard also includes structured content support which allows for a key-value transmission of content, but the protocol has only been approved recently and previous implementations and suggested standards did not include this support [36]. The content of the free-form message can range from human readable sentences, through messages that include a human readable part, but also list fields and their values, to fully structured messages where each message of the application uses the same structure. While this freedom offers the applications high flexibility in generating their log content, it also presents challenges for log analysis. Even if one application uses structured log messages, other applications may not follow this approach [53]. Furthermore, the message content is defined as a free-form Unicode string, which means that it supports a wide range of characters, including control characters such as the line feed character (\n). Similar to simpler text file based logs, this raises the issue of log entry injection which is discussed in Section 3.4.3. The standard advises that syslog applications should avoid using such control characters and that they may be modified by any application, however, using them is still allowed and special handling is optional [36].

Some systems, especially Security Information and Event Management (SIEM) software, which integrates log aggregation, analysis, and storage into a single software package, may also use proprietary formats. If additional software is used, this can present a problem because this software may be unable to read the proprietary format. Kent and Souppaya[53] also note that SIEM software is often more resource-intensive than syslog based software and more complicated and expensive to maintain. On the other hand, such software may store more data fields than a syslog based log, and it typically includes more log management capabilities.

Log entries may also be stored in databases, which can be beneficial for later log analysis [53]. Using a database also addresses the problem of log entry injection because each log entry will be stored in one row in the database. This means that the database handles record separation which means that a line feed character is no longer the separator and thus cannot cause problems.

Chuvakin and Schmidt[22] explain that in some instances it may also be necessary to keep log data available for a relatively long retention period such as months or years. For example, PCI-DSS requires an audit trail of at least one year [22]. In this case, the log data typically needs to be archived, which includes considering the impact of the used log format, choosing archive storage media, implementing integrity verification, and ensuring secure media storage. When archiving

logs, the choice of the log format is even more important than within a short-term log system. Archived logs may be kept for multiple years and are often not changed during that time to ensure that they cannot be tampered with. However, after some years, software may be unable to read old, proprietary formats. This means that administrators must choose if they wish to archive logs in the proprietary format, a standard format, or both. Once a format is chosen, the data can be stored on a variety of storage media, such as backup tapes, CDs, DVDs, and online storage services such as a Storage Area Network (SAN) or dedicated log archival servers. Care should be taken that the storage media which are used, are able to last the retention period. If this is not the case, the archived data needs to be transfered to new media before the old media degrade. When data is transferred to the archival media, it is important to ensure that the data has been transferred correctly [53]. Checksums, which are discussed in Section 2.1.2 can be used to verify that the content of the source and destination media are the same. The checksum can also be stored on the media to allow for repeated verification. If authenticity is also desired, the checksum can be replaced by or augmented with a MAC as discussed in Section 2.1.5.

Finally, the archive media should be stored in a physically secure location. This may include preventing unauthorized physical access, and monitoring temperature and humidity. The media may also be stored in an off-site facility to ensure availability in case of a disaster. The German Federal Office for Information Security (BSI) [37] recommends that redundancies should not be located in the same fire compartment within a building and, more generally, they should be sufficiently far apart such that a local disaster does not affect both of them. Following these guidelines they recommend a minimum distance of 5 km between redundant data centres.

3.2.4 Access to Log Data

While it is nice to have and keep logs, they are not very useful if they are never accessed. Access to log files is often only protected by file system permissions. Regular users of a system should not be allowed to read log files. Ideally they should not even be allowed to access log files in any way, expect when some level of access is strictly necessary for creating log entries. In any case, only administrators should be able to rename, delete, and perform other file-level operations, such as changing access permissions, on log files [53]. The syslog protocol allows reducing log file access in this manner because the data is sent to the syslog daemon which then writes the entry to the log file. The user only needs to be able to talk to the daemon and file access is not required. Additionally, the syslog protocol allows storing log data on different machines and transfer data via a network connection [36, 53].

When an incident occurs, it is often necessary to review and analyse log data that has been created around the time of the incident. Additionally, it is sometimes necessary to review all other log data for indications of similar incidents that have been missed or to see what else an attacker might have done before or after they gained access to a system [77]. This requires access to any archived logs, as well as current log files. First it may be important to review all data, but later it might be necessary to review only data from specific time frames. Thus, data should be available for chronological, but also random access patterns.

Various applications that also need access to log data include log visualisation and log analysis tools, which are discussed in Section 4.3. Depending on their exact usage scenario they either require access to all log data or only to newly arriving log data. Such access should be restricted to be read-only and to only cover the data that the application should analyse. If an application should only be used to analyse, for example, login and logout activities of a particular service, then the log data should be filtered before being sent to the analysis application. Original syslog applications only supported filtering based on the severity of a message and the message's facility. This was intended to link a message to a specific service, but the field only had a limited number of possible

values, thus limiting its usefulness. Some more modern syslog implementations support more complex filters, such as handling messages differently based on the source host or program. They also support usage of regular expressions to match specific content in a message and sometimes even multiple filters can be applied to the same message. Matched messages can then be sent to different destinations, such as different files or other machines [53].

3.3 Types of Logging

There are various types of logs that may be created depending on where and which data is logged. Different types of logs also result in different properties of the resulting log files. Consequently, some logs can be used towards certain goals, while others cannot [22]. For example, a web server's error log can be used to track problems in the web server, but it generally does not contain information about all pages the users of the server have accessed. Thus, the log cannot be used to check which files of a website were never accessed by users.

Different types of logs may be combined into a single log file. For example, errors may be logged to the common application log with a simple prefix that distinguishes them from other log data. The same log file may even contain debug logs which are also often prefixed to help differentiate them from other entries. When a log aggregator is used, all log data may be written to a single file so that it is easy to see which order certain events took place in. Obviously, such a log combines all log types used by the log sources [22, 53].

3.3.1 Application Logging

One of the more varied logging types is application logging. For the purpose of this thesis, application logging describes log data that is specific to an application and shows what is happening inside the application itself. In some cases, it may provide a high level view of the inner workings of an application, while in others it may include various details about specific behaviour. It does not focus on any specific events, such as errors, but instead includes a relatively broad spectrum of log entries from various places inside the application. The goal of an application log is to help the reader gain an understanding of what the application does. It may explain why the application performed certain actions, which requests it received from other entities, or it may report suspicious events [22]. Depending on what is considered to be an application, some of the other types of logs may be considered to be specialized versions of an application log.

Chuvakin and Peterson[21] explain that application logs are important because the application developers can gather necessary context when creating the log entry. Other types of logs, such as system call or network logs (refer to Sections 3.3.3 and 3.3.4), have very limited visibility and can only log the information they have at the time. The application on the other hand, can be built in such a way that it records important state information which can be used when logging events. When logs are analysed for forensic evidence, it is often important to answer questions like who was involved in the event, what happened, where, when and why. Knowing who was involved may, for example, be supported by the application logging the user name and the authenticator that confirmed the user's identity. The application can also augment the log with information regarding the component where the event was created, the result of a requested action and the reasons why the application arrived at a particular result. All of this information is difficult to come by when the application is treated as a black box. It may not even be available in all cases, since an application might only remember the access level of a user once they authenticated successfully. For the functionality of the application it may not be necessary to remember the user name, but for the log it makes a big difference in terms of usefulness if such information is available.

Another important question regarding logging is the issue of what to log and when to log something. Marty[62] notes that applications often log too little, or they log information in such a way that it is difficult to understand. Which events should be logged, depends on the specific application and use-cases. Possible log entries include business, operational, security, and compliance events. Business relevant logging may include information as to when a specific feature was used or how long the execution took. This allows the organisation to remove or disable unused features and to improve the performance of badly performing ones. Operational events include errors that occur, starting and stopping of subsystems, changes to objects, such as configuration changes, backup creation, and code updates. These allow the organisation to keep an overview of their systems and ensure they are running smoothly. Security relevant events include login and logout of users, password and authorization information changes, denied authorizations, and actions performed by privileged users. Compliance related events depend on the exact standard or regulation that the application should be compatible with, but often include logging privileged access. Most of this information is difficult to obtain from other sources, thus it is important that the application developers build the application in such a way that meaningful application logging is possible [62].

3.3.2 Command Logging

A command log is a specialized kind of application log that only concerns itself with commands that are received by the application. For the purpose of this thesis, a command is defined to be a high-level request that is sent to an application by another agent. Commands may be sent by various agents, such as users or other programs. In response to the command, the application then performs an action and, depending on the command, it may return a result to the agent that issued the command. A command is considered to be high-level, compared to system call or network logs (refer to Sections 3.3.3 and 3.3.4), because one command to an application may result in many operations being performed. For example, an application may issue multiple system calls or network operations to perform the requested action. Examples for commands are shell commands and Structured Query Language (SQL) commands. Garfinkel, Schwartz, and Spafford[34] explain that shells are programs that are used by users to issue commands to a computer. Shell commands often instruct the shell to run another program and display the output, which can be called the result of the command. Programs that may be run by a shell include „rm“ and „cp“ which allow deleting or copying files. Duncan and Whittington[28] explain that SQL commands are supported by various databases and these commands allow an agent to access or modify the database's content. Certain commands, especially those that allow the agent to retrieve data from the database, may return a result to the agent.

A command log contains the command itself and may include additional details such as the user that issued the command, how long it took for the command to be executed, or the result of the command. The command logs discussed in the remainder of this section are brief and do not include details about what actions and decisions the application took internally to arrive at the result. This is in contrast to more general application logs, system call and network logs, which are discussed in Sections 3.3.1, 3.3.3 and 3.3.4.

Duncan and Whittington[28] store audit data of an application's MySQL database for forensic analysis in form of a command log. This audit data includes connection attempts, queries that were issued and their results as well as changes to table definitions. Changing table definitions in MySQL is done via SQL commands, which means that this audit log essentially contains all interactions of the database server with users or applications. The idea behind creating such a detailed log is that it allows them to chase down the root of any attack on the database. Ideally, it also enables them to reconstruct the entire database from scratch in the event of it being deleted. They note that this generates a vast amount of log data, since it essentially duplicates the entire database content and even augments it with additional information. It is arguably more verbose

than the database content itself since it logs the full commands in SQL syntax. Deletion or update commands are also logged, which is in stark contrast to the database content itself where a deletion results in reducing the size of the database content. Finally, the log includes commands that failed to execute due to syntax or permission errors as well. They explain that keeping such a detailed log is important because it allows them to partially rebuild a corrupted system by selecting which of the commands should be used and which should be excluded. For example, if an attacker issues a deletion command and the effects are not discovered in time, the last backup that still includes the deleted data may be already gone. With an extensive command log, it is possible to see which actions an attacker performed in the database and even replay all commands on a new database and exclude the malicious commands to neutralize their effect [28].

A simpler form of a command log is a shell history file. Mateljan, Peter, and Juričić[63] explain that these files are commonly created by interactive command shells and that they record each command that is executed by the user. The location of shell history files differs between operating systems, but Mac OS X and Unix systems default to storing the command history in a file, while Microsoft Windows only saves the commands in memory. The user can, for example, use this history file to look for commands that they have used before to avoid having to retype long commands. They can also recall a command and edit it, which can be useful if the saved command has a syntax error. Furthermore, they can fetch all commands to extract them to a different context, such as rework them into a script or put them in a document [63].

While these possibilities are helpful for users of a system, Balduzzi et al.[5] have also used command logs to gain insight into which actions attackers perform on a system once they have achieved shell access. Command logs provide a concise log of the initial actions of attackers, which allows analysing attacker behaviour, especially when well-known commands are executed. After successfully attacking a machine, attackers use various commands to check which hardware the machine uses and which operating system and kernel are running. They also check who else is connected to the machine, and which processes are running before they change the password and download additional software which they then install on the machine [57, 84]. Some commands may even include authentication credentials which are then recorded in the command log. Software such as database clients, VNC servers, and Domain Name System (DNS) management clients may receive the password they need to perform their action via the command line. This presents a problem, when the command log is leaked to third parties since these can then extract the relevant passwords [5].

3.3.3 System Call Logging

Keniston et al.[52] explain that system calls are at the boundary between an application, the hardware it runs on and other applications. As the name suggests, a system call allows the application to call a function in the operating system. The operating system generally provides abstractions for hardware, such as network interfaces and storage devices, and other processes that are running on the system. Logging calls to these functions can be used to gain insight into how an application interacts with its environment. Such a log may also contain the arguments that are passed to the system call and the value that is returned from the operating system to the application. A system call log can be seen as a special application log of the operating system or the operating system related parts of the application.

A common tool, used by Dagenais et al.[23], to log system calls on a Linux system is „strace“. Strace uses the „ptrace“ system call to tell the operating system that it wishes to attach itself to the another process, called the „tracee“, and trace the execution of this process. Ptrace can also be used to implement debuggers which, for example, allow going through a program's execution step by step or inspecting the memory of a program during execution. After issuing the ptrace call, the

kernel transfers control to strace each time the tracee issues a system call. The tracee is blocked while strace has control. Strace can then extract the values it needs, such as the name of the system call that is about to run and the parameters passed to it. Control then returns to the tracee since strace only wants to monitor the execution and not interfere with the regular process operation. After the system call is executed, control goes back to strace which now fetches the return value of the call and again returns control to the tracee. The collected information is then logged to a file or shown to the user of the strace tool [23]. During execution, processes sometimes execute other programs to perform specific tasks. A great example for this are shell scripts, which often mostly consist of calls to other programs. These programs generally run in their own processes and thus Abed, Clancy, and Levy[1] note that monitoring only the top-level process, which is the shell itself, is not generally very useful. Helpfully, strace provides an option to follow child processes automatically and also collect trace information for their system calls [1]. The resulting log file then contains the process ID, the system call name, the arguments to the specific call, and the return value [23].

However, strace is not the only system call logging solution. Strace runs in user-space and collects the information via the ptrace API, which was intended to be used for debugging purposes. This API does not only allow the application to capture system call, but as mentioned it also allows it to perform various other actions, such as reading and writing access to tracee's memory, registers and user area. Having this much power obviously allows for many problems to occur. Keniston et al.[52] note that ptrace based tracing can have substantial overhead because control often switches between the kernel and the tracer application, the process calling ptrace must be the parent of the tracee, there can only be one active tracer process per tracee, handling more than one tracee is non-trivial, and especially signal handling and delivery are tricky to get right. If the goal is to only log system calls, a simpler approach is to log the events in the kernel directly, rather than providing a complex user-space interface. The linux kernel contains various implementations of such tracers, including Kprobes, Utrace, Uprobes, SystemTap, and the audit framework, which is discussed in Section 4.1 [52]. It also contains other frameworks that can be used to implement such functionality, such as LSM, which is discussed in Section 4.2.

3.3.4 Network Logging

While the routing functionality of a router may be seen as an application on that router, network logging in this thesis is defined to contain logs that describe network packets and their handling. Such logs may include the addresses of the sender and recipient of the network packet, the size of the data, or even the entire data itself. However, logging the data may not always be helpful, since it may be encrypted and thus unreadable without the corresponding key.

The most simple form to create a network log, is to collect all network packets and dump them into a file. Sanders[90] shows that this can be achieved with tools such as „tcpdump“, which despite its name cannot only handle Transmission Control Protocol (TCP) traffic, but also various other network protocols. „Wireshark“, which is a software package that includes various tools to dump and analyse network traffic, can also be used. For dumping from network interfaces, both tools require that the network traffic that they should record, somehow arrives at the system they are running on. This presents a challenge because to reduce wasted network bandwidth, most networks use network switches, which only send data to the systems that the traffic should go to. Other systems in the network generally do not receive network packets that are not addressed to them. Ways to work around this limitation include, dumping the traffic at a choke point in the network, configuring switches to mirror traffic to another machine, or using Address Resolution Protocol (ARP) poisoning to trick other machines into sending traffic to the dumping host instead of the actual target [90].

The first way of dumping at a choke point obviously requires that a choke point is known. Even for a small network with a few machines and a single router that connects the network to the internet, this may be rather difficult. For example, if the goal is to log all network traffic, dumping traffic on the router alone will be insufficient since internal traffic will be inside the network and not cross the router. It may be necessary to either redefine the goal, or create logs in multiple places which means that traffic may be logged multiple times unless it is filtered properly. However, even if the traffic is filtered correctly, dumping all network traffic can result in an enormous amount of data. When a network has 23 machines that are all connected with full-duplex 100Mbit/s, they may push out a total of 23*100*2 = 4600Mbit/s of traffic. While this may not be a very likely scenario in an office environment, even a much smaller amount of traffic may already cause the logging setup to overload and fail. This example also hints at the amount of data that needs to be stored if the network dumps are to be kept for some time. Storage and network bandwidth can clearly quickly become an issue when using network dumps as logs, which calls for more efficient methods [90].

Network firewalls can be used to create more compact forms of logs by logging connections on a higher level, rather than logging all data that was passed back and forth. According to Chuvakin and Schmidt[22], such a log may contain the source and destination address and port of a connection, the duration of the connection from being established to being closed, and the decision of the firewall whether to accept or reject the connection, as well as the total amount of data transferred in either direction. While this type of log contains less information than a full dump, it can still be used to gain an overview of the network activity and correlate with other log sources. However, even on its own, such a log can, for example, show an attacker probing various ports on a network, finding a service that replies, attacking this service and uploading some data after which the server hosting the service makes suspicious connections to other hosts. From such a network log it can be deduced that the service was vulnerable to an unknown security issue, which the attacker managed to exploit and which then allowed them to take over control and try to attack other hosts [22]. Systems such as Snort, created by Roesch[86], can be used to automatically analyse network traffic and directly generate a log file that contains only interesting events, rather than information about all connections.

3.3.5 Error, Debug, and Request Logging

While application, command, system call, and network logging differ in which kind of data they log, there are other types of logs which differ by their later purpose. These include error, debug, and request logs. Error logs are logs that contain only errors, which means they are very valuable when trying to diagnose problems in a system. They are relatively concise and avoid a lot of noise by focusing on collecting only data that a software developer determined to be an actual error. Just because an error occurred that does not mean that this also automatically leads to a failure in some place, but when problems are detected, error logs are a good first place to look for clues on what is causing the problem. However, they are often just a start and further investigation is required to fully understand the problem and find the root cause. An example of such an error message is one saying that a file cannot be created because the target file system is full. The application cannot know why the file system is full, but when an administrator sees such an error in an error log, they know what to look for, and they can investigate which files take a lot of space and determine if the file system size should be increased or if there is left over data that should be removed [22].

A debug log on the other hand often contains a lot of messages with the aim of helping debug an application that behaves incorrectly in the field. When software developers want to diagnose a problem in an application, and they do not yet know what exactly is wrong and why this particular issue can even happen, they often turn to debug logs to look for hints or anomalous behaviour. However, debug logs are not only useful to developers, but they can also provide insight for system administrators. Even if the operator does not have access to the source code, debug logs can

explain the inner workings of the application and help the administrators understand why the application behaves in certain ways. At the time of creating the application, it is not known which issues will have to be solved by using the debug log. This means that it generally contains as much information about the things that are happening in an application as possible. It contains failures, errors, faults, but also information about expected, regular behaviour such as variable values, calls to certain functions, and many other details about the internals of the application. Recording this much information also means that the resulting log file grows in size very quickly. Because of this, debug logging is generally disabled during operation and only enabled when necessary [21, 22]. Oliner, Ganapathi, and Xu[72] note that creating this much log data and storing it somewhere may also impact performance of the system in general since the logging system must process the additional data. If resource usage data is gathered for accounting reasons, excessive logging may also introduce unacceptable overhead in these measurements [72].

Request logs record some information about each request that an application receives. They are enabled all the time and provide a middle ground between an error log and a debug log. They can be used for various purposes such as monitoring the number of requests the application receives, billing purposes, or analysing which resources are requested. An example for this type of log is a web server access log, which often records the IP address of the user, the URL they have accessed, as well as the return code [22].

Audit logs are similar, in the sense that they are always enabled, but they are relevant to secure operation of a service and they allow demonstrating accountability. Audit logging is covered in detail in Chapter 4.

3.4 Log Management and Security

Biskup and Flegel[16] explain that log data may sometimes contain sensitive information, such as passwords or health information. While it is generally advisable to reduce logging so that only necessary fields are logged, logging sensitive information may sometimes be required for various reasons. In these cases it is especially important to properly protect log files against unauthorized access by third parties. This can be achieved, for example, by using encryption or pseudonymization. Another security consideration is availability and integrity of log data, which requires preventing log data injection and ensuring storage security as described in Section 3.4.3 and 3.4.4.

3.4.1 Log Encryption

To protect the confidentiality of log data, it may be encrypted. Confidentiality is defined in Section 2.1.1. While encrypting data may sound like a simple solution, there are various trade-offs to be considered when implementing log encryption in an organisation. There are also various stages at which data may be vulnerable to security threats, such as during transfer and storage. Examples of threats against transfer of log data between systems are discussed in Section 3.2.2. When log data is stored on storage media, encryption can be used to ensure that if data is leaked or even if an attacker gains access to the data on the machine, they are unable to read it due to a missing decryption key.

If only the data is leaked, many encryption schemes provide adequate protection for confidentiality. For example, symmetric encryption uses a secret key with which data is encrypted, but which can also be used to decrypt it again. Without knowledge of the secret key, it is infeasible for an attacker to reverse the encryption or to guess the used key. Another solution is to use asymmetric encryption, which uses a public and private key. Public key cryptography is also used by SSH for authentication and this usage is explained in Section 2.5.3. Public key encryption also allows protecting data against an attacker with access to some machine in the logging system. The public key

can be used to encrypt data, which can then only be decrypted by the private key [92]. This means that if public key cryptography is used and the data is encrypted on the log generator, the data's confidentiality is ensured even if the data is sent to other machines. Only when someone wishes to read the data again, they need the corresponding private key. In both cases, it is important that the key that is used for decryption is required to read the data, which means that this key needs to be kept available, but it also needs to be kept secure, since it is the reason why an attacker cannot read the data. If the private key is stored on the log storage machine and an attacker gains access to that machine, they can use it to decrypt the log.

While it may be infeasible to break the encryption at the time of creation, it is also important that encryption is not the only line of defence against misuse because then everything rests on the decryption key being kept secret and on the encryption algorithm being secure forever. An attacker could simply keep an encrypted copy of the data and decrypt it years or decades later [34]. Depending on the exact data this may or may not be an issue, but it is always advisable to also properly protect the encrypted data.

However, the encrypted data and key must not only be protected against outside attackers. The encryption and decryption keys themselves may be known to system administrators, which can present a problem when they leave the organisation. This is similar to the argument Ylonen et al.[115] provide for SSH public key authentication where and administrator may keep a private key of an automated service to access the organisation network, even if their own key's access is revoked. In this case, the administrator may keep the log data encryption or decryption key. Doing so, they could potentially use the keys to add, change, or view log data at a later date if they are able to gain sufficient access to the encrypted data [8]. It is also possible that they become the target of an attacker who wishes to steal these keys. Thus, it is important that, just like with SSH keys, log encryption and decryption key are changed regularly and when certain special events occur.

Compared to rotating SSH access keys, rotating data encryption and decryption keys is a more complex problem. While an SSH key can be removed from the „authorised keys“ file of the server, which ensures that the server no longer accepts the key for authentication, doing something similar is not possible for encrypted data. Existing logs must be re-encrypted with the new key to ensure their continued protection. This is especially crucial if log data is archived for a long time, such as multiple years.

Additionally, Barker[8] recommends that system that use keys should strive to limit the potential for catastrophic failure. For example, if data from multiple sources or users is encrypted, it should use multiple keys instead of a single, global key to encrypt all data. If multiple keys are used and a single one is compromised, this ensures that the remaining data stays secure [8]. With only a single, global key, a compromise would compromise the security of all data at once.

3.4.2 Pseudonymization

Biskup and Flegel[16] explain that pseudonymization is the process of changing data in such a way that it no longer points to a real person, but can still be used for the original goal. For example, a goal may be performing activity auditing and this may require that suspicious activities can be traced back to a real person. However, protecting users is important and personal information must be protected according to various regulations, such as the GDPR created by the European Union (EU) [31]. This can be achieved by means, such as, for example, replacing names, birth dates, or other personal information with pseudonyms. These pseudonyms may either be random values or they may be some mangled form of the original value, but it should be practically impossible to reverse the pseudonymization without additional information. Such additional information could, for example, be a list that connects the original value to the pseudonym and thus allows reidenti-

fication when necessary. Pseudonymization is often used to remove personal information from a log file, and replace it with pseudonyms, before processing or storing it. This way, the personal information of users can be stored elsewhere, which makes it more difficult for malicious parties to obtain it and correlate it with log file content. It also allows using the log data for various collection or processing activities which are prohibited or may require explicit consent if the data contains personal information. Once this information is removed, even if it is replaced with a pseudonym, such restrictions may no longer apply, easing usage of the data [16].

A naive initial approach is to use encryption to encrypt each value on its own. Thus, for example, an IP address of a user can be encrypted by using symmetric encryption and the resulting value can be put in the log entry in place of the original value. To avoid readability issues, the value can be converted to printable characters by using encoding such as base64. If the value is always encrypted with the same key, the result of the encryption process will be equal, which means that the log file can still be used for analysis. When necessary, the value can be decrypted to determine which user performed an action. However, since symmetric encryption is used, an attacker also has the key and can also decrypt all values. Additionally, a malicious administrator can also still gain access to all data, even if only a single user performed a policy violation that warrants decrypting the values.

Flegel[33] describes an approach that does not suffer from this problem. He encrypts values in such a way that they can only be decrypted in certain circumstances and not in general. For this, he uses a technique called „secret sharing“ which allows splitting data into multiple pieces, of which a sufficient number have to be present to be able to recover the original data. In this instance the secret sharing algorithm he uses is Shamir's threshold scheme created by Shamir[95]. However, the algorithm is used in a slightly modified version to achieve the solution. The original algorithm is based on polynomial interpolation and the basic idea is that it constructs a random polynomial and each piece is then a point on the curve of that polynomial. If a sufficient number of values are present, it is possible to recreate the polynomial that is described by those points. However, if too few points are known, it is impossible to determine which polynomial has been used originally. While the original version of Shamir's scheme generates pieces of the data, these are not helpful for protecting log data since multiple pieces are required to reconstruct the original value, but only one log file exists and so all pieces are in the same file. Furthermore, Shamir's scheme assumes that the pieces are sent to trusted parties, but Flegel[33] does not trust the recipients of the log file. Instead, the goal is that the data can only be recovered under certain conditions. To achieve this goal, the values that are to be protected, are grouped by their, possibly attack related, „event type“ and each event type is always encrypted with the same x-coordinate. Therefore, recovering the original data requires a sufficient number of different event types to occur and thus even an attacker is unable to recover the original data as long as the user does not trigger a sufficient number of event types [16].

The methods discussed here are not the only options to implement pseudonymization and other methods also exist. Dijkhuizen and Ham[25] provide a survey of anonymization methods and explain that some of them can also be used to provide pseudonymization.

3.4.3 Data Injection

Garfinkel and Spafford[35] explain a potential problem in logging setups that use log files, called data injection. In their example, an attacker injects fake log data into the log file that records administrative logins on a system. When administrators view the log file, they get confused and start chasing shadows because they believe that an attacker obtained administrative privileges. In reality, the problem was that the log file was writable for the attacker and thus the attack was as straightforward as writing to any normal file. While the attack in this example was mostly a

prank [35], Chuvakin and Schmidt[22] note that a malicious attacker might use the confusion and mount other attacks on top of it. Thus, a log data injection attack could be used to hide the real attack, for example, by faking more serious looking attacks. Administrators may then investigate the more serious attacks first and only later find the real attack and attacker [22].

Even without direct write access, an injection attack may be possible. For example, an application may log validation failures of user supplied values using a format like „Failed to validate value: $value“ with „$value“ being replaced by the user-supplied value. Suppose that the content of entries is not restricted sufficiently, thus allowing line feed characters (\n) to occur in the message content. This is the case in the syslog protocol, as discussed in Section 3.2.3. If the user, or an attacker, now asks the application to validate the string „test\nUser logged out: joe“, then the string will be written to the log file as shown in Listing 3.1. If an administrator later looks at the file, they will be unable to tell if the second line has been created by the application or if it was injected. Garfinkel and Spafford[35] show that adding prefixes like a log level or even a time stamp raises the difficulty of exploitation slightly. However, the problem still persists since the attacker simply has to come up with the correct string to match other entries and result in a valid looking log [35].

```
1 Failed to validate value: test
2 User logged out: joe
```

Listing 3.1: Log file showing log entry injection with user input „test\nUser logged out: joe“

A similar issue appears when insecure protocols are used to transfer log data across a network. The case with syslog systems using User Datagram Protocol (UDP) is discussed in Section 3.2.3. Using secure communication protocols, such as SSH or TLS allows mitigating this problem.

Data is also vulnerable when it is stored, as explained in Sections 3.2.3 and 3.2.4. These issues can generally be addressed by configuring strict access permissions to files, using encryption, and using MACs to prevent unauthorized modification.

However, depending on the complete system, data may also be injected much later. For example, if analysis software is used, an attacker may be able to connect directly to this software. If the analysis software is not protected sufficiently, an attacker can bypass the complete log data handling chain up until that point and hook in there. They can then feed fabricated data to the analysis system which then either causes it to report bogus attacks, also called false-positives, or not report attacks, called false-negative. Both types of issues cause administrators to lose confidence in the analysis system if they happen too often and thus they may stop responding quickly, or at all, to alerts or simply shut the system down since it is not working correctly [22]. Possible protection methods include using of firewalls to prevent network level access, and authenticating sending and receiving systems. However, exact protection measures depend on the type of system that is used and how data flows between different parts of the larger logging and analysis system. Authentication between systems can again be achieved by using SSH or TLS. This example shows that a logging system does not end when the log data is stored in a central system, but care also has to be taken when this data is later used by other systems. The purpose of these other systems is often to offer better insight into log data, but when administrators rely on them, they may no longer look directly into log files and thus these systems become just as important as the log files themselves.

3.4.4 Secure Storage

If log files are kept, it is important to ensure secure storage so that they can be referred to when necessary. Secure storage especially means that the storage where log file data is kept, needs to provide sufficient availability and integrity.

Preston[83] explains that threats to availability of log file data storage include hardware failure, disasters, human error, or deliberate deletion of data. Hardware failures differ slightly between storage media, but common storage media, such as hard disks, tapes, and optical disks, may all suffer either complete unreadability of data or they may contain incorrect data. For hard disks, a common solution to reduce potential data loss is to configure them in a RAID setup. However, such a system does not protect against user errors because actions affect all devices instantly and old data is not kept.

The simplest form is called RAID 0 and is actually not a redundant array, as the name would suggest, since the data stored on the array is stored only once. The goal of RAID 0 is to be able to use multiple disks with a single file system without having to add dedicated support for multiple disks into the file system's code itself and also to improve performance because all disks can be queried simultaneously for parts of their data.

Patterson, Gibson, and Katz[80] define RAID 1 as the simplest form that actually provides redundancy. It works by putting the exact same data on multiple disks so that as long as one disk of the array is available the data can still be used as if it were stored on a single drive. Once a disk fails, it can be replaced to restore full redundancy. There are also more complicated schemes like RAID 5 which allows for one disk to fail, regardless how many disks are in the array, or RAID 6 which allows for up to two failures [80].

However, simple disk failure is not the only failure mode of hard drives so even a RAID system cannot fully protect the data. Even if disks worked 100% correct until they fail a RAID system only duplicates the data and protects against hardware failure and it does not protect against operator mistakes or deliberate deletion of data [83].

Elerath and Pecht[30] show that it is also important to note that undiscovered data corruption due to latent defects can cause serious trouble. This is especially dangerous once a disk in the array fails and its data has to be restored from other disks. Missing or incorrect data on those other disks can happen due to various reasons like bad media, inherent bit-errors, high-fly writes and corroding media. The data stored on a RAID array has to be regularly checked for such defects by scrubbing the array. Scrubbing means that all data from all disks is read, verified, and if corrupted data is detected it is corrected using different, good copies.

Tapes and optical media suffer from similar problems, which also requires that any data is stored on more than one medium and that the integrity of the data is regularly verified. If data fails to verify, it must be renewed. Furthermore, such media are often off-line, which means they are accessed infrequently. This can be a problem since technology advances and old media may become unreadable by modern drives after some time. At some point it might even be impossible to find any devices to read old tapes just like it was for NASA and MIT [34].

Storage media must also be protected against unauthorized physical and electronic access. If old logs are deleted from a device, their content may still be stored on disk. In this case it may be possible to recover the data by using filesystem undeletion tools such as „extundelete“ and „Winundelete“ [5]. Thus, it may be necessary to additionally protect data by encryption, ensure that it is fully deleted when no longer needed, and properly discard or destroy used storage media.

4 Activity Auditing

Activity auditing is a special kind of logging with the goal of enabling an organisation to show their accountability by allowing auditors to validate and prove compliance with regulations [77]. They often contain information about detected attacks and faults, but also about activities in a system that are of interest. Interesting activities are often defined by regulations and may include users reading, creating, or changing certain, or all, information on a system. Regulations also often require that these logs are securely kept for a long time [21, 28]. Wickramage et al.[109] explain that in healthcare, audit logs can, for example, be used to demonstrate that sensitive patient information is only used in accordance with approved policy. However, even if the organisation does not have to prove compliance with regulations, Lins, Schneider, and Sunyaev[60] note that audit logs are still valuable by allowing an organisation to respond to abuse of the service that it provides and the data that it handles.

4.1 Linux Audit Framework

Jahoda et al.[51] and SUSE LLC[100] explain that one place to get audit logging information from is the operating system. The operating system handles file system, network, and general device access, as well as user permissions and process management. This means that the operating system is also an ideal place to record actions at these boundaries. The linux audit framework is an implementation of such a recording solution for the linux operating system.

4.1.1 Architecture of the Linux Audit Framework

The linux audit framework, shown in Figure 4.1, is implemented partially in the linux kernel, but it is also supported by various applications and tools in user-space. The administrator can use filters to configure when and which events should be logged. Such filters can be created by using the „auditctl“ command which can also read a configuration from a file [100]. Filters are separated into 4 filter lists which are checked for matching filter rules at different times in the execution of each system call. The first is the „User“ filter list, which can log an event when the system call of a user-space application arrives in the kernel. When a new task is created by the system call, for example, using „fork()“, the „Task“ filter list can be used to log an event. The „Exit“ filter list can log an event when a system call returns to the caller. According to the „auditctl“ manual by Grubb[40], this means that the call has been executed and the result is returned to the application. All events, generated by any of these three filter lists, are then passed to the „Exclude“ filter list which can exclude events based on their details, although no events are excluded by default. If an event passes the exclusion list, it is sent to the „auditd“ user-space audit daemon. Auditd is responsible for writing the event to a log file and forwarding it to the „audispd“ audit dispatcher daemon, which can relay events to other local or remote applications [100]. The audit log file can be searched through and analysed using the „ausearch“ and „aureport“ commands, amongst other tools [51, 116].

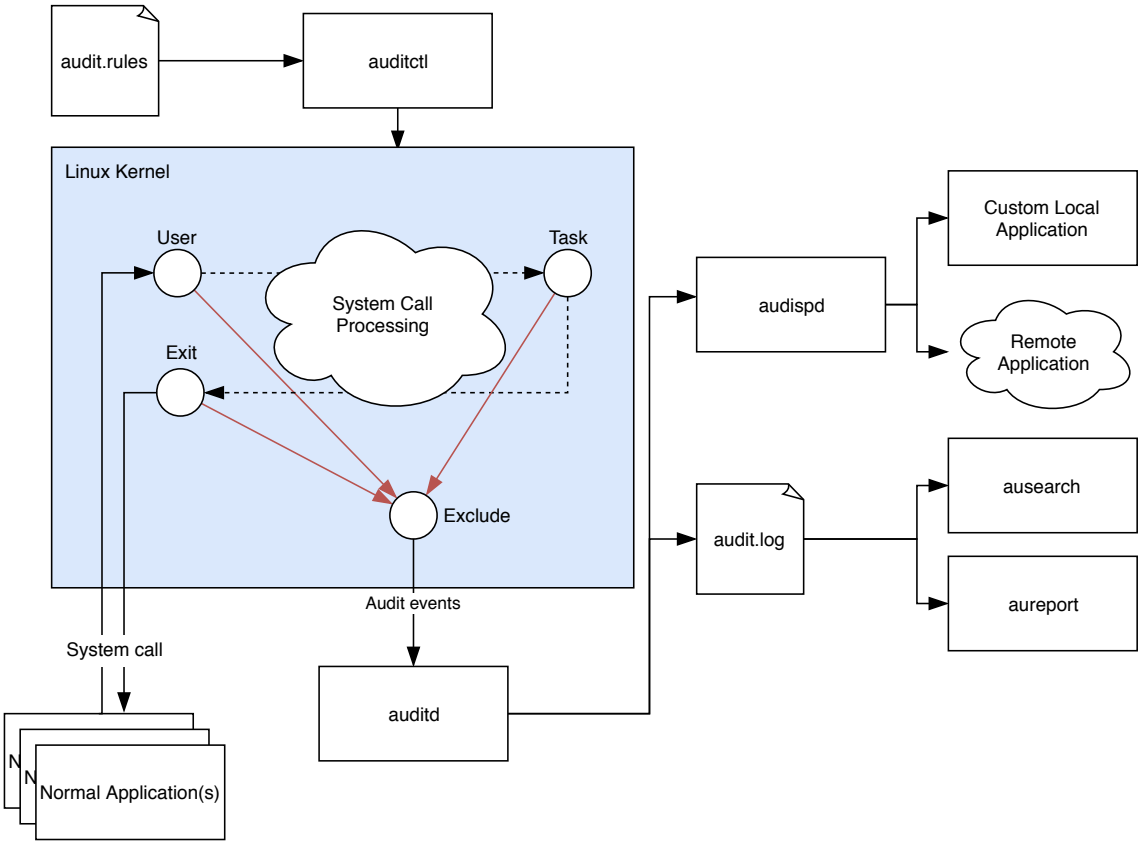

Figure 4.1: Diagram showing interactions between various components of the linux audit framework. Based on figures by Jahoda et al., SUSE LLC, Zeng, Xiao, and Chen[51, 100, 116]

4.1.2 Linux Audit Features

```
1 type=SYSCALL msg=audit(1364481363.243:24287): arch=c000003e syscall=2
  ↪ success=no exit=-13 a0=7fffd19c5592 a1=0 a2=7fffd19c4b50 a3=a items=1
  ↪ ppid=2686 pid=3538 auid=500 uid=500 gid=500 euid=500 suid=500 fsuid=500
  ↪ egid=500 sgid=500 fsgid=500 tty=pts0 ses=1 comm="cat" exe="/bin/cat" subj
  ↪ =unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key="sshd_config"
2 type=CWD msg=audit(1364481363.243:24287): cwd="/home/shadowman"
3 type=PATH msg=audit(1364481363.243:24287): item=0 name="/etc/ssh/sshd_config
  ↪ " inode=409248 dev=fd:00 mode=0100600 ouid=0 ogid=0 rdev=00:00 obj=
  ↪ system_u:object_r:etc_t:s0
```

Listing 4.1: Example events that may be generated by the Linux audit framework. Source: Jahoda et al.[51]

The linux audit framework can be configured to meet the requirements of various guidelines and certifications, such as PCI-DSS, and the Controlled Access Protection Profile and the Labeled Security Protection Profile [100]. To satisfy the requirements of these certifications, the audit framework can record various information about different event types. The „type“ field is the first field in each record and it describes the event type. Possible types include „SYSCALL“ to describe an event triggered by a system call, „CWD“ to record the current working directory at the time of the system call, and „PATH“ to record each path that the system call has accessed. Some commands, such as „open()“ only accept one path argument, while other commands may use more path parameters and thus generate more than one „PATH“ event. Listing 4.1 shows example log

entries for each of these types. The list below describes most of the fields that are used in a record of type „SYSCALL“ [51, 100]. More detailed information about the remaining fields and the exact values shown in the listing can be found in Jahoda et al. [51, page 177ff].

- The „type“ field contains the type of the event. In the case of Listing 4.1 the value is „SYSCALL“ since we are looking at the records of events of this type.
- The „msg“ field contains a message ID which identifies the original system call event. The message ID includes the date and time of the system call and a unique event ID. All events that are generated due to one system call issued by an application receive the same message ID. Thus, it can be used to group information about each system call.
- The „syscall“ field records the type of the system call. This is recorded as a number and can be looked up to reveal the name of the system call. Listing 4.1 shows an example where this value is „2“ which corresponds to the „open()“ system call.
- The „exit“ field records the success and exit status of the system call.
- The „a0“ to „a3“ fields record first four arguments of the system call. Which exact values are logged, depends on the system call. For example, for an „open()“ call, the first argument is a path, but the log only includes the memory address of the passed path name. The real path is recorded in a separate event of type „PATH“. The next arguments to „open()“ are flags, and the mode. The 4th argument is not used by „open()“ and is thus 0.
- The „items“ field records the number of „PATH“ records generated as part of this event.
- The „pid“ and „ppid“ fields record the process ID of the process that issued the system call, as well as the process ID of the parent of that process.
- The next fields record various („real“, „set“, „file system“, and „effective“) user and group IDs of the process, as well as the audit ID. The audit ID is created when a user logs in and it is passed down to all child processes. It stays the same, even if the user changes their identity, for example, by switching to the „root“ user. This makes it possible to trace any action back to the original login user.
- The „tty“, „comm“ and „exe“ fields record the terminal used to start the application, the application name and the path to the application executable.
- Finally, the „key“ field contains an administrator specified key that allows administrators to identify which rule generated an audit event.

„PATH“ events similarly record the message ID and special information, such as the full file or directory path, the device that contains this path, the file or directory permissions, and the file owner user and group IDs. „CWD“ events also contain the message ID and the current working directory of the process that issued the system call [51, 100].

4.1.3 User-Space Tools

To support the „audit“ kernel module, there are various user-space applications, some of which have already been briefly mentioned in Section 4.1. The „auditctl“ application can be used to enable, disable or lock audit configuration, control the failure mode when there are issues during audit data collection or transmission, and control rate and backlog limits [100]. A locked audit configuration means that the kernel will no longer accept changes to the configuration, including

the audit rules, and any attempts to change the configuration will trigger audit events and be denied. It will also prevent disabling the audit system unless the operating system is rebooted [40]. Apart from changing these basic parameters, the command can also be used to configure various types of audit rules. Audit rules can either be control rules, which allow the audit system's behaviour or configuration to be changed, file system rules, which allow auditing access to a particular file or directory, or system call rules, which have already been discussed in Section 4.1.2. File system rules are rather basic and can also be expressed with system call rules that simply match system calls that refer to the path [40].

Making sense of audit logs may be difficult since they often include numerical values for various fields. The „ausearch“ command can be used to search for records in the audit log file, and it can try to convert some values into human readable text. It is best used when one is interested in a particular event. If the goal is to gain an overview, the „aureport“ tool provides various types of summaries. The report can be limited to certain time frames and to specific types of events. For example, aureport can be instructed to display information about specific files or users, but it may also limit the report to more general groups of events, such as all executions of applications [100].

The audit system can not only be used to track all activity on a system, but also only that of a specific process. This is similar to the „strace“ utility (see Section 3.3.3), but it logs the information available to the audit system and also logs these events into the audit log file. While it is possible to create the rules manually, the „autrace“ command provides a shortcut. It allows administrators to create the necessary rules, run an application, and remove the rules again afterwards. However, it is important to note that using this technique may interfere with other auditing rules and thus other rules should be removed first [100]. Therefore, using this approach in a real environment leaves the system open to potential misuse during the use of „autrace“.

4.2 Linux Security Modules

The linux audit framework is not the only option to obtain audit data in a linux system. Morris, Smalley, and Kroah-Hartman[67] describe Linux Security Modules (LSM) as a subsystem that aims to simplify security support in the linux kernel by delegating various decisions to external modules. During its creation, various parties were interested in extending the access control features of the kernel, while others also wanted other functionality related to security auditing or virtualized environments. Due to complexity and security concerns, the implementation of LSM focused on simplicity. This means that it primarily supports restricting access to various resources by calling hook functions just before any access is permitted. The module that implements an LSM hook can then decide if the access should be allowed or denied. For performance reasons, as shown in Figure 4.2, the kernel returns early when any one of the various checks that are run during handling a system call, fails. However, this also means that if access is disallowed by the kernel for any reason, the LSM hook is not called. Checks that are run before the LSM hook include lookups if the accessed path exists, error checks, and regular Discretionary Access Control (DAC) permission checks [67]. This is in stark contrast to the audit framework, which is discussed in Section 4.1, and which always generates audit events right at the start of a system call handling, even if the call may later be denied.

One notable exception to the rule that LSM can only deny access is that it was also important to rewrite POSIX capability support as an LSM module. Capabilities allow a process to be special, for example, by having the capability to override DAC restrictions. If the kernel were to return early when the DAC check fails, it would be impossible to override this behaviour with an LSM hook since this hook would never be run. Therefore, there is an exception in the DAC code so that if the regular DAC check fails, the code checks for a special DAC override hook which can change this decision. Placing similar hooks in other functionality in the kernel would have potentially

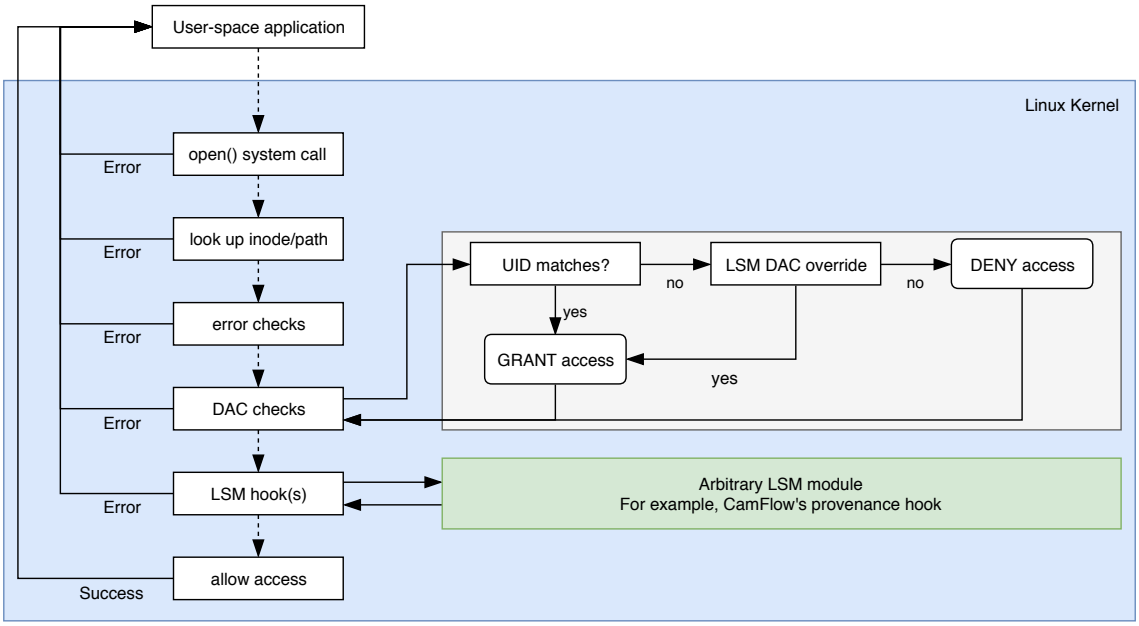

Figure 4.2: Diagram showing architecture of LSM and the integration point of CamFlow provenance capture using LSM. Based on figures by Morris, Smalley, and Kroah-Hartman, Pasquier et al.[67, 79]

required extensive changes to the code base and would also have increased the complexity of the LSM interface for modules [67].

Even though LSM only calls hooks for otherwise allowed requests, Pasquier et al.[79] use it to capture provenance information. Data Provenance refers to information about how data came into its current state and who accesses certain information. Their system, called CamFlow, hooks into various functions in the Linux kernel that deal with the items they want to capture information about. This includes interactions between processes and kernel objects such as files or sockets which they can intercept using the LSM framework. The LSM interception point is marked in green in Figure 4.2. They also intercept network traffic by using the NetFilter framework. Additionally, CamFlow provides an API for user-level applications to send application level provenance data to. This data is merged with the other sources before then being streamed to a dedicated application, written by the user, that processes and stores it. To ease analysis, they also provide information about associations between data items. Items are linked by their usage of system resources such as file descriptors. It is also possible to configure which information shall be collected by, for example, specifying criteria such as paths, network addresses, user or group ids [79]. While most of this is relatively similar to the way the linux audit framework works, they do not discuss the reasons for implementing their own solution based on LSM, compared to using the audit framework.

4.3 Log Analysis

Making sense of the content of log files is called log analysis and this process involves studying log files to discover interesting events. There are various reasons for performing log analysis, such as analysis after a problem or continuous analysis during operation [22, 53]. Different goals and usage patterns may also call for different kinds of automation. Automation generally helps to respond more quickly or even proactively to events, but an automated monitoring solution may require continuous maintenance and adjustments [22, 53].

4.3.1 Goal of Log Analysis

A common goal for logging is to only look at logs when something goes wrong and to try to figure out what is happening and why. This often involves manual review of log files of particular services to discover the underlying cause of the malfunction. The log file may contain pointers that show what the problem could be, it might contain an error message, or it might even contain nothing obvious at all. Looking for an error message is a form of looking for known-bad log entries. In this case the known-bad pattern happens to be something like „error“ or „failed“, but this idea can also be generalized slightly to include events like the creation of new users, configuration changes, or sometimes even known security exploits. An analyst may analyse logs to look for information left behind by such events and check if a particular security issue has been used to attack a system or not [22].

Such infrequent log analysis is often caused by an organisation only performing analysis reactively, when a problem has already been discovered by other means. For example, the security issue may have come up in a security advisory by the vendor and after installing the updated version, the administrators checked to see if there have been any undiscovered incidents. While this is a valid usage scenario for logs, they can provide greater benefit if analysis is performed proactively. This way, it may be possible to spot signs of impending problems or discover problems that users dismiss at the beginning [53]. Users may assume that, for example, a website is currently being modified if it does not work properly, but after a few days someone notifies the administrators that it is still not working. The administrators then look into the log files and notice that the application has been crashing every time, but apart from that one person, everyone else simply dismissed the problem. Depending on the service, those users may stop visiting the site or the continued problems shine a bad light on the organisation providing it. With proactive analysis it may be possible to catch such issues when they start happening and resolve them before they cause too much damage.

Another goal may be obtaining statistical data about events on a system to influence business decisions. For example, an analyst or, more likely, a program, such as the visualization tool created by Humphries et al.[49], can determine how many requests a web server receives over a time period. This can then be plotted in a graph to see if the number of users is staying constant, increasing or decreasing. Similarly, one can look at which browsers users used to decide if a website should be tested on a particular browser or not [49].

4.3.2 Manual Log Analysis

Manual log analysis is the simplest and most traditional form and also one of the reasons why log analysis has often been treated as a low priority task [53]. The obvious form of manual analysis is reading a log file line by line. This is generally done by using a text viewer for text-based log files, or a dedicated viewer for non-text log files. While a text editor may work fine for smaller logs, on busy systems log files can easily reach a size of multiple gigabytes with a huge number of lines and editors are often not built to handle such large files. Another obvious issue is that if the number of lines is high, reading all of them is a very time-consuming effort. This can easily lead to getting frustrated, skipping or only skimming lines and thus missing important information, or even deprioritizing analysis and only performing it when absolutely necessary [22].

A slightly less frustrating approach which is supported by many log viewers, such as „less“, is searching or filtering entries. The analyst still has to know what they are looking for, but they no longer have to read every line. For example, if there is an issue with the SSH daemon, they can filter the log file to only display lines that mention the keyword „sshd“, which is a common name for the SSH daemon on a UNIX or Linux machine. Looking at the filtered lines, there is much less clutter from other services on the system and thus it is easier to spot lines that do not look right

or to look for lines that happen around a specific time. Such filtering may either be performed inside „less“ or by using the „grep“ command. „grep“ takes a file and a pattern and looks for any line in the file that matches the pattern. These lines are then output to the standard output, which can again be fed into other commands or saved in a file. Using this technique a log file can be quickly reduced to only include mostly interesting lines and the result may for example be used, kept, archived, or sent to someone else [22].

However, the pattern accepted by „grep“ is not only used for simple string matching, but it can actually include a regular expression. Regular expressions allow users to specify that parts of the pattern may occur multiple times, or that a part may be either of a list of sub parts. Thus, a log file can be easily filtered to include all lines mentioning either „ssh“ or „telnet“ by using the pattern „ssh|telnet“ with the pipe denoting the alternation [22]. Regular expressions also support placeholders to denote parts of the pattern where the content does not matter, as long as it is matched by the placeholder. The dot character is a placeholder that matches any single character and the star repetition operator modifies the item in front of it and means that this item may occur zero or more times. Together with the dot placeholder, this allows matching an unspecified number of arbitrary characters. For example, „sshd.*refused“ will only match lines that contain the string „sshd“ before the string „refused“ and the content between these two does not matter.

Furthermore, common UNIX and Linux tools such as „awk“, and „sort“ allow users to quickly construct a command that extracts parts of a log message and then only shows each unique part once. This can be used to, for example, generate a list of devices that have logged to a log file [22].

Sometimes, administrators also refer to log files when they are dealing with an ongoing issue and need to check what is happening on a system right now. In these instances it is often useful to see log data as soon as it is written to the log file. This can be achieved by using the „tail“ command. If tail is only called with a file as an argument, it displays the last few lines of that file. However, if the „-f“ option is used, it also follows the file and outputs new lines that are written to it. This is already quite useful by itself, since it, for example, allows an administrator to trigger some action, such as opening a website or submitting a form on a website, and then watch the log to see what messages happen to be generated. In addition to the log file, they can watch what happens in the browser and correlate slow loading times or error messages with the log file. Similar correlation can be performed, for example, if users are unable to send a mail via a mail server. Mail servers often handle lots of mail and thus have many lines in their log file. If the user is having issues, it may help while debugging the problem to call them, follow the log file, tell them to perform the action and watch the file when they say that they just pressed the button. The log file might, for example, reveal that they did not configure their client correctly. The client either does not even connect to the correct server, or it does not authenticate at all or with an incorrect user name. This correlation may be difficult to recreate from just a log file alone since getting accurate timing information for human actions may be difficult. Even though following a log file may be very useful, a likely problem is that, on busy systems, the log file receives so much data that it is impossible to follow it and manually pick out the pieces that are required for debugging the problem. Luckily, it is again possible on UNIX and Linux systems to pass the output of „tail“ to other commands and thus create a command chain such as „tail -f /var/log/mail.log | grep 1.2.3.4“ to only show new lines that match the supplied pattern, which in this case could be an IP address.

Similar approaches can be used, when log data is stored in a database. If it is stored in an SQL database, standard SQL queries can be executed to search and retrieve specific entries. More complex queries can be used to group log entires and provide an overview over the different types of entries that are in the database [22].

4.3.3 Simple Automated Monitoring

Manual log analysis is very important, but it is time-consuming and for sufficiently busy systems manual analysis alone is impossible. As explained in Section 4.3.1 it is important to continually monitor log content to detect potential problems or intrusions early. A simple solution to this problem is to configure the logging software, such as a syslog daemon, to write different types of log entries to different files. For example, the most important log level, possibly called „critical“, „highest“ or „high“, likely contains entries that are of interest to administrators [22]. If these messages are written to a dedicated file, administrators can regularly check this file, or they can create a cron job that sends them a mail when the file contains new data. If the software supports it, such messages may even be sent directly via email, in addition to being written to the log file.

Such a solution may potentially increase the efficiency of log monitoring since it filters out less interesting or less important messages. However, not all important and interesting messages are always marked with the highest log level by the application that creates the log entry. For example, a web server expects that some files may not be found and thus cannot be served to the user. Therefore, such an event, may only be logged as an error or warning, rather than a critical message. Similarly, an email system has to deal with spammers that try to send mails to random addresses and as such, the event of a mail being sent to a non-existent recipient may also only be classified as an error. However, the dedicated log only includes critical messages and not errors and administrators will not be notified about them. A solution to this problem is to summarize the content of other logs so that administrators can look for abnormalities or trends [22]. Such trends could be as simple as monitoring the bandwidth used by each machine or user, looking at a daily list of the top event types, or monitoring the total number of requests in web server logs. If there are any abnormal or abrupt changes, further investigation can be conducted to search for the cause.

An automated version of the „tail -f | grep“ idea from Section 4.3.2 is provided by the „tenshi“ application. The manual by Barisani[6] describes tenshi as a software that can be configured to watch a single or multiple log files, collect log entries in multiple queues and regularly send emails to administrators with the collected entries for each queue. Each queue may also have different notification intervals and different recipients. Entries are assigned to queues based on regular expressions that describe the format or content of the entry. Uninteresting information can also be removed from the entry by using a grouping operator in the regular expression, which is simply a set of round brackets („()“). Masking such information is helpful because each email notification will group equal log entries together and only display the entry once in addition to the number of occurrences of this particular entry. If information such as port numbers, IP addresses, or process IDs is not masked, each entry will be different and this grouping of messages will not work. For performance optimization, regular expressions can also be grouped using the „group“ command to instruct tenshi to skip the content of the group if the group expression does not match. Listing 4.2 shows a trimmed example configuration using these features and Listing 4.3 shows an email notification example.

The configuration in Listing 4.2 results in messages regarding the login and logout of users to be recorded without the user name. Therefore, the notification only shows the number of occurrences of these messages. Some additional messages are also included in the queue and, consequently, in the notification. At the end of the „ipop3d“ group, all unmatched messages are thrown away by using the special „trash“ queue. They are still in the original log file, however, no notifications are generated for them. At the end of the configuration file, all remaining (unmatched) messages are assigned to the „misc“ queue. This allows administrators to react to previously unknown, unanticipated, or changed messages and update their configuration accordingly. Without this, the messages would likely get overlooked unless administrators happen to notice them in the log files by other means. Command executions that use the „sudo“ command to gain root permissions, are sent to the „critical“ queue, which is configured to instantly create an email notification when a

message arrives in the queue. The created notifications look similar to the example, except that they only contain a single line.

```
set queue mail     tenshi@localhost sysadmin@localhost [0 */12 * * *]
set queue misc     tenshi@localhost sysadmin@localhost [0 */24 * * *]
set queue critical tenshi@localhost sysadmin@localhost [now]

group ^ipop3d:
    mail ^ipop3d: Login user=(.+)
    mail ^ipop3d: Logout user=(.+)
    mail ^ipop3d: pop3s SSL service init from (.+)
    mail ^ipop3d: pop3 service init from (.+)
    mail ^ipop3d: Command stream end of file, while reading.+
    mail ^ipop3d: Command stream end of file while reading.+
    critical ^ipop3d: Login failed.+
    trash ^ipop3d:.+
group_end

critical ^sudo: (.+) : TTY=(.+) ; PWD=(.+) ; USER=root ; COMMAND=.+
misc .*
```

Listing 4.2: Trimmed example configuration file for the „tenshi“ log monitoring application. Based on example from the tenshi website [7]

```
host1:
    79: ipop3d: Login user=___
    74: ipop3d: Logout user=___

host2:
    30: ipop3d: Login user=___
    30: ipop3d: Logout user=___
    19: ipop3d: pop3 service init from ___
    12: ipop3d: pop3s SSL service init from ___
    1: ipop3d: Command stream end of file while reading line user=??? host=
    ↪ bogus.domain.net [192.168.0.1]
    1: ipop3d: Command stream end of file, while reading authentication host
    ↪ =bogus1.domain.net [10.1.7.1]
```

Listing 4.3: Example notification from the „tenshi“ log monitoring application for the „mail“ queue using the configuation in Listing 4.2. Based on example from the tenshi website [7]

4.3.4 Problems of Manual Log Analysis

An obvious problem with manual log analysis is scalability. With quickly growing log files it is impossible for a human to analyse each message and filtering the log data by removing uninteresting lines is a double-edged sword. If the filters are very strict, the analyst only notices very obvious problems. On the other hand, if they are not strict enough, the result still contains a lot of noise. Filtering the noise from the signal then again requires manual analysis either by applying stricter filters or reading all the data. Furthermore, filtering log messages using patterns bears similar problems. A pattern may match too little or too much by accident. While matching too little is generally easy to spot because there is too much output, filtering too much is not as easy to notice. A possible solution for this problem when creating a new filter pattern is to first filter the log for everything except the pattern. Thus, the analyst can look at what would be filtered and only if the output is what they expected, they can reverse the filtering and keep everything except for the lines matched by the pattern.

While such an approach may work well for manual analysis where the analyst has a log file and filters it, this is not as simple for automated analysis systems like „tenshi“, which is described in Section 4.3.3. With such a system, the patterns are written to a configuration file and are used for analysing new log data. While it is possible to verify that the pattern only matches what is expected when it is created, this may not remain true in the future. Ya et al.[111] note that software updates and configuration changes may change the log format either drastically or only slightly [111]. For example, a software may log simple messages, such as „Login user=$username“ with „$username“ representing the user name of each user that logs in to the application. At some point the developers of the application may decide to improve the logging capabilities and either completely change the message or append additional data at the end. If the message is changed completely, a configuration like the one in Listing 4.2 would no longer match the login message. Instead, the last line of the group would match, which in this case says that all other messages should be ignored. If additional data is appended to a log line such as this, it is important to look at how the pattern works. In this case, the pattern uses the „.+“ regular expression, which matches an arbitrary string with at least a single character. Specifically, this means that if the message were to be changed to read „Login user=$username result=failed reason=too-many-attempts“, the regular expression would still match the line and include the additional information. The configuration also specifies that the part matched by this subexpression should be hidden from the report. This makes it arguably unlikely for the change to be noticed. It is thus important that patterns are crafted carefully and that they are regularly reviewed to ensure they still work as expected.

A related issue is that software may create different log data for different inputs. For example, email software may log the subject and sender of an email to the log file. As discussed in Section 3.2.1, the syslog protocol only guarantees correct transport for rather short messages [36]. Developers that are aware of this limitation may build software that automatically breaks log messages into multiple lines if they become too long. Such a feature may not be noticed during testing and initial pattern creation, since in the case of an email system, it may, for example, only be triggered when someone sends a message with a very long subject. This may raise similar concerns as the data injection issue discussed in Section 3.4.3, except that in this case the application decides when the log message is split instead of this place being dictated by the message content itself. However, a clever attacker may be able to deduce where a log message is broken apart and create a corresponding input. Similar to the simpler data injection case, this may again lead to false-positive or false-negative alerts generated by the monitoring solution.

4.4 Automated Analysis and Anomaly Detection

As shown in Section 4.3.4 manual and simpler automated analysis methods require human intervention to react to changing circumstances. Ya et al.[111] explain that these solutions cannot adjust their configuration automatically if the log data format changes and that they also need to be reconfigured to correctly deal with new or removed applications. Solutions to these problems include automated log format extraction, and log file or network based anomaly detection methods.

4.4.1 Automated Log Format Extraction

While log monitoring and analysis using patterns may be a good solution for many contexts, creating and maintaining these patterns is often difficult. An analyst has to manually inspect the log file and reason about the log data to come up with a useful pattern to match a variety of messages. Different applications, devices and vendors can use wildly different formats and often manual format extraction is time-consuming and costly [22, 111]. Automated approaches reduce manual labor by using various algorithms to transform the log file into a list of patterns. These algorithms can be run regularly to adjust the pattern list for changing log data.

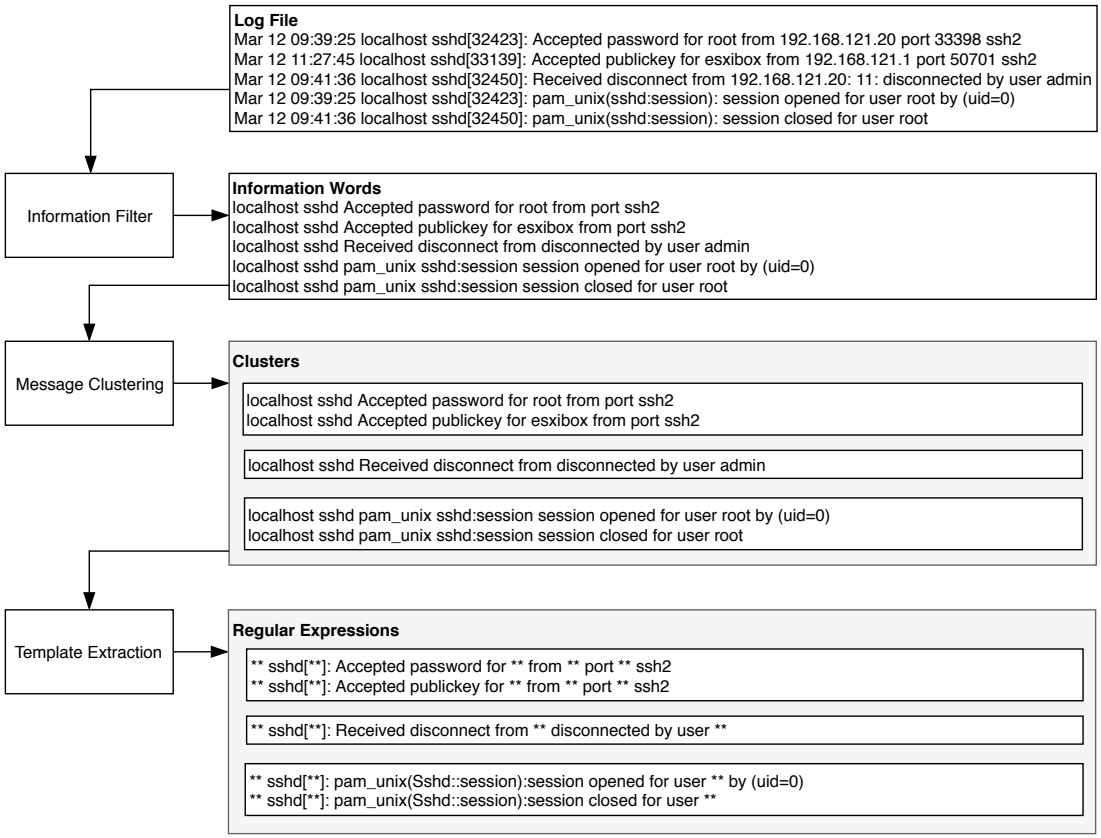

Figure 4.3: Architecture of the Log Template Extraction (LTE) approach, showing processing of an example log file. Based on diagram by Ya et al.[111]

Log Template Extraction An automatic approach created by Ya et al.[111] is called Log Template Extraction (LTE). It can work with log data from a variety of different sources and thus different log formats and create a list of formats that describe the various different messages. LTE requires log data as its input and returns a list of inferred log message formats. It is implemented as three distinct modules which pass the data from one to the next as shown in Figure 4.3. The first module removes highly specific information such as timestamps or IP addresses. Afterwards, the second module clusters the cleaned log messages based on a Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. The last module extracts a template for the messages from each cluster by looking at the distributions of words in the cluster [111].

The information filter module removes common information that depends on the runtime environment of the log generator. For example, in the case of network logs, this information includes dates, timestamps, and IP addresses. To perform this cleanup, the module has a list of regular expressions that match each of these specific data pieces [111]. While the reason for performing this step is not explicitly stated, it may be done to prevent the later modules from misinterpreting the data and keeping pieces of this information in the log format. It is conceivable that the later algorithms might, for example, interpret the dots in an IP address as parts of the log message format rather than treat the number and the dots as one variable entity.

The message clustering module takes as input the cleaned log message from the information filter module. Clustering is performed by using a DBSCAN algorithm. This algorithm tries to find clusters, but, contrary to some other clustering algorithms such as k-means clustering, DBSCAN does not rely on a predefined number of clusters. Instead, Schubert et al.[94] explains that it detects clusters by using two parameters, which are a neighbourhood radius ε and a minimum cluster size $minPts$. If a point has more than $minPts$ neighbours within the ε radius, it is considered to be a

core point. Points that do not have a sufficient number of neighbours are considered to be noise. Such a noise point may still become part of a cluster if it is part of the neighbourhood of another *core point*. In the case of Log Template Extraction (LTE), while they do not provide a detailed description, they explain that DBSCAN is applied on the different words in each input message. An input message is a single, cleaned log line. The result of running the algorithm are multiple lists of messages that each form a cluster with a common log message format. An example is shown in Figure 4.3 [111].

The template extraction module deals with a single cluster of log messages at a time. These messages use the same log format and are equal to each other, apart from the values substituted in the format. To find such variable parts, LTE generates a set of template words for each cluster. Template words are those words that appear in each message in a particular cluster. Words that do not occur in the set of template words are the variable parts of a log message and are replaced by a wildcard. The result of this operation is a pattern that describes the log format. This approach can obviously result in too many text parts being detected as template words. For example, paths that are equal in each message, such as `/usr/local/`, are incorrectly classified as template words and not replaced by a wildcard in the pattern [111].

Parallel Log Parsing He et al.[45] describe a log parsing algorithm called Parallel Log Parsing (POP). Log parsing is similar to format extraction, except that it also extracts the variable values from each log message into a list of key-value pairs. Like LTE, POP starts by having a short list of regular expressions that remove unnecessary data, however, afterwards, they employ some heuristics based on the message length. They define message length as the number of tokens in a message, with a token being a string that is terminated by whitespace. For example, the message „Verification succeeded for X“ contains 4 tokens. Messages with the same message length are grouped since they are likely to be messages with the same log format. Obviously, messages may actually be different even if their message length is equal. To detect such messages, POP looks at each message from a group and goes through each token position. The groups are then split such that for each token position, all messages in the group have the same value, except when the value is determined to be a variable value because its cardinality is above some configurable threshold values. Afterwards, the tokens can be concatenated to form a pattern. Tokens that were detected to be variable values are replaced with wildcards in the pattern. Finally, some groups may have been split too much, for example, because a variable value contained whitespace. To resolve this, groups are clustered, using the Manhattan distance between two log event texts to determine their similarity. Similar groups are merged [45].

Format Extraction using Source Code While many approaches to automatically generate log format patterns work by analysing log files, this is not the only way. Xu et al.[110] describe an analysis system that extracts the log message formats from the program source code. They note that open source software is present in many internet systems and, therefore, they believe that depending on the availability of source code does not pose an issue. Contrary, He et al.[45] dismiss source code analysis because they believe that source code is often inaccessible, especially for third party libraries [45]. The usability of this approach, therefore, depends on the used software and the environment.

Extracting log format patterns from source code is relatively simple for applications written in a C like language since it is likely that the application uses a variant of the „printf“ command. This command accepts a format string as well as a number of arguments that are substituted for placeholders in the format string. The format string often directly translates to the log format and the placeholders also specify the type of the variable that should be substituted. Available

types include integers, floating-point values, and strings which can be replaced with corresponding regular expressions in the log format [110].

Other languages, such as object-oriented languages like Java, present a bigger challenge because the log statements may be further decoupled. For example, a Java program may use a logging library such as „log4j“ or string concatenation to generate a log message with the string returned by a „toString“ method of an object. Just like other methods in an object-oriented language, the „toString“ method can be inherited from other classes and must not necessarily be defined in the object's own class. This „toString“ method is used to return the internal state of an object as a string for various purposes, one of which is writing it to a log file. Since objects may have multiple internal variables, this string may include any number of them and thus has an internal format itself. It may be important to reflect this nested format in the log format pattern and, therefore, the pattern generator needs to understand the source code sufficiently to extract it and work with the existing language features, such as inheritance of methods [110].

The implementation of Xu et al.[110] solves this „toString“ problem as shown in Figure 4.4. First, it gathers all partial formats in a list and then recursively replaces each occurrence of a non-primitive type with the format of that type. It is also possible that a class has multiple subclasses that implement „toString“ methods with different formats. In this case, the partial format is duplicated in the list for each subclass and each format is then changed to match a specific subclass as shown in Figure 4.4. Primitive types are, for example, simple strings, integer and floating-point numbers, while non-primitive types are, for example, classes that implement a „toString“ method. Once all such types are replaced, the format only contains primitive types which are replaced by regular expressions to result in a log format pattern [110].

4.4.2 Log File Based Anomaly Detection

Anomaly detection allows administrators to automatically discover potentially interesting events in a system or network. Log file based anomaly detection is a good starting point for multiple reasons. First, nearly all applications and devices generate some kind of log file or log data. Therefore, the information source is readily available and only needs to be put to use. Secondly, it can often be performed on existing log files as well as future ones. This allows an organisation to also analyse past events when the anomaly detection itself is improved in the future or when it is first implemented [22]. He et al.[46] show that anomaly detection algorithms often rely on a feature representation in form of a numerical matrix. The first step to creating such a feature matrix is extracting the log format as discussed in Section 4.4.1. When the log format is known, it is possible to count the occurrences of each specific log format. Furthermore, the format can be used to extract the parts matched by the wildcards and these values can then also be used as features, either directly or after additional processing like counting occurrences [46].

Anomaly Detection using strace Abed, Clancy, and Levy[1] created a Host-based Intrusion Detection System (HIDS) which implements log file based anomaly detection. It detects anomalies in the system calls issued by applications that run in containers, such as docker or LXC. Containers provide one-way isolation between the applications running inside the container and those on the host system. While the applications on the inside cannot see applications running on the outside, applications that run on the host system can directly interact with those inside the container. This allows using the „strace“ tool, which has been described in Section 3.3.3, to log the system calls used by applications inside the container. While strace can log a variety of information, further analysis only uses the names of the system calls that are executed so only these are extracted from the output. The extracted system calls are collected in a sliding window of size 10. This means that the application keeps a list which contains at most 10 items and when a new item is added, if there are already 10 items in the list, the first one is dropped so that the total size remains at 10.

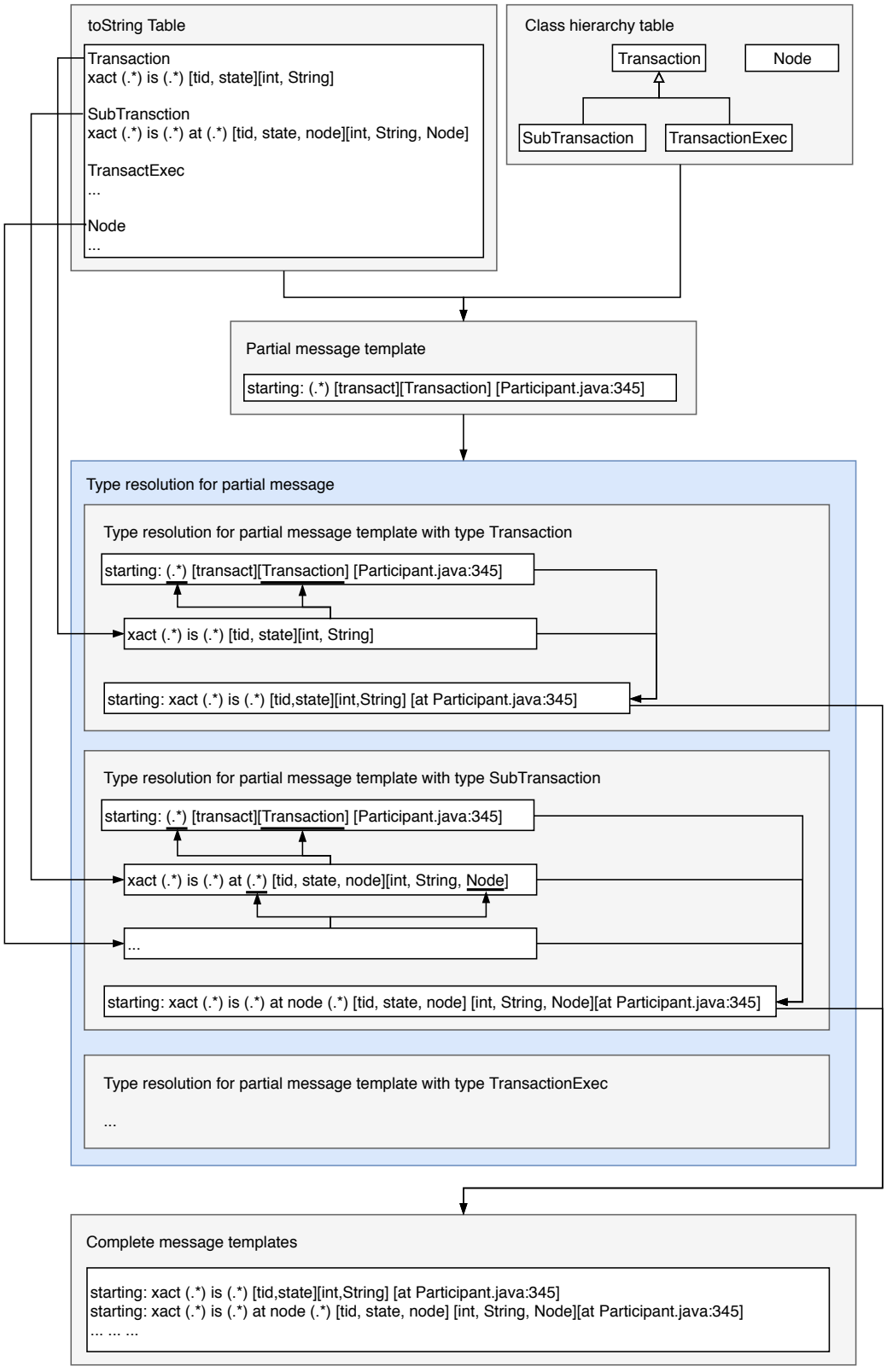

Figure 4.4: Construction of log format patterns with source code analysis showing handling of multiple subtypes. Source: Xu et al.[110]

This window is then passed to a classifier which can work in either training or detection mode. In training mode, the classifier adds the window to a database that contains „normal“ behaviour of the application. This database keeps track of the frequency with which each window content appears. If the same window content appears multiple times, a counter is incremented. The application that shall be monitored is then used in all possible „normal“ ways and the system calls issued by this usage gets recorded in the database. Once all good behaviour patterns have been recorded, the training mode is turned off [1].

In detection mode, the classifier checks that each window occurs in the previously trained database. If a window cannot be found, it is treated as a mismatch with the learned-good behaviour and if a sufficient number of such mismatches occur within a specified time frame, an anomaly has been detected. Detection mode also includes a continuous training mode. Similar to the regular training mode, this mode records all events to a temporary database. If no anomaly has been raised during a time frame, this temporary database is merged into the „normal“ database. This allows the system to slowly adapt to changes in application behaviour without requiring a completely new training session [1].

Machine Learning He et al.[46] believe that adoption of log-based anomaly detection is slowed down by the fact that, prior to their work, there were no open source implementations available. While it is common in academia to publish descriptions and sometimes also sample implementations of algorithms, fully working implementation that can be used without modification are much more rare. Furthermore, to compare different methods, they have to be reimplemented from the relevant publications with enormous effort. Implementing anomaly detection methods from papers proves difficult because there are no test oracles to verify the correctness of machine learning algorithms once they have been implemented. By providing open source implementations of the methods, others can use these as a starting point for future improvements or simply use them the way they are [46].

He et al.[46] compare six anomaly detection approaches using different types of input data. They start by extracting the log format and extracting features of events into numerical values. These values are calculated by various rules and the resulting sets are then passed to machine learning algorithms for training and analysis. Rules to group the values include fixed windows, sliding windows, and session windows. Fixed windows are based on timestamps and each window includes all events that happened during the specific time span. Sliding windows are also time based, but each window still covers a part of the last window's time frame. Session windows are based on identifiers rather than timestamps. All log events that carry the same identifier in their log message are put in the same window. The events from each type of window are then counted according to their event type and the results are recorded in a vector that holds the count for each possible event type. If an event has not occurred in a window, the respective event count in the vector is 0. A vector of $[0, 0, 2, 3]$ thus describes a window from a log file with 4 different types of events. The first two event types did not occur during the particular window, while the last two types were seen two and three times, respectively. All individual event count vectors grouped together form the event count matrix [46].

The event count matrix is then fed to various machine learning algorithms. The comparison includes supervised algorithms, such as logistic regression, decision tree, and Support Vector Machine (SVM), but also unsupervised algorithms, such as log clustering, Principal Component Analysis (PCA), and invariants mining. Logistic regression is a statistical model that estimates the probability of all possible states of an input. In this case the set of states only includes „normal“ and „anomaly“. The algorithm is supervised, which means that it must be trained with labeled data before it can be used for analysis. Once it is trained, it calculates all probabilities and He et al.[46] treat an instance as anomalous when the probability for an „anomaly“ state is at or above 50%.

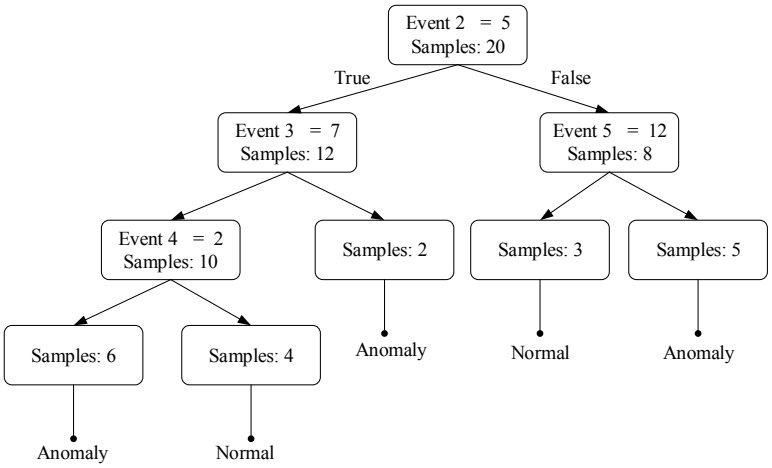

Figure 4.5: An example decision tree showing the decisions to classify an event vector as „normal“ or as an „anomaly“. Source: He et al.[46]

A decision tree is a tree structure that uses simple decisions to determine the state of an input. Figure 4.5 shows a decision tree that differentiates between „normal“ event vectors and anomalies. During training, each value of the input vector is analysed to determine the value that provides the most information gain. This value is selected for the root node and it splits the entire data set into two subsets. In each subset, the respective next best value is selected to again split the current set into further subsets. Once all vectors in the set belong to the same class, the tree nodes become leaf nodes and point to the state of these vectors, which in this case can either be „normal“ or „anomaly“. The benefit of the decision tree is that the visual representation of the tree clearly shows the decision process that the machine takes to arrive at a conclusion [46, 110].

The third supervised learning method is called Support Vector Machine (SVM). It constructs a hyperplane in a high-dimensional space that separates the various classes. In this case, there are two classes that are separated by the hyperplane. The two classes are the two states, which are „normal“ and „anomaly“. Calculating the location of an input vector in the high-dimensional space can then be used to determine to which class it belongs [46].

Apart from the supervised methods, He et al.[46] also compare unsupervised methods. Different from supervised methods, unsupervised methods do not require labels, but rather rely on detection of outliers in the data set. The first unsupervised algorithm is log clustering using the LogCluster method by Lin et al.[59]. LogCluster uses a knowledge base that contains known logs and for new log entries it checks if the entry is already part of the knowledge base. This knowledge base is initialized during the training phase and then contains two sets of vector clusters, which are those that are „normal“ and those that are „abnormal“. When the training phase is complete, they calculate the distance of a new input vector to the representative vector from the knowledge base. If this distance is above a threshold, the vector is treated as an anomaly. Otherwise, the vector is reported as normal or abnormal depending on which type of cluster is closer [46, 59].

Principal Component Analysis (PCA) is a statistical method that projects high-dimensional data into a lower dimensional space. This is done in a way that preserves the major characteristics of the input data by looking at the variance of each component. PCA uses the components that have the highest variance since these can best be used to describe the data. The space constructed using these high-variance components is called the „normal space“, the space constructed by the remaining components is called the „abnormal space“. A data point that is not highly correlated with the normal space is considered to be an anomaly [46, 110].

The third unsupervised method, invariants mining, works by determining a set of invariants for an input. An invariant is a statement that holds true regardless of the state of the application. In this case, the algorithm searches for invariants in the log messages. For example, such an invariant could be that the number of opened and closed files must always be equal. The application logs each open and close operation and the algorithm automatically detects this invariant from the log file. If the invariant does not hold for a window of the log content, this is reported as an anomaly [46].

For the evaluation, He et al.[46] use two public data sets. They show that supervised methods generally achieve high accuracy on both data sets, while the accuracy of unsupervised methods varies greatly depending on which method and which data set are used. Supervised approaches achieve high precision, which means that most of the anomalies they report are real anomalies. They can also perform better with sliding windows instead of fixed windows, but for some data sets, they only have a recall of around 57%, which means that they detect only 57% of all anomalies in the dataset. From the unsupervised methods, invariants mining provides relatively consistent accuracy. However, while most other methods scale linearly in terms of log size, the running time of invariants mining highly depends on the number of different event types. Its running time is two to three orders of magnitude worse than most other approaches from this study [46].

4.4.3 Network Based Anomaly Detection

Another way to detect abnormal activity is to analyse the network traffic of systems and look for anomalies there. Oprea et al.[75] note that advanced malware may be difficult to detect and remove in sufficiently large enterprise networks. There are many variants of malware that manage to evade existing security measures during the early stage of a multi-stage infection. However, early discovery is important to limit the potential impact of an infection. Therefore, the authors propose a system that monitors which systems talk with each other over the network by analysing for example DNS server or HTTP proxy logs [75]. While this is also a form of log file analysis, similar to the approaches discussed in Section 4.4.2, it is more focused on the network communication and less focused on providing general purpose anomaly detection for arbitrary log files.

Machines that are being attacked by malware, often visit multiple domains that are under the attacker's control. Stringhini, Kruegel, and Vigna[99] explain that after clicking on a malicious link, victims are often redirected across multiple sites and domains. This redirection chain exists because it allows attackers to change individual points inside the chain relatively easily, without having to update all the entry points that channel victims to the chain. This allows the attacker to hide their various sites and also makes it more difficult for anyone to shut them down or take down their operation. It also allows them to perform various checks during the redirections to further hide their important delivery sites. For example, attackers use a technique called „cloaking", which means that they only show themselves when they believe that their victim can actually be attacked. Users use a variety of different software and different software versions. Some versions may contain an issue that can be exploited to attack the machine that the software is running on, but other versions may either not yet contain this problem or it may be fixed. If attackers try to attack everyone, it is comparatively simple for security experts to obtain a sample of the attempted attack and ensure that all systems of an organisation are protected. To avoid this, attackers check for the presence and version of browsers, installed plugins, or operating systems. Only when vulnerable software is detected, the potential victim is sent onwards in the redirection chain until they are eventually sent to the malware that attacks their machine. Otherwise, they may be sent to a different chain or not be attacked at all [99].

The redirection chains lead to victims resolving many domains. In an enterprise network these attacker-controlled domains are generally only visited by few machines and thus looking for

anomalies in the resolved domains allows detecting such attacks. DNS server or HTTP proxy logs allow recording the domains that each machine in the network visits in a central place. However, not all domains that are rarely visited are malicious. To detect anomalies, Oprea et al.[75] created a belief propagation framework that models the relationships between internal machines and external domains. If a machine contacts a domain at least once during a set time frame, which in their case is one day, an edge between the machine and the domain is added to the graph. To prevent the graph from exploding in size, they generate a list of domains that are contacted often and by many hosts. Before adding an edge to the graph, they first consult the list and verify that the domain is indeed either a new domain or a rarely accessed one. To actually detect anomalies, they then apply belief propagation to the graph. Belief propagation is a graph inference method that allows labelling nodes in a graph based on prior knowledge about the node itself and its neighbours. This allows the authors to detect communities of malicious domains which are likely part of the same malware campaign [75].

Oprea et al.[75] calculate a score for each domain in their graph by using linear regression to calculate the probability of a domain being used for a command and control server, and the similarity to known suspicious domains. The probability of a domain being used as a command and control server is determined by checking how many hosts contact the domain without sending a web referrer, which user-agent strings are sent by the browsers, and how recently the domain has been registered. A web referrer is an HTTP header field that indicates from which site a user came to the URL in question. For example, if a user is on a search engine web site and clicks on a result link, the site will receive a referrer header that tells it that the user came from the search engine. However, the referrer is not sent in all cases. Nielsen et al.[71] argue that it should not be sent when a request is sent over a non-secure connection and the referring page „was transferred with a secure protocol.“ Additionally, sending such headers may be disabled in certain browsers, but in the case of malware, a connection may be initiated directly by the malware without browser interaction. In this scenario it is unlikely that a referrer is set at all, which means that the number of such requests may be an indication of a malicious domain. Similarly, malware may send an arbitrary user-agent string, which is another HTTP header and which is generally used to indicate the used browser version. Enterprises often use similar browsers, which means that they will send similar user-agent strings. Malware may send a different one or none at all so these cases can again be used as indicators. Finally, attackers often register domains for short periods of time to minimize risk of detection and blocking. Therefore, if a domain has only been registered recently, this can hint at an attacker domain [75].

The similarity to other domains is determined by the number of hosts that contact the domain, the time difference between a host contacting each two domains, and the distance between two domains in IP space. Two similarly suspicious domains are contacted by very few hosts, by each host in quick succession, which indicates a redirection chain, and are hosted on IPs that are in the same, small subnet. The importance and weight of each of these factors is determined during a training stage by using linear regression. Rare domains are sent to the VirusTotal service to determine if they are known to be malicious. VirtusTotal's results are used to provide a base line from which the algorithm can use belief propagation to discover related domains [75].

4.5 Storage of Audit Data

In order for audit logs to show accountability or prove regulation compliance, it is important that they are stored properly. While regular log files also often require secure storage in some respects, as discussed in Section 3.4.4, audit log storage has additional requirements, especially concerning integrity and authenticity. While it may be acceptable for a regular log file to be protected against specific attackers with varying privileges, an audit log may be required to be stored in such a

way that even an administrator in the organisation itself cannot alter it. Such requirements can be addressed by using tamper-evident storage or immutable databases. However, audit logs may also require increased availability and dependability compared to other logs. For example, audit logs are often required to contain all events, which places restrictions on what applications can do if the audit log is unavailable [22, 53].

4.5.1 Tamper-Evident Storage

Since audit logs often need to be verifiable by third parties, these external auditors have to be able to ascertain if an entry in an audit log has been modified, replaced or removed. A tamper-evident audit log is one where it is apparent that its content has been changed and thus that the log has lost its integrity. Waters et al.[108] differentiate between tamper resistance and verifiability of a log file. Tamper resistance concerns itself with the defense against an external attacker, while verifiability is concerned with enabling auditors to verify that the log has indeed not been tampered with. Regarding tamper resistance, they explain that while it is impossible to prevent an attacker from modifying future log entries when they gain sufficient privileges, the goal of tamper resistance is to prevent them from modifying existing log data. Thus, any traces that the attacker leaves in the log file, before they are able to tamper with it, are still recorded and if their integrity is ensured, they can still be used for tracing the attacker's activity.

Depending on who is able to verify a log, it can either be „publicly verifiable“ or it may require a „trusted verifier“, which is an agent that is in possession of a secret verification key. Verifiability of logs is generally achieved by methods that ensure integrity, such as MACs which are discussed in Section 2.1.5. However, logs are generally not static, but rather extended with new entries all the time. The goal of verifiability is that the complete log can be verified and missing entries can be detected, which means that simply authenticating each line on its own, while required, is not sufficient. A solution towards this problem is to link entries, or blocks of entires, in a chain by using cryptographic hashes such that each entry contains the hash of the previous one [108]. An implementation of this idea is described by Haeberlen, Kouznetsov, and Druschel[43]. The chain begins with a well-known value because there is no previous entry to calculate a hash from for the first entry. Next, the log contains an entry followed by the hash value of both, this entry and the hash value of the previous entry, which, in case of the second entry, is the well-known start value. Verification of such a log is as simple as repeating the operations, using the log entries from the log file, and ensuring that the calculated hash matches the hash recorded in the log file. If a mismatch is detected, the log has been tampered with, however, all entries before the mismatch can still be trusted [43].

Public verifiability of such a hash chain can be achieved relatively easily by publishing the hash of the most recent entry via a trusted third party, such as a newspaper [108]. Without such regular checkpointing, an attacker could still delete log data from the end of the log file and recreate it by forging entries. Sutton and Samavi[101] implemented a system which provides similar public verifiability by using the Bitcoin blockchain as a trusted third party.

Blockchain technology works by storing data in a continuous list of blocks that each contain a hash of the previous block, or more specifically a hash of the header of the previous block as shown in Figure 4.6. A blockchain itself is also a form of tamper-evident and tamper-resistant storage and it works similar to the hash chain, however it has a slightly extended block format, which includes the hash, a timestamp, a nonce, and a hash of the block's own content, which is a list of transactions and optionally arbitrary data. The nonce is a field that does not exist in all implementations, but when it exists, it is often used by the consensus model of the particular blockchain [112].

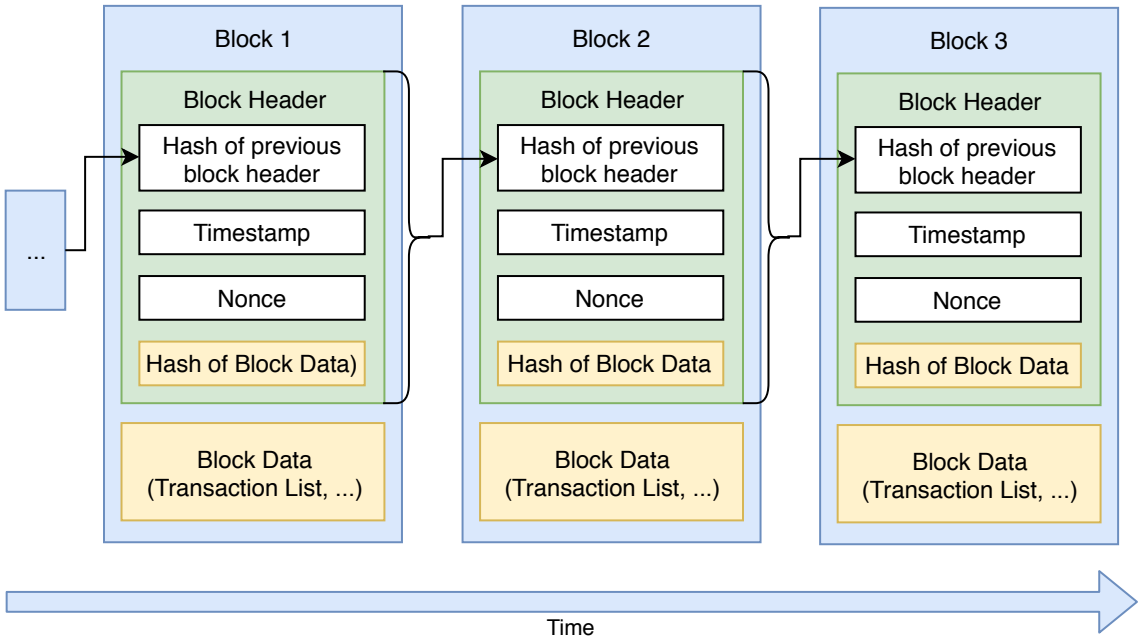

Figure 4.6: Diagram of a generic blockchain showing the structure of the chain. Based on diagram by Yaga et al.[112]

The Bitcoin blockchain uses a Proof of Work (PoW) consensus model, which means that a node must calculate a hash value for its block header that is below a specific target value. Since the previous block value, timestamp and the hash of the block data are (relatively) constant, the only way to change the block is by changing the value of the nonce field until the hash satisfies the requirements. The difficulty is that finding a correct nonce value takes many tries since the result of the hash function is unpredictable and, therefore, nodes have to invest time and resources to find a correct value. Once a node manages to find such a value, it can publish the complete block and other nodes can quickly check if the value is indeed correct and accept the block into the chain [112].

To prevent tampering, the chain is stored on many systems that then form a network and only blocks that have been accepted by a sufficient number of nodes are a part of the official chain. If an attacker tries to change a previously recorded block that block will then have a different hash. By induction, the hash of the next block will also change and so on. Therefore, if all hashes in a chain are verified and the other nodes in the network agree on the most recent hash, the entire chain can be trusted [101].

Sutton and Samavi[101] use the Bitcoin blockchain to build a tamper-evident, chained data store for audit logs. While they store their actual log data locally, they generate a signature of their data and store that signature in the Bitcoin chain. Once the network accepts their signature, it is stored there forever. The signature can then be used by an external auditor to verify that the audit logs have not been tampered with. However, while Sutton and Samavi[101] claim that the Bitcoin blockchain is an immutable database, this is not strictly true and the nuances of immutable databases are discussed in Section 4.5.2.

4.5.2 Immutable Databases

An interesting property in the case of audit logs is immutability, which means that once data has been created, it cannot be modified or deleted. This is helpful for logs because if the log is immutable, it protects the audit data from tampering and thus provides stronger assurances than

tamper-evident storage, which is discussed in Section 4.5.1. An immutable database is defined by Duncan and Whittington[28] as a database that only allows addition of new entries and prevents deletion or editing of existing ones. This means that, by design, such a database protects the integrity of its content.

In their example, Duncan and Whittington[28] decide to repurpose a MySQL database and use it to implement such an immutable database. They provide a list of three possible solutions, with the first one being the restriction of the privileges of users in the database such that they cannot issue modification or deletion queries. Their second option is removing the modification and deletion commands from the software code and their third solution is to use an „archive“ database. While the first and third options do not require software changes, they claim that an attacker could easily change the permissions in the first case if they manage to get sufficient privileges in the database. For the second case they believe that an attacker is unable to reverse the modifications, however the maintenance overhead is high since the changes have to be repeated for new software updates. Regarding the third option, which they have chosen as their solution, they do not provide a lot of details except that it does not require software modification and does not support „key searching“. It appears that they are talking about the MySQL „ARCHIVE“ storage engine, which is described in the MySQL manual by Oracle[76].

The MySQL Archive storage engine provides support for database tables that store large amounts of data with a small overhead. This means that it also misses a lot of database features supported by other engines, such as search indexes, and it also does not support changing or deleting data from the database [76]. However, while Duncan and Whittington[28] claim that this makes it an immutable database, this is only true under certain circumstances, which are only hinted at in the paper. The authors note that their approach requires that the immutable database runs on a well-protected, dedicated system. They especially mention that this database must not run on the same system as, for example, a web application that logs its log data to the immutable database [28]. What they do not mention is that an administrator can change the configuration of the „immutable“ database. They can also dump the content of the database, change it, and load it into a new ARCHIVE database. While this does make it more difficult for an attacker to tamper with the database, this solution is not much different from a database with another engine and restricted permissions. If an attacker manages to gain sufficient privileges to change user permissions, they can likely also dump the database content and create a new database with their modified data. This is especially true, if the attacker manages to gain administrator access to the machine that the database is running on.

A similar issue is present in blockchain ledgers such as Bitcoin. As explained in Section 4.5.1, blockchain technology works by maintaining the same chain on many machines and this network needs to agree on what is part of the chain and what is not. Yaga et al.[112] explain that, in some blockchain implementation, nodes consider the longest chain to be the legitimate one. This permits an attack which is known as the „51% attack“ and which describes the situation where an attacker controls more than half of the computing power of the network. This situation is problematic because the attacker can then create new blocks in the chain faster than the rest of the network. This allows them to create new blocks faster than the remaining network and thus force their acceptance. By forcing the network to accept their blocks, they can also modify the chain's history [112].

To modify a blockchain, the attacker starts at the transaction that they want to modify or remove and then works their way forward to the current state of the chain. When they reach this state, the remaining network may have already added new blocks to the chain and the attacker, therefore, also has to add blocks to their alternate chain. Once both chains have the same number of blocks, the attacker only needs to add one additional block for the network to accept the alternate chain. After adding this one block, the alternate chain is longer than the original chain. The process that

the attacker has to perform is not particularly difficult since it is simply the regular process of the respective blockchain implementation, with certain transactions either included or excluded. The important difference is that it needs to be performed at a faster rate than the regular operations of the network so that the attacker can eventually catch up with the length of the original chain. While this type of attack becomes increasingly more difficult the further back the attacker wants to modify the chain, it is always possible and the only question is how much money the attacker can afford to spend on the operation. If the chain is modified further back, it will take more computing power to catch up within a reasonable time frame [112].

Both solutions, the MySQL ARCHIVE engine and the Bitcoin blockchain, are vulnerable to changes or attacks by intentionally privileged agents. In case of the MySQL solution, these agents are administrators that have the power to change the database, while in case of the Bitcoin blockchain, they are the developers of the software and of the cryptographic algorithms. If the developers add a loophole or an unintentional bug to their software, this can allow them or others to change the chain and harm it's integrity. Similarly, the Bitcoin blockchain relies heavily on cryptography to ensure fairness and if the cryptographic functions are broken, it may be possible to attack the chain. Additionally, the trust in other nodes can be an issue, especially for users that do not run a full node. A full node is one that has the entire blockchain and thus the history of the chain. However, the entire chain may be quite large in size and obtaining it may cost considerable resources, which is the reason why there are lightweight nodes, which contact a full node when they need to process transactions. This places trust on the full node that it accepts and correctly processes these transactions fairly [112].

4.5.3 Secondary and Off-Site Storage

Different from some other types of log data, such as debug logs, audit logs must be kept securely for extended periods of time. Additionally, their content may include many or all actions happening on a system. Both of these aspects mean that they have unique storage requirements, such as a focus on availability and integrity, especially when audit logs are archived. It is also necessary to ensure that recently created audit data, which has not yet been archived, is subject to the same security requirements. This often results in audit data being transfered to specialised, secondary systems quickly after its creation so that even if the original is compromised, the audit data stays secure [22].

Depending on the availability requirements, it may be necessary that audit data can be accessed immediately, which requires the use of „online“ storage. This is typically the most expensive option since dedicated hardware must be maintained and powered on to serve requests when they come in. Examples for this include servers with local disks, database systems, or Storage Area Network (SAN) systems [22].

If availability does not have to be instantaneous, which may be the case for archived and thus older data, „nearline“ and „offline“ storage systems can be used. An offline storage system is one in which the data is stored on drives or disks that are disconnected from the network and where human interaction is required to bring the data back online. These systems are highly-scalable in terms of storage space and generally the cheapest option, but their access time is usually high since disks have to be fetched manually [22].

Nearline storage systems are an in-between solution, which combines some of the benefits of online and offline storage. They typically consist of many drives or disks which contain the data and which are connected to the network by a robot when their specific data is required. This can be implemented as an optical storage jukebox or a robotic tape system and when archived data is requested, the system automatically connects the respective drive. This obviously takes some time

and typically the access time for such systems is in the order of seconds to a few minutes. Costs can vary greatly, but scalability is generally better than with online systems [22].

A common problem of all of these solutions that is especially important when archiving data, is media degradation, which is discussed in Section 3.2.3. In online storage systems these problems are less disastrous since these systems often use RAID storage solutions, which are described in Section 3.4.4. In nearline and offline storage systems, media may not be checked as often since access is more difficult. It is thus important to chose media that have a sufficient expected life time and to rerecord data to new media when that life time is reached [22].

Each of these types of storage solution can be located either at the same or at a different site than the primary storage system. If they are located at the same site, the second system is said to be on-site, while it is off-site if it is in a different data centre, building or city. Since a single copy's integrity can be harmed by various events, such as accidents, disasters or attacks, it is advisable to maintain multiple copies of important data for increased availability. This is similar to the storage of other log data, which is described in Section 3.2.3, however, audit data collection and availability often has specific requirements by regulations which increases the importance of proper data protection. Some regulations may even dictate which type of storage solution must be used. For example, the PCI-DSS requires organisations to „retain audit trail history for at least 1 year, with a minimum of 3 months online availability“ [22].

On-site storage has the benefit that is easy to access and it shares protection mechanisms with other infrastructure of the organisation, while using off-site storage may transfer some of the responsibilities and control to a third party. Off-site storage can be fully managed by the organisation itself, but other options include storage service providers, tape or disk storage facilities, and cloud providers. Many cloud providers offer various types of services, ranging from simple storage to complete log data management and auditing. While cloud solutions often are fully maintained by the provider and resources are shared with other customers, which is called „public cloud“, there are also other types, which are „private clouds“ and „hybrid clouds“. Both of these types can either be provided by on-site or off-site infrastructure. In a private cloud, the resources are used only by a single customer, while the management of the cloud can be performed by either the cloud provider or the organisation. A hybrid cloud is a combination of public and private clouds and allows moving data and services between the public and private clouds as necessary [22].

Services offered by cloud providers include storage, infrastructure, platforms, and software. Storage as a Service, also called „cloud storage“, means that the provider offers only storage space and customers rent the space they need and often pay for the network traffic that they generate. Infrastructure as a Service (IaaS) means that the provider supplies computing resources such as CPU, memory, disk and network, and the customer can use these to run their own operating system and applications. Platform as a Service (PaaS) offers are those where the customer can use a provider's platform and does not interact directly with the operating system or the hardware. The customer only supplies their own application code which is then executed by the platform. Finally, Software as a Service (SaaS) describes offers which the customer simply uses. All software and hardware maintenance is performed by the provider [22]. Ray et al.[85] explain that when using SaaS cloud logging solutions, the customer sends log entries to the cloud provider's service and the provider handles storage and analysis.

An important issue when dealing with cloud providers is that, depending on the used service, the provider often has direct access to the data they are processing. For example, in a cloud logging solution, the customer has to trust the provider that the data of each customer is properly protected against attackers, other customers and employees of the cloud provider [22, 85]. While encryption may quickly come to mind when thinking about data confidentiality protection, encryption generally defeats analysis, however, data can be pseudonymized as described in Section 3.4.2. For

increased protection, this pseudonymization can be performed on-site, before data is sent to the cloud provider. It is also important to protect the transfer of data from the customer to the cloud provider against various IT security threats. As described in Section 3.2.2, a good solution is using secure network protocols such as those protected with TLS.

Availability is also a special requirement when dealing with audit logs, since audit logs are often expected to contain all important events. This raises questions when dealing with setups that rely on off-site storage systems since these systems may become inaccessible, which means that the application may be unable to log its audit data to them. Kent and Souppaya[53] note that stopping the log data generation, but continuing operation of the service is generally unacceptable because this prevents monitoring of security related events. Overwriting the oldest log entries may be an acceptable solution in cases where the log data is of low importance and the only issue is a lack of storage space. The best option for critical logs, which audit logs arguably are, is to stop the log generator and the service it provides [53]. For example, an application may be configured to shut itself down if it is unable to write its audit log data to a log file. Errors that trigger this fail-safe may be as simple as full disks and thus known triggers, such as the free disk space, should be monitored closely by administrators. Similarly, if the log is sent to an off-site destination, administrators may wish to monitor the network connection to the off-site recipient and free disk space on the destination system. Many logs grow relatively slowly and continuously, which means that monitoring the situation often gives administrators sufficient time to archive any required log entries, remove those that are no longer needed, or extend the storage space.

4.6 Accessing Audit Logs

Being able to quickly and securely access audit log data is very important and arguably more difficult than it is for other types of log data. Audit log data often contains personal information and protecting the confidentiality of this data is highly important, but audit log data must also often be kept for extended periods of time due to regulations. Different from some other types of log data, recording audit log data, and consequently properly protecting it, is not optional, but also required by certain regulations. Ensuring confidentiality can be achieved by properly securing the data storage and correctly dealing with storage access permissions or by employing advanced encryption schemes. Regulations also require that audit logs are regularly reviewed, with PCI-DSS stating that logs for system components must be reviewed at least daily [22]. Such reviews can be supported by search tools like 'grep' or Kibana.

4.6.1 Data Storage Permissions

Section 3.4.1 explains that even if data is encrypted, it is important to still protect it and prevent access by unauthorized parties. A common solution to this problem is using file system permissions to restrict which users are allowed read or write access to a file or directory. However, there are various possibilities for problems in such a setup, especially if users and thus potential attackers have direct access to the machine storing the log data.

Log generation or aggregation software is often configured to store log data locally and additionally forward it to a central storage system. Forwarding data is important for analysis, as explained in Section 3.2.2, but local storage is also important. Local storage ensures that log data is not lost if it cannot be forwarded for any reason, such as networking or hardware problems on other machines. However, employing local storage also generates some problems. While it may be difficult for an attacker to target the central log file, they may have more luck with any local files on the log generation machine. The file or directory permissions may not be set to strict enough values, the attacker may be able to read the content of the hard disk, or permissions may be changed during operation of the machine [22].

A log file may be world-readable, which means that any user on the system can read it. Alternatively, read access may be restricted to a group, or it may be restricted only to the file owner. Chuvakin and Schmidt[22] note that a world-readable log file may be created intentionally so that users can diagnose problems they face, such as during login. However, if a user mistakenly enters their password as part or instead of the username, this may show up in the log file and thus allow an attacker to gain administrative access.

Even if a file is not marked as world-readable, it may still be readable to a group of users. This means that any user that is part of this group is allowed access, but such restrictions also need to be used with care since the members of a group and the files or directories that are readable by the group may change over time. Certain services, such as analysis tools or log forwarding software, may require read permissions on the log files and their users may be placed in a dedicated group to provide them with the necessary access [22]. However, simply changing the permissions of the log files once when introducing such a group is not sufficient. Many systems use software that rotates log files, so that old files will eventually be deleted automatically. This rotation software may rename the original log file and then recreate it with a set of permissions from its own configuration. It is important to ensure that any such software also enforces the same permissions and most importantly that it does not configure more permissive ones. Such an issue may arise, for example, if the permissions are changed to be more strict by removing access for a group. The log rotation software may only rotate a file once it satisfies certain criteria and thus any changes it performs may be delayed. If it then configures more permissive permissions than intended, it may be unlikely that this problem is noticed since all other services continue to work. If permissions are changed to be more strict, services will likely report an error and thus administrators are more likely to notice the problem.

Similarly, when creating new (log) files, it is advisable to configure strict file system permissions and only relax them when necessary. On a Linux system, Bauer[9] shows how this can be achieved by using the „umask“ command, which configures a permission mask that is applied when new files are created. File system permissions can be represented as bit masks for the values read, write and execute for each type of four access types. The types are „special“, „user“, „group“ or „world“ access. A short representation uses a number between 0 and 7 for each type, with 0 meaning that no permissions are granted, and each other value being a sum of the numbers for read (1), write (2), and execution permissions (4). This representation is also called a numeric „mode“ [9].

When a new file is created, the operating system defaults to a mode of 0777, which means that no special permissions, but all others, are granted. With a umask, this mode can be restricted, however, the umask value works, differently from the regular permission value, by being subtracted from the permission value. A umask setting of 0022 tells the operating system that new files should be created without permissions 0022, which results in $0777 - 0022 = 0755$, which means that the owner has all permissions, while group and world permissions are set to only include read and execute bits [9]. A more strict umask, which defaults to prohibiting access for group and world, is the value 0077.

Finally, permissions are only enforced in the running system. This means that if an attacker gains access to the raw disk data, they can circumvent the permissions set on the files. Similarly, Section 3.2.3 describes a situation where deleted data is recovered from a disk image.

4.6.2 Fine-Grained Record Encryption

Goyal et al.[39] note that a common problem when using encryption to support the confidentiality of log files is that encryption often makes it difficult to provide read access to only parts of a file instead of the complete file. This presents a challenge when, for example, an auditor shall be granted access to log data from a particular time range and with a particular IP address being

involved. Even if files are split per day, each file is often encrypted with only one key and sharing that key with an auditor risks compromising the confidentiality of all the other log entries that are not required to be seen by the auditor. If the organisation performs the decryption and filtering for the auditor, the auditor cannot tell if they filter the entries correctly or not and thus this solution is arguably not satisfactory [39].

A solution to this problem is to encrypt each log entry differently, such that it is possible to provide an auditor with a key or set of keys that decrypt the entries they are interested in and nothing else. However, the auditor must be able to trust that the keys they get cover all log entries they wish to analyse. Cryptosystems that can provide this type of guarantee include Key-Policy Attribute-Based Encryption (KP-ABE) by Goyal et al.[39] and Ciphertext-Policy Attribute-Based Encryption (CP-ABE) by Bethencourt, Sahai, and Waters[15]. In Attribute-Based Encryption (ABE) schemes, decryption access is provided based on a set of attributes either of the message or the user. Wang et al.[107] explain that, depending on the scheme, the policy that determines which messages a user can decrypt, can be configured when creating the key or when creating the message. In both solutions, KP-ABE and CP-ABE, keys are managed by a trusted key authority and distributed to users, which are either publishers of encrypted data or subscribes that wish to decrypt data [107].

In KP-ABE, the attributes are embedded in the cipher text and the policy is embedded in the private key of the subscriber. Decryption requires that the access policy in the key matches the attributes embedded in the encrypted data [107]. This allows the key authority to issue a key that can decrypt events that match a specific set of attributes, such as a year and IP address, however this also places trust in the key authority to correctly determine which attributes a subscribers should be allowed to decrypt. The publisher has no control over the subscriber other than by choosing descriptive attributes [15, 39].

In CP-ABE, the access policy is part of the encrypted message and the attributes are embedded in the key of the subscriber. Decryption requires that the access policy of the cipher text matches the attributes of the subscriber. This means that the publisher can determine which attributes the subscriber must satisfy in order to be able to read the message. The key authority only asserts which attributes describe the subscriber, but they are not able to directly provide access to certain messages. The control over the potential readers is thus placed on the publisher, which is the entity that encrypts the message [15].

An important property of both encryption schemes is that they are collusion-resistant. This means that two users are only able to decrypt the messages that each of them can decrypt on their own. They are not able to combine their keys such that they can read messages that neither of them can decrypt alone [15, 39].

A similar, although not equivalent, solution is the approach by Biskup and Flegel[16] based on Shamir's scheme, which is described in Section 3.4.2. The difference between that solution and ABE is that the former only allows decryption if a user triggered a sufficient number of different log event types, while ABE allows decryption of all events if a user is supplied with the matching key.

4.6.3 Search in Audit Logs

While the „grep“ command discussed in Section 4.3.2 works well for infrequent searches or small log files, it becomes slow when the log data size increases. It becomes especially cumbersome to use when multiple consecutive searches are performed since each search has to read the whole log file. Faster searching can be achieved by preparing the data and storing it in a search friendly format. An example for this is a database with an index, which can be searched for specific values much more quickly by the database engine than grep can search in a text-based file.

A suite of tools aimed specifically at searching in log files is the Elastic Stack. Berman[12] explains that it was formerly also called the ELK stack after its three main parts, which are Elasticsearch, Logstash, and Kibana. Since then, an additional important group of pieces called Beats became a part of the common solution and thus the name changed from ELK Stack to Elastic Stack. Together, these components allow collecting log data, extracting values from it, storing these in a database, and searching and analysing the stored data [12]. This stack combines various responsibilities and methods which have been discussed in Sections 3.2, 3.4, 4.3, and 4.4.

Following the flow of the log data, the first part of the stack are the Beats. Beats are a collection of various agents that collect log data or metrics. They are small and lightweight compared to other parts of the stack and run on the machines that should be monitored. Filebeat allows collecting file based log data, while Packetbeat allows collecting network data, and Metricbeat collects various server metrics. Data collected by a Beat can be sent to Logstash for further processing, or directly to Elasticsearch [12].

In many cases, log data is preprocessed by Logstash before being stored in the database. Logstash collects, parses, normalizes, and augments the log data so that it can be searched more efficiently than by performing full text search. Logstash receives arbitrary log data and, by using various filters, transforms this data into well-structured data. If log data is in a JavaScript Object Notation (JSON) format, it is often already structured and this structure can be reused. Other log data may contain key-value like data, which is data with a well-defined schema where each value is prefixed with a key that describes the value. For example „ip=1.2.3.4 request=login user=bob“ contains three keys and the respective values. This format can easily be transformed by extracting these relationships. Other formats are the readable messages discussed in Section 4.4.1 which do not follow a well-defined schema. Berman[11] explains that for such log formats, Logstash can extract key-value pairs by using regular expressions that define which parts of a message should be treated as belonging to which key [11]. While Logstash already contains some patterns, Rohmann[88] note that there is a wide selection of community created ones. However, these patterns suffer the same issues that are discussed in Section 4.4.1, namely that they only work for messages that follow the pattern. If log formats are changed by an administrator or by a software author, the patterns have to be adjusted for continued log analysis [88].

Elasticsearch provides a NoSQL database based on Apache Lucene that offers a wide range of querying and search capabilities. It stores data in an unstructured way by storing „documents“, which are JSON objects that contain the data. Data inside a document is stored within fields that each have a key and a value. When compared to a relational database, a document is similar to a row in a relational table. In the case of an Elasticsearch database for log data, a document may be a single line from the log file, which has been preprocessed by Logstash to be represented in a key-value JSON object. The documentation by Elasticsearch BV[29] explains that documents are stored in indexes, which group similar documents together and make them accessible for search, update, and delete operations [29]. Users can then search for documents in the index using simple string matches, range queries, regular expressions or fuzzy search queries [12].

Kibana is a web-based front end that allows users to run such search queries against the Elasticsearch server by using a graphical interface. It allows users to perform various types of search queries, including free text searches, field level searches, and logical statements that combine multiple search queries. The search results can then be visualized as graphs or charts, or multiple results can be grouped by using aggregations such as summation, average, and min/max operations. Visualisations include bar, line and pie charts, gauges and metrics which show only a single value, heat maps and geographic maps, tables and tag clouds. For continued use and to gain a better overview over the data, multiple visualisations can be displayed at once on a „dashboard“. Dashboards can be saved, changed, shared, and the content of each visualisation can be filtered for all visualisations that are part of the dashboard [12].

While a solution like the Elastic Stack offers much flexibility, all the data inside the database is stored in plain text. The only data protection available is encryption of the file system that the data is stored on[1], however Waters et al.[108] note that it is important to properly protect audit data, since this type of data may contain sensitive information. They further explain that it is important that the search capabilities include support for secure delegation. Secure delegation means that an agent can be provided with access to specific data items and they can only search within these items and cannot obtain information about other items. This can be solved by having a trusted key holder that performs the search, such as is the case with an Elasticsearch server that performs the search for a client. However, Waters et al.[108] argues that such a solution is undesirable since it exposes a highly trusted and potentially complex part of the system to attackers. Different encryption approaches are discussed in Section 4.6.2.

[1] https://discuss.elastic.co/t/encryption-at-rest-support/113537 (visited on 2019-05-02)

5 Case Example

The following chapters use the foundations discussed in this thesis to create a concept for implementing an activity logging and auditing solution for a security exercise environment at a university. The solution allows the course administration to monitor student activities inside the exercise environment and respond to abuse of their service. Such capabilities are important because students are taught various techniques to attack networked and local services and the exercise environment purposefully contains vulnerable services that students can experiment with. This chapter describes the example exercise environment and the requirements that a solution should satisfy. The solution concept is described in Chapter 6.

5.1 Description of the Example Exercise Environment

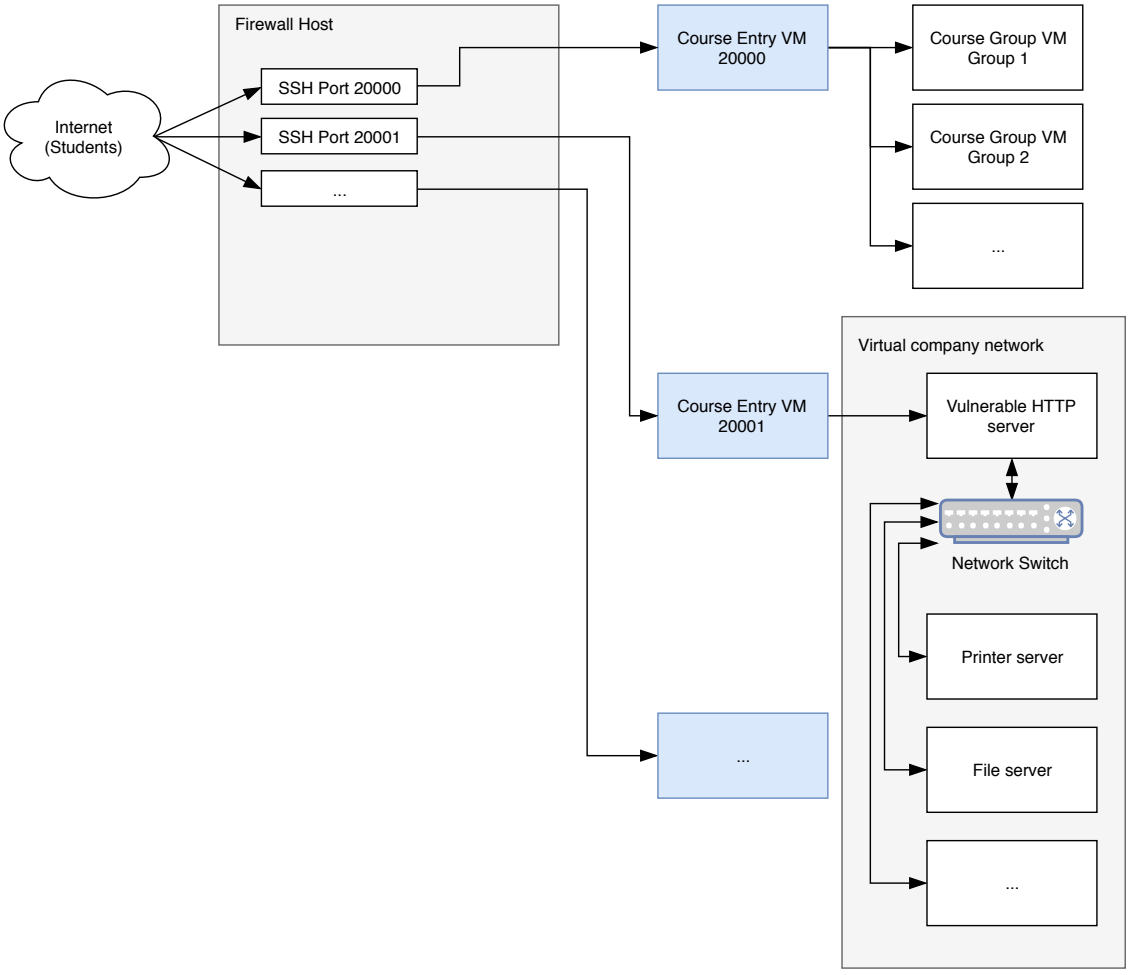

Figure 5.1: Diagram showing the network setup of the exercise environment

Figure 5.1 shows the network setup of the exercise environment. The environment is connected to the internet at large so that students can work on their assignments from anywhere. To prevent abuse, the environment is separated from the internet by a machine called „firewall host“ in the

figure, which works as a router and firewall for the exercise network. For each course, there is a dedicated „course entry VM“ which students that take the course, can connect to. These machines, marked in blue in Figure 5.1, can be reached from the internet via forwarded SSH ports. Once connected and logged in, students have access to an interactive shell, such as bash, running on a Linux system to work on their assignments.

What students are allowed to do by using this shell depends on the exercise. In one exercise students might be supposed to maintain their own virtual server, find and correct security issues on it. This simulates a real world situation where they take over administration of a server used by a company and have to securely maintain it. For such an exercise, each team gets access to their own server, which they can reach by executing SSH on the course entry VM. These servers are called „course group VM“ in Figure 5.1. They can then look around on the machine to discover which services it provides and investigate if they are configured correctly or if there are any security issues present, such as software with known vulnerabilities.

In another exercise, students may use a vulnerable service to scan and potentially attack other machines in a network. Figure 5.1 shows a virtual company network with various servers, such as printer and file servers, and an HTTP server with a vulnerable website that is reachable from outside the company network. Students can then use this „vulnerable HTTP server“ to gain access to the network the server is located in. They can then perform various activities such as scanning the network or attacking other services that are not reachable from outside the virtual company network.

Further assignments that do not necessarily require additional machines behind the course entry VM, may include downloading and uploading files to and from the course entry VM. While an obvious solution may be to use file transfer utilities, this particular exercise may use a machine which does not allow such transfers. This forces students to learn how to use pipes and input/output redirection.

Another exercise may be exploiting a vulnerability in an executable program, such as a setuid binary. Using this vulnerability, students can read a secret file on the course entry VM.

5.2 Threat Model

This section discusses a selection of potential security vulnerabilities in the exercise environment network. The potential issues have been discovered by using threat modelling techniques, which are described in Section 2.2. For better readability, each vulnerability is directly accompanied by a short risk assessment and potential management solutions as described in Section 2.3.

Students connect to the exercise environment via an SSH connection that is forwarded to a course entry VM. The SSH protocol, when implemented and used correctly, already addresses a wide range of potential vulnerabilities from the STRIDE list, however, DoS attacks, and their side effects on mitigations for other attacks, potentially in other services, may not be addressed sufficiently. Many SSH servers create log entries when users connect to the server and try to authenticate themselves. An attacker may be able to abuse this to fill up the file system that the log data is written to and thereby prevent additional data from being written. When the SSH server is unable to write log data, it may either stop providing SSH service, or it may continue to provide service without log data. Any further abuse performed by a logged in user may be difficult to trace without log files. It is important to adapt the file system size to the user base and account for some additional data. If necessary, the SSH service can be disabled when the file system is nearly full to prevent possibly untraceable abuse. Depending on the size of the file system, it may be unlikely that this issue occurs in a real world scenario, but the situation must at least be monitored since it also has the potential to affect other applications using the same file system.

Once students or attackers are connected to a course entry VM, they may be able to exploit security vulnerabilities in the operating system and the installed applications. This may allow them to perform a DoS attack or even elevate their privileges. With elevated privileges they can then tamper with almost any data on the system. The risk of such attacks is relatively small since the course entry VMs only run with a limited amount of installed software and student accounts only have limited privileges. Even if attackers are not able to exploit vulnerabilities on a system itself, their actions cannot be traced directly without additional logging. The SSH server only logs when students connect or disconnect, but it does not log which actions they perform inside their session. One log that does record some actions, is the shell's history file. However, this file is writable for the user to which it belongs, which means that it can easily be tampered with, and, additionally, its creation can easily be disabled. Furthermore, it only records the commands that are executed at a shell level, but not what these commands actually do. A user can, for example, start any scripting language interpreter, or even a compiler and then input code via the standard input. They can also start an interactive editor to create a shell script that they later execute and remove. This interactive input is not being recorded and thus the attacker can easily execute commands without them showing up in the history file. To be able to trace such abuse, a log that records all input sent by the student to the course entry VM is necessary. The solution described in Chapter 6, provides such interactive input recording capabilities.

Next, for some exercises students connect to „course group VMs“ on which they are granted administrative privileges. An attacker can again execute arbitrary commands, but they can also interfere with many auditing solutions that may running directly on the machine. While the linux audit framework can be locked, this can still be disabled with a reboot (see Section 4.1). To reduce suspicion, an attacker can make it look like they accidentally deleted the configuration file and then reboot the machine. Afterwards they can use the machine without any audit logging recording their actions. Furthermore, the linux audit framework generates an extensive log file, which contains much information that may be of little use, but which may cause a DoS scenario since there are many course group VMs and the total resources are limited. Especially, the disk space of each individual machine is limited and creating large amounts of data may fill the file system very quickly. Sending all log data over the network to a collection host may also be similarly problematic. Additionally, the solution must already collect audit information for the course entry VM to detect abuse occurring there. Since students use SSH on the course entry VM to connect to their course group VM, this log already contains all the actions performed on the course group VM. This works because SSH is executed directly on the course entry VM and thus the input and output of this SSH command is part of the interactive usage of the shell on the course entry VM. Since logging the input is already required by the threat discussed above, the logging solution also addresses the threat in this paragraph.

However, the assumption that SSH is executed on the course entry VM and that the input is visible in the log can be circumvented by an attacker. SSH can also be used to provide a network tunnel as described in Section 2.5.2. Such a tunnel may be used to forward traffic from the attacker host to the course group VM and thus bypass the interactive input log. The attacker can then use SSH to connect directly to the course group VM, using the encrypted tunnel, and the logging solution on the course entry VM will not log this connection since it only logs interactive input and not tunneled traffic. A solution to this problem is to disable tunneling, but even then, the same idea can be implemented via the remaining channels, which are standard input and output. While it is also possible to implement a custom protocol or a custom version of the SSH protocol that uses standard input and output instead of a TCP connection, this is not necessary. The OpenSSH Authors[74] explain in the „ssh_config“ documentation that the „ProxyCommand“ setting allows an SSH client user to use a command that uses standard input and output to connect to a server. It can be any command or chain of commands that eventually connects the standard input and output channels to an SSH server. An attacker may use a command that connects to the destination

SSH server on the course group VM. Since the SSH protocol protects the connection against an adversary that can intercept the traffic, this means that the data that passes through such a standard input and output based connection is also protected against interception since it uses the same protocol. The only place where the data can be read is on the client and on the destination SSH server, but not in-between. Thus, a log that records all input and output on the course entry VM cannot reveal what the attacker does inside the encrypted connection since this connection only terminates on the attacker-controlled course group VM.

Due to the administrative privileges on the course group VMs, students are also able to perform various network attacks, such as ARP spoofing. Chomsiri[20] explains that ARP spoofing allows attackers to intercept network traffic between two hosts on the same switched network, which presents a risk of information disclosure, spoofing and tampering for any affect network connections. During operation, once a network switch knows which systems are located behind which ports, it only sends traffic to the systems that are the intended recipients. This is accomplished by maintaining a MAC address to port lookup table and it means that an attacker cannot see traffic that is not directed at their machine. A way around this problem, for an attacker, is to send forged ARP packets that look like they came from the gateway router to the host of which they wish to intercept traffic, but rather than sending the MAC address of the router, they send their own. Hosts maintain their own ARP lookup table, which associates IP with MAC addresses, and they will happily update it when a new ARP packet arrives. Therefore, after the attacker sent their forged packet, the host will send traffic for the gateway to the attacker's system which can either forward, modify or simply drop it. This process can be repeated for other machines until all desired traffic is redirected through the attackers system [20].

The risk of network traffic interception attacks being performed on the course group VMs can be addressed in various ways. First, each course group VM can be placed in a dedicated, virtual network with a router that connects the various networks with each other such that the attack is no longer possible. Second, the risk can be addressed by using only protocols that are resistant to these network attacks. Connections from students to each machine can use SSH which addresses the issues by using various techniques described in Section 2.5. Similar protection is offered by TLS protected connections, such as HTTPS [24]. However, in this case, an attacker can still use the redirected connection to perform a DoS attack by, for example, simply dropping all traffic instead of forwarding it. An activity auditing solution, as described in Chapter 6, can help with determining what an attacker did and how they influenced the network if such an incident occurs.

In another exercise, students share the virtual company network with each other. An HTTP server in this virtual network runs software which is intentionally vulnerable to attacks and depending on the type of attack that students perform, this may lead to a DoS scenario. For example, if the software is vulnerable to an injection attack, students may inject attack payloads that lead to extensive CPU or memory usage and which causes the service to crash. If the vulnerability is a buffer overflow, students may accidentally crash the service by provoking a segmentation fault or similar error. While these types of issues may be caused by accident, they can also be abused by an attacker who wishes to intentionally corrupt the exercise environment. The probability of accidents largely depends on the specific type of exercise, but a general solution for DoS issues in this environment is to ensure that services are restarted automatically if they crash and limit their allowed resource usage. Further investigation into such issues can be supported by an interactive input log as described in Chapter 6.

Another type of DoS attack that can occur easily in this exercise is overloading services. This is especially likely when students are expected to perform repeated requests. An example where such behaviour is necessary is an exercise where students have to extract a password from a database by performing a blind SQL injection attack. Nagpal et al.[69] describes a blind injection attack as an attack in which a service is vulnerable to SQL injection, but only returns a boolean answer [69].

For example, a login form that allows SQL injection and replies with a positive or negative login status. By repeatedly asking the database to return yes if the password has a specific character at a given position, it is possible to extract the complete password. These repeated questions may be automated by using simple scripts, but this automation can lead to a high number of concurrent requests if multiple students try to solve the assignment at the same time. This can then obviously overload the service and thus result in a DoS, either by accident or because an attacker purposefully creates a script that simply places load on the service without actually trying to solve the exercise. The problem of this vulnerability is that it is expected of students to issue multiple queries since doing so is necessary to find the solution. To determine if a problem has happened accidentally or on purpose, it is important that student actions can be audited as described in Chapter 6. This includes being able to see which commands students executed, which network connections they created and when each of these events happened.

5.3 Requirements

Audit Log Content The main functional requirement is that the solution must provide the course administration with the capability to investigate abuse of their exercise environment. For this, they require extensive logs that contain at least the following data:

- The date and time when a student connected to and disconnected from the exercise environment, as well as the identity of that student.
- All commands executed by the student on machines in the exercise environment.
- The content of any files the student has created on machines in the exercise environment.
- Any input the student supplied to commands that they ran.

IT Security - Confidentiality, Integrity, Availability, Authenticity A very important requirement for the solution is that it considers IT security attributes, which are described in Section 2.1, throughout the concept. Specifically this means that the solution must provide confidentiality and integrity during storage and transport of the audit data. It must also ensure authenticity of the log data so that later investigations can trust the log data on which they base their conclusions.

IT Security - Dependability The solution must be dependable, which especially includes that it must be reliable and maintainable. This dependability requirement must also consider that the course administration has to manage the solution implementation by themselves. This is an important consideration because their resources, especially personnel time, are limited. Other limited resources include the disk space available to store the audit data as well as the CPU time which also ties in to availability since exceeding the available resources may lead to interruptions in service. The solution should, therefore, minimize the impact it has on these resources by using efficient data storage formats and only performing actions that are strictly necessary. The solution should also ensure that the audit log is complete in the sense that it contains all student sessions. For example, the solution should disallow system usage if the disk is full and audit data cannot be recorded.

Monitoring To support availability during operation, it is also necessary that the solution includes support for monitoring important metrics. These metrics then can be monitored by using

an external monitoring solution, such as Zabbix[1], so that potential problems can be addressed in time.

Data Deletion When the audit data is no longer required, it must be possible to delete it. This includes deleting all audit data of a specific course once that course is over and there are no open abuse cases. Considering the maintainability constraints, this must be possible by performing a single action that deletes all audit data of a course.

Usability A soft requirement is that the audit data must be easily searchable to reduce the time required to investigate an incident. However, since incidents happen rarely, it is important to prioritize other requirements, especially dependability.

[1] https://www.zabbix.com/ (visited on 2019-05-02)

6 Activity Auditing for the Security Exercise Environment

This chapter describes a concept for an activity auditing solution and its implementation. The concept is targeted towards the environment and requirements described in Chapter 5, but with small modifications it can also be used in other environments. The chapter finishes with an evaluation of the performance of the concept and an example scenario walkthrough.

6.1 Concept Description

The structure of the concept is based on the flow of data and thus starts with data acquisition, followed by processing, storage and later analysis. The theoretical foundations of these parts are described, loosely in this order, in Chapters 3 and 4. After the description of the concept, it is cross-checked with the requirements to ensure that they are satisfied. Finally, some limitations of the concept are discussed.

6.1.1 Audit Data Acquisition Method

The first step in an auditing solution is data acquisition. As explained in Section 3.2.1, this step cannot be viewed in isolation, but the remaining system and requirements must also be taken into account. The main requirement for the auditing solution is to allow the course administration to investigate abuse of their exercise services. For this, they require various information outlined in Section 5.3.

Potential solutions include application logging in all or in central services of the exercise environment, command logging of all commands and their arguments, including input data, and system call logging. Network logging, described in Section 3.3.4, is not well suited for many parts of this environment since many connections in the exercise environment use SSH. As explained in Section 2.5, the SSH protocol provides secure, tamper-resistent and confidential network connections and thus the only information that may be gained by employing network logging is metadata about the connections. This metadata includes dates and times of connections, the duration of the connection and the amount of data that was transferred. Similar metadata, excluding the amount of transferred data, is also logged by the SSH server itself in the form of various entries at the beginning and end of each session. However, network logs cannot contain information about the commands that were executed and about the input that these commands received, which means that, at least on its own, network logging cannot record sufficient data.

Command logging, described in Section 3.3.2, can be implemented in various ways. While a shell history file may contain the names of commands and their arguments, it does not generally contain, possibly interactive, input that they receive. It is therefore unsuitable since all command input is required to be recorded. A more thorough version of this is to extend the command log to include the supplied input, which, following the definition in Section 3.3.2, changes the solution to be an application log. In the case of the exercise environment, a central place to capture all user input is the SSH server, since this is the only way for students to connect to the environment. Recording all data received over this connection ensures that commands, their arguments and input are captured. One feature of shells that is also of interest in this case is their history support. Many

shells allow the user to recall previously entered commands or parts of these commands and use them to create a new command. These can often be accessed with special keys, such as arrow keys or combinations of keys. If only the input data stream is recorded, later analysis of the log data has to reconstruct the effect of these history commands to determine which command the shell that received the input has executed. A simple way to avoid this problem is to additionally capture the output data stream that the shell sends back to the user. Essentially, capturing both data stream, input and output, allows the course administration to reconstruct exactly what the original user saw and entered into their session. For easier analysis, the solution also records the time at which certain data has been transferred. This timing information allows for a video-like replay of the SSH session.

An alternative solution is to use system call logging, described in Sections 3.3.3 and 4.1. These solutions record a large amount of data, which arguably contains a large amount of noise, such as accesses to temporary files, configuration files, various repeated IDs. Some solutions, such as the Linux audit framework, also do not record data read from and written to a file. While it is possible to use solutions that record this data, such as „strace“, filtering the recorded data to include only interesting events is challenging and complex. Additionally, the data generated by strace is comparatively more difficult to analyse manually than the previously discussed input/output log.

In the interest of dependability and maintainability, the concept uses an input/output based log that records the SSH activity. Additionally, it also uses the regular system logs, which include the logs of the SSH server, to record metadata about connection attempts, user sessions, and log data from other services running on the system.

6.1.2 Audit Data Processing and Storage

The shell input/output data may contain repeated sequences of text, especially if commands are used that regularly update the user's screen to display progress information. An example for such a command is the „watch“ command which can be used to regularly run another command and display the output, thereby allowing the user to watch and see if the output changes. Such output can quickly lead to a big log file, which means that the log file must be compressed as soon as practical to avoid wasting storage space.

More complex processing, such as automated anomaly detection, is not performed since solutions for this problem introduce an unnecessary amount of additional complexity. Dependability is arguably harmed by more complex solutions. When setting up a complex system, there is a larger margin for error and as Kernighan and Plauger[54] put it „Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?“ [54]. This view is also shared by Schneier[93], who explains that „complexity is the worst enemy of security“. Increased complexity means that issues that arise during usage are more difficult to track down and resolve since understanding a complex system is more difficult. The solutions described in Section 4.4 only work with specific log types and not with the shell activity logs that are primarily collected in this concept. Industry solutions such as the Elastic Stack threaten the dependability and reliability requirements because they are arguably very complex and because they require regular adjustments, for example, when software is updated and log formats change. For details about the Elastic Stack, refer to Section 4.6.3. Additionally, automatic abuse detection is not a requirement and while usability is important, dependability, especially maintainability, is more important, as described in Section 5.3.

Log data is stored on two systems to ensure integrity and availability. The primary storage of all log data is the server on which the data is generated. The secondary storage system maintains a copy of the log data. The primary storage being local ensures that data is stored at least once, even if the secondary storage system is unreachable or data transfer is impaired, for example, due to a

networking issue. In case of the system log, the copy is maintained by software that implements the „syslog“ protocol, which is discussed in Section 3.2.

For the shell activity input/output logs, the syslog protocol is unfit since the logs include raw control characters and more data than is supported by the protocol on a single line. Additionally, breaking up and encoding the data makes it more difficult to analyse later since it may need to be decoded and reassembled. Therefore, these log files are transfered to the secondary storage system as files by using a file transfer program. To simplify transfer and later data deletion, all activity log data is stored in a dedicated directory. Thus, only this directory needs to be copied or deleted.

If storing the shell activity logs is not possible, for example, due to a full file system, the user session is terminated. New user sessions are also terminated if they are unable to write the log data to the respective log file. This prevents any user from using the system if their activity cannot be logged. Consequently, the logs that do exist cover all user activity.

6.1.3 Audit Data Analysis

Analysis of the collected audit data highly depends on the exact circumstances and the reason for analysis. This is because analysis is only performed when necessary and not continuously since proactive analysis is not required by the course administration. The data is used only for incident response rather than proactive monitoring. In most cases, some information about the incident, such as a rough time stamp or an affected machine, will be known. Using this information, an investigator can start by reading the system logs that were written during or before the incident time. This reveals which users were connected to a system and system logs from other services than the SSH server may also indicate what had happened. If necessary, the list of shell activity logs can be filtered to include only these sessions that were active at or before the incident time. If no incident time is known, all logs of a course, which corresponds to a course entry VM, must be reviewed. These logs can then be reviewed either by directly viewing the input and output of the session, or by replaying the session.

Viewing the shell activity logs allows investigators to gain a quick overview over their content. However, interactive programs, such as the „vi“ text editor, regularly update large parts of the user's screen, which results in a lot of clutter in the log file. These logs can be analysed by replaying the session in a video-like fashion. To support the investigator, this replay can be paused at any time and the speed at which events are replayed can also be increased or decreased.

Additionally, the logs can also be analysed with search tools, such as „grep“ as described in Section 4.3.2. Using grep, the analyst can find occurrences of known text or patterns, such as, for example, an IP address that was attacked. The „agrep“ tool is similar to grep and allows investigators to perform approximate searches. It can detect a match even if a configurable number of characters are different from the search string.

6.1.4 Audit Data Security

This section discusses a selection of security measures that are used to protect collected audit data. It does not discuss operational issues, such as how administrative access is handled in the organisation or how physical protection of any equipment is ensured.

To protect the security properties of all log data during storage, suitable access controls are configured. This means that log files and their parent directories are created with strict permission settings that, where possible, only allow administrative access. Students are not able to modify any existing log data or add fake log data. This is achieved by separating the log related processing from the user sessions. For the user activity recording, this means that the recording is created by a privileged process that is able to access the restricted log directory. This privileged process is

located between the user and the user's session and only records the data to the protected log file. The user's session is handled by a process with user privileges which receives and executes the user's commands. Since this process does not have administrative privileges, it cannot access the log files.

The local storage of log data may allow data to be compromised by an attacker that gains administrative privileges. However, the scope of such a compromise is limited by multiple measures. First, the log data that is being recorded only contains interactions with the exercise environment of the specific course. Data from other courses is not affected because these use different course entry VMs. Second, audit data can be removed when it is no longer required, such as soon after the end of the course, when it is clear that no abuse of the service has taken place. This means that each course entry VM only contains log data from the current semester and not from previous semesters. It is also possible to regularly purge log data from the primary system even when this data is still needed. Before purging, it must be ensured that the data has been transferred to the secondary storage system. However, if data is purged like this, the only copy maintained by this concept is on the secondary storage server. It may be necessary to archive data as described in Section 3.2.3. Third, log data is regularly copied to the secondary log storage system which continues to ensure availability and integrity even if the primary system is compromised. While an attacker may be able to modify the log data stored locally, the secondary copy is protected from such modification. This is achieved by implementing a solution that only allows adding new log data, but prohibits accessing existing data. This means that existing files are not overwritten if the source file on the primary system changes. The connection used to transfer log data from the primary to the secondary system must be properly protected, especially against network based attacks. This is achieved by performing mutual authentication of both communication partners. An example solution that offers this guarantee is the SSH protocol, discussed in Section 2.5. Once data is transfered to the secondary system, it is again protected with strict access control so that only administrative access is allowed.

System log data is transfered to the secondary storage system via the syslog protocol, which is discussed in Section 3.2. Similarly to the file transfer, this connection also uses mutual authentication of both endpoints. This means that the primary server is configured to only connect to the intended secondary storage system and that the identity of this system is verified. Since the authentication is mutual, the secondary storage system also verifies that the client that is connecting is an authorized client. This prevents an attacker from sending arbitrary log data to the secondary storage system. Differently from the SSH based file transfer solution, syslog log data forwarding does not use SSH. Instead, it uses TLS, which is also suggested in the protocol specification by Gerhards[36]. The certificates and private keys for the TLS connections are protected with strict file system permissions. Syslog log data is stored in log files on the primary server and secondary storage system. On both machines, it is protected by restrictive access control and only administrative access and access required by the syslog daemon is allowed.

6.1.5 Coverage of Requirements

This section verifies that each requirement is addressed and it explains why each requirement is satisfied. For a description of the requirements themselves, refer to Section 5.3.

Audit Log Content The solution captures the entire input/output transferred via the user session. The input consequently includes all commands executed by the student, the content of created files and any input provided to commands. The SSH server logs the connect and disconnect time stamps and user information in the system log. Refer to Section 6.1.1 for details.

IT Security - Confidentiality, Integrity, Availability, Authenticity The stored log data's confidentiality and integrity are protected with restrictive access controls. Availability of the log data is ensured by storing a copy of all log data on a secondary storage system. During transfer, log data is protected by SSH and TLS connections. These protocols ensure integrity and confidentiality of the data during transfer. Both connection use mutual endpoint authentication which ensures authenticity. Refer to Sections 6.1.1 to 6.1.4 for details.

IT Security - Dependability The solution minimizes the number of different data sources by only using two. The two data sources are the shell activity log, which records all input/output, and the system log. The shell activity log is compressed to save disk space. To ensure the shell activity log is complete, the user sessions are terminated if logging is not possible. Refer to Section 6.1.1 and 6.1.2 for details.

Monitoring Monitoring the performance is possible by monitoring the number of files stored on the file system that contains the shell activity log data and the system logs. Additionally, the free space on this file system can be monitored. The solution does not store data in other places. Refer to Section 6.1.2 for details.

Data Deletion As mentioned in the „Monitoring“ paragraph, the solution only uses a single data storage location on each system. It is therefore possible to easily delete all activity auditing data at once. Refer to Section 6.1.2 for details.

Usability The solution provides search, replay and viewing capabilities for the shell activity logs. The system logs can also be searched and viewed. Refer to Section 6.1.3 for details.

6.1.6 Concept Limitations

This concept provides an activity auditing solution for a specific context and environment. It builds upon various requirements (see Section 5.3) and a threat model, which also includes a risk assessment (see Section 5.2). Based on the risk assessment and the selected mitigations, the concept focuses on specific issues and only partially addresses others. If the concept shall be used in a different environment, this risk assessment may no longer apply and thus the concept must be adapted. Section 2.2.1 explains that even the threat model may not apply since no two systems are ever truly equal.

The threat model (Section 5.2) explains a situation in which a user, or an attacker, connects directly, via a network tunnel, to a course group VM. It also explains that auditing user activity on these course group VMs is very difficult and potentially impossible because students have administrative access to them. Therefore, these threats are not addressed by this concept. Section 2.2.2 explains that when addressing threats, it is important to not lose sight of the big picture. Shostack[96] notes that is easy to get obsessed with a particular part of the threat landscape and ignore other problems. He also states that time and money are essentially finite and thus they have to be spent wisely by performing risk management. As explained in Section 2.3 this results in trade-offs such as this one.

A similar issue, albeit arguably less severe, is that an attacker can disguise their attacks such that, even if they are contained in the activity logs, they are difficult to find. An example of such a disguised attack is one where the attacker puts the attack commands into a compressed script file. They then tell their shell to uncompress and execute this file after a specific amount of time has passed. While the shell is waiting, they continue to use it until it eventually executes the attack. An analyst that investigates the situation may need to review all activity log data of a user prior

to an incident to discover such issues. This is an inherent limitation of providing users with a full command shell since such features are part of, for example, the „bash“ shell. Learning to use this shell and its features is a part of the exercises that students perform in the exercise environment. In other environments, restricting access and features may be more suitable than implementing an activity auditing solution as described in this thesis. As Schneier[93] likes to explain: „Secure systems should be cut to the bone and made as simple as possible. There is no substitute for simplicity.“ [93].

6.2 Implementation Description

To simplify maintenance and increase dependability, the implementation reuses existing components where possible and configures them appropriately rather than creating custom code. This section describes some of the components, the reasoning behind their choice, and their interactions.

6.2.1 Implementation overview

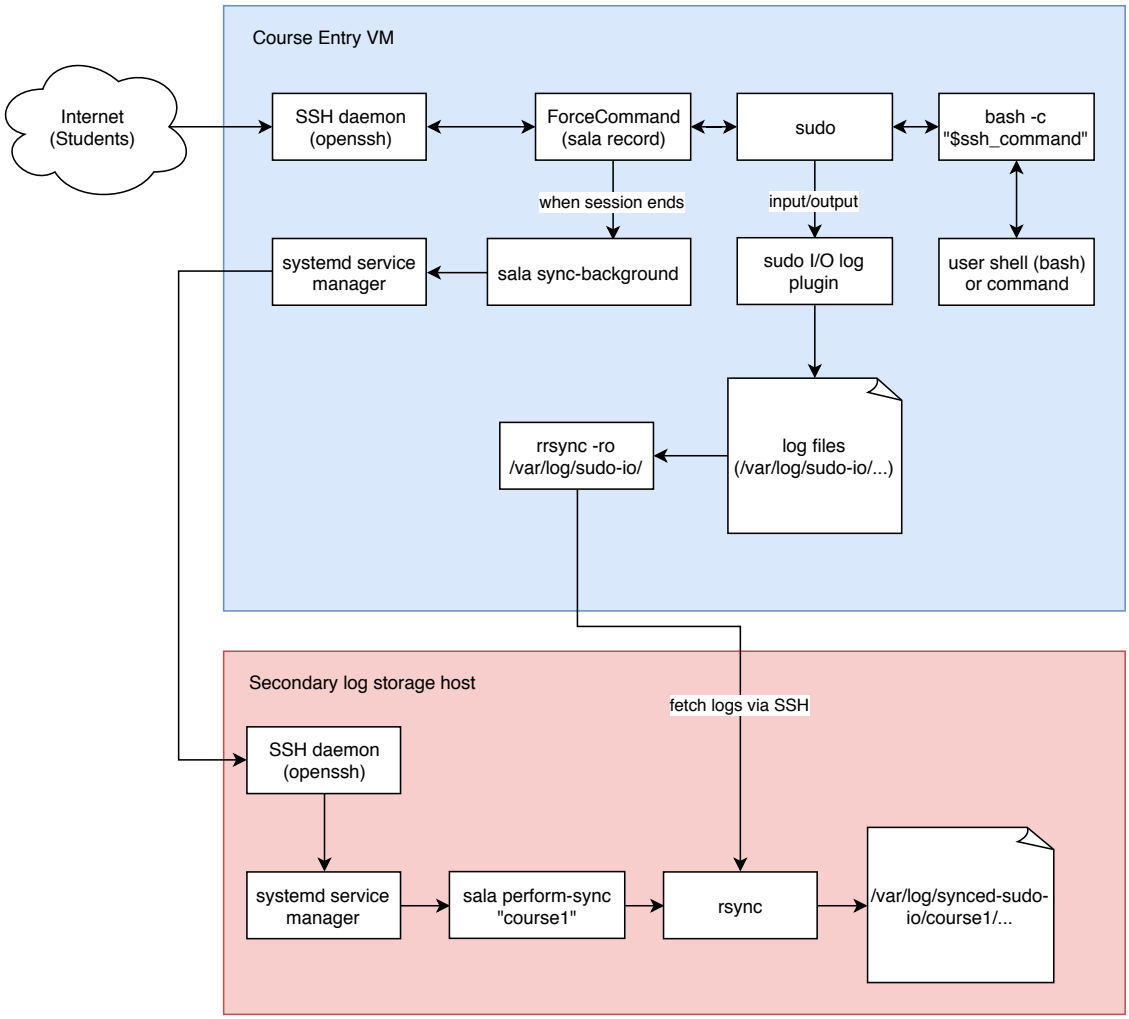

Figure 6.1: Simplified overview of the session recording and synchronization data flow.

The implementation builds on existing software, such as the OpenSSH daemon, the systemd service manager, the rsync file synchronization program, and the sudo authorization software. Some

glue code and maintenance commands are provided by the custom Shell Activity Logging and Auditing (sala) script. Figure 6.1 shows an overview of the interactions of these pieces.

When a student connects to the exercise environment, their SSH connection terminates at the SSH daemon on the course entry VM. There may exist multiple course entry VMs, one for each course, as explained in Section 5.1. The logging data is captured on each of the course entry VMs in the same way and, for simplicity, Figure 6.1 only shows a single course entry VM. The solution only uses a single secondary log storage host for all machines. There the logs of different course entry VMs are differentiated via an identifier as described later on in this section.

The SSH daemon on the course entry VM is configured to run a forced command instead of directly starting an interactive shell for each student. This forced command, „sala record“, uses the sudo application to start a sudo session to the user itself. The sudo application is generally used to switch to a different user, but it can also be used to switch to oneself. In addition to user switching, it contains an I/O logging plugin that allows recording each sudo session. This input/output log data is saved to the file system and automatically compressed while it is written. Additionally, the sudo application itself runs with administrative privileges, which allows it to write to files and directories which the user normally would not have access to. In this case, this means that the log files, written to „/var/log/sudo-io/“, can be and are restricted to only allow access from the root user. These restrictive permissions prevent confidentiality and integrity attacks by the users themselves (see Section 2.1.2).

Sudo is used to run a user-supplied command if available, or an interactive shell otherwise. A forced command receives the command from the OpenSSH server via an environment variable called „SSH_ORIGINAL_COMMAND“. Since the command may contain shell escape and quoting characters and these must be resolved before the command can be run, it is necessary to pass the command string to a shell for interpretation. This shell is the bash shell which is ran inside the sudo session. Interpreting the command inside sudo is important because this reduces the attack surface of the sala script. It ensures that any actions that may be performed based on the user-specified input are performed inside the sudo session and their effect is thus captured by the sudo I/O log.

When the user terminates their session, sala automatically starts a synchronization in the background. The „sala sync-background“ command starts a special systemd service which connects to the secondary log server via a dedicated SSH key. The SSH daemon on the secondary log server is configured to automatically start another systemd service via a forced command for each key. This service then starts the actual synchronization command via „sala perform-sync“, which also accepts a course identifier. The course identifier is different for each course and thus for each course entry VM. In Figure 6.1 the example identifier is „course1“ and it is configured as part of the forced command of each SSH key that can connect to the secondary log storage server. This identifier later allows investigators to distinguish log files stored on the secondary log storage host and determine which course they are associated with.

The synchronization is performed by using the rsync application to connect back to the course entry VM, also via SSH, and run the „rrsync“ command via another forced command. The rrsync command is a wrapper which is part of the rsync application package and which can be used to restrict the actions that can be performed by the rsync client. Especially, it allows configuring that the connection may only be used to read, and not write, files in a specified directory. This means that the secondary log server can read the log files created by the sudo I/O log, but it cannot change them. This is important to prevent an attacker from deleting or modifying the original log files alongside the secondary log files if they manage to gain access to the secondary log server. The secondary log server can also not access files outside the specified directory and it cannot run

any other commands than rsync. It is therefore restricted to using rsync to retrieve the allowed files.

In addition to the synchronisation triggered by a user terminating their SSH session, the systemd service manager on the secondary log storage server is configured to regularly, automatically trigger a synchronisation via a „timer“ service. Using the systemd service manager to trigger the synchronisation at both places, on the course entry VMs and on the secondary server, ensures that only a single synchronisation process can run at the same time. If a synchronisation job is already running, triggering either service again will not cause any problems and the request will simply be ignored. While this can be configured differently, the default behaviour prevents jobs from stacking up if many users connect and disconnect while a job is already running. Their logs may not be synchronised instantly when their session ends, but the regular timer still ensures that they are synchronised after a defined maximum time.

Analysis of the logs can be performed by using the „sudoreplay“ tool or by simply examining the log files directly. Sudoreplay is also part of the sudo application package and, for simplified usage, the sala script provides commands to view, replay, list and search in the recorded sessions. These commands reuse sudoreplay with some preselected options. Since the log files simply record the input and output data, they can also be viewed by using tools such as „less“, but due to compression, they need to be decompressed first. The „zless“ tool, which decompresses the file and then passes it to „less“, can also be used.

Finally, the sala command also provides commands to delete the logs on the course entry VMs and on the secondary storage host. Similarly to the analysis commands, these commands are thin wrappers around the „rm“ tool with certain preselected options and primarily serve to simplify usage of the entire system.

6.2.2 Security Considerations

The solution uses the OpenSSH daemon for automated cross-machine communication. As explained in Section 2.5.4, this may also allow attackers to access machines in unintended ways if the access key are not managed and deployed carefully. Such problems are prevented by using the „restrict“ option for each automated access key in the authorised keys file. The OpenSSH Authors[73] explain that the „restrict“ option is a special option that enables all current and future restrictions. This means that only explicitly listed actions are allowed [73]. In addition to the „restrict“ option, a forced command is configured for each key which limits its potential usage.

It is also important to remove old keys when they are no longer used, such as when a semester is over and the course entry VMs along with the log data shall be deleted. If the machines are kept or the solution is deployed in an environment where the user facing machines are not regularly deleted, it may be necessary to regularly rotate the automated access keys as explained in Section 2.5.4.

An attacker that gains administrative access to a course entry VM may modify locally stored log files. To prevent such modifications from being synchronised to the secondary server, rsync is configured to never change files and instead rename the old file by adding a timestamp to the file name. This renaming may result in many files being created if an attacker continually modifies all log files and then triggers a synchronisation. It is therefore important to monitor the number of files on each machine and ensure that the secondary server always has sufficient free space and a sufficient number of free inodes to store new log data.

The disk on a course entry VMs may also fill up and prevent new log files from being created or new log data from being written to existing, active log files. For this case, the sudo application has an option, which is enabled by default, to automatically terminate the user session when errors

occur while writing to the log file. However, sudo also transparently compresses the log file during creation and this compression works better, if it compresses more data at once. Therefore, by default, log data is kept in memory until a threshold is reached and only then it is compressed and written to the log file. If a write error occurs at this point, all log data that has been buffered in memory may be lost and not stored in the log file. Depending on which actions the user is performing, this buffer may span an extended period of time. This behaviour can be changed by enabling the „iolog_flush“ option, however this „may significantly reduce the effectiveness of I/O log compression“ according to the „sudoers“ manual created by Miller[66].

Finally, the „rrsync“ application created by Smith and Davison[98] only restricts the options that may be sent to the rsync server application. The rsync server application is executed on the remote server as part of an rsync client fetching data. The source code of rrsync contains a note saying that it „assumes that the rsync protocol will not be maliciously hijacked“ [98]. It is not explained if the rsync protocol contains commands that may allow an attacker to perform actions that should not be allowed by the server, but even if the protocol itself were secure now, this may change in the future, especially if the rsync application is changed and new features are added. It is therefore important to protect the SSH key used by the secondary server and the server itself with care. Even if the protocol is secure, the application may contain bugs, just like any other software, and thus an attacker may be able to attack the course entry VMs from the secondary storage server. Similarly, the SSH server itself may contain security issues, which is why Section 2.5.4 advises to regularly update it and its configuration.

6.3 Evaluation and Testing

This section evaluates the effectiveness of the solution in two areas. First, an example scenario is analysed by performing an attack and showing how this attack can be discovered in the audit data. Second, performance and the overhead of the data collection is measured and compared to a system without data collection.

6.3.1 Testing Method and Environment

The evaluation and performance tests are performed in two virtual machines. One machine represents a course entry VM, while the other represents the secondary log storage host. Both machines are shown in Figure 6.1 in Section 6.2.1.

Both virtual machines use the Linux-based Debian operating system distribution in version 9 (codename Stretch[1]). The host system that provides the hardware for the virtual machines runs Arch Linux with the 4.19.6 Linux kernel. The host system uses an Intel E3-1230v3 CPU on a Supermicro X10SAE mainboard with 32GB of ECC memory. Each of the virtual machines is provided 1GiB of memory and a single CPU core. For storage, the host uses a 500GB Crucial MX200 SSD on which the virtual machine's data is stored in „qcow2“ images. These images are attached to the virtual machines via the „VirtIO“ driver. The data on the host's SSD is encrypted with „LUKS“, a Linux disk encryption solution. The encryption uses LUKS' „AES-XTS-plain64“ cipher with a key size of 512 bits.

6.3.2 Example Scenario Walkthrough

An example DoS attack where it is difficult to determine if a user follows the exercise description or not, is described in the threat model in Section 5.2. In some exercises a student may need to

[1] https://wiki.debian.org/DebianStretch (visited on 2019-05-02)

perform repeated requests against a network service, such as a web server. While the exercise may ask students not to overload the service, an attacker can purposefully ignore this and perform a DoS attack by sending more requests than the service can handle.

A command that an attacker might execute to purposefully overload a web server is shown in Listing 6.1. It consists of a loop that performs 30 iterations. Each of these iterations start a sub shell in the background, with an endless loop of wget commands inside. Thus the command runs 30 instances of wget in parallel. An attacker may run much more than 30 instances so that the target service becomes overloaded and cannot respond to requests from other students in time.

```
1 for i in {0..30}; do (while :; do wget --quiet -O/dev/null http
  ↪ ://192.168.4.247/; done ) & done
```

Listing 6.1: Example DoS attack command that tries to overload a web server with many requests by running 30 instances of wget in parallel.

An analyst that investigates the problem may start at different points, depending on how they are notified about the issue. In one case, they may be notified by a monitoring system that the load on a machine, such as the web server or the course entry VM, is high. In other cases, they may be notified that the web server is returning errors due to too many requests and other students are unable to work on their assignments. After investigation in the web server's error and request log files they will eventually determine that the requests are coming from the course entry VM. If the attack is still on-going they may see the culprit directly by looking at active processes on the course entry VM.

If the attacker has already stopped attacking the target when the analyst investigates, they need to perform further analysis. Due to having been notified about the problem and having confirmed it via the web server's access logs, the analyst has a date and time they can use as a starting point. In this case, they can use the activity audit logs recorded by „sala“. Using sala, the logs can be listed, viewed, replayed, and searched as shown in the remainder of this section.

Listing Session Recordings The sala application provides a command that lists all recorded sessions. This list is well suited for an initial investigation, especially if there are few active users on the system or if a date and time for the incident are available. Listing 6.2 shows a few lines of output that are produced by the „sala list todate '2018-12-11 16:07'“ command. The command already limits the output such that it only includes session which were started before the incident took place. Sessions started later are of no interest since they could not possibly have executed a command at the incident time. The output shows that only „user1“ has used the course entry VM shortly before the incident. It also shows that user1 always started an interactive command shell and never supplied a command to be run directly. Further analysis will require the „TSID“ value of each session since it is used to retrieve the activity log for this session.

```
1 [trimmed for brevity]
2 Dec 10 17:49:00 2018 : user1 : TTY=/dev/pts/0 ; CWD=/home/user1 ; USER=user1
  ↪  ; TSID=00000T ; COMMAND=/bin/bash -c /bin/bash
3 Dec 10 18:10:09 2018 : user1 : TTY=/dev/pts/0 ; CWD=/home/user1 ; USER=user1
  ↪  ; TSID=00000U ; COMMAND=/bin/bash -c /bin/bash
4 Dec 10 18:22:53 2018 : user1 : TTY=/dev/pts/0 ; CWD=/home/user1 ; USER=user1
  ↪  ; TSID=00000V ; COMMAND=/bin/bash -c /bin/bash
5 Dec 11 15:55:39 2018 : user1 : TTY=/dev/pts/1 ; CWD=/home/user1 ; USER=user1
  ↪  ; TSID=00000W ; COMMAND=/bin/bash -c /bin/bash
6 Dec 11 16:04:50 2018 : user1 : TTY=/dev/pts/1 ; CWD=/home/user1 ; USER=user1
  ↪  ; TSID=00000X ; COMMAND=/bin/bash -c /bin/bash
```

Listing 6.2: Example output of „sala list todate '2018-12-11 16:07'“ with 16:07 being the time of attack, which has been determined by analysing the web server's request logs.

Viewing a Session Recording After determining which sessions were recorded before the incident time, the analyst proceeds to review each sessions actions to determine if any of them performed the attack. Using the „sala view $TSID“ command, they can view the complete output of each session, as shown in Listing 6.3. „$TSID“ represents a TSID value shown in the „sala list“ output. Since interactive shells output the keys pressed by the user, viewing the output of the session also allows the analyst to see which commands the user executed. The analyst starts with the newest session and works their way backwards in time until they discover a session that looks suspicious. In this case, they start with the last session in the list, which is the session that was recorded at 16:04:50 and which has the TSID „00000X“. In this example, this session already contains the attack code in line 2. For comparison, the attack code is also shown in Listing 6.1. In other situations, it may be necessary to analyse additional, older sessions if the last session in the list is not the one that performed the attack.

```
1 Replaying sudo session: /bin/bash -c /bin/bash
2 0;user1@debian: ~^Guser1@debian:~$ for i in {0..30}; do (while :; do wget --
  ↪ quiet -O/dev/null http://192.168.4.247/; done ) & done
3 [1] 2196
4 [2] 2197
5 [trimmed for brevity]
6 [31] 2260
7 0;user1@debian: ~^Guser1@debian:~$ ^G^G^G^G^G^G^G^G^M0;user1@debian: ~^
  ↪ Guser1@debian:~$ ^M0;user1@debian: ~^Guser1@debian:~$ exit
```

Listing 6.3: Example output of „sala view 00000X | less -R“, which shows the recorded session data of the attack. „less -R“ is used to interpret color information from the output and to make it slightly more readable.

Replaying a Session Recording While being able to view a session recording is helpful, the output can be confusing for multiple reasons. First, the output may contain control sequences, which place parts of the output at certain positions on a screen, or which may change the color of the text. In Listing 6.3, color information has been removed by using the „less -R“ command, however, other control sequences are still present. Second, it may be meant for a wider or narrow screen than what is being used to view it. This can result in lines breaking in places where they originally did not break and thus resulting is misaligned output.

A way around these limitations is replaying the activity log in a video-like fashion by using the „sala replay $TSID“ command. Again, „$TSID“ represents a TSID value shown in the „sala list“ output. The replay command interactively shows the session recording as it was seen by the original user. The analyst can pause the replay at any point and increase or decrease the playback

speed. They can also skip long pauses, for example, if the user waited for some time before entering a command. Any control characters that are part of the recording will have their correct effect and not be directly visible.

Searching for a Known Pattern in Session Recordings If an analyst knows what to look for, but does not know where to start, they can search all activity logs that match a specific pattern. Listing 6.4 shows the output of the „sala search 192.168.4.247“ command which searches for the pattern „192.168.4.247“. The pattern in this command, „192.168.4.247“, is the IP address of the web server that was attacked in this scenario. The output lines start with the file name and path of the log file that contains the match, followed by the matching line. In this case, there are matches in the input and output log files. The „ttyin“ file contains the data that was received by the course entry VM, while the „ttyout“ file contains the data which was sent back to the attacker. Searching is performed by using the „grep“ application, which means that it's options can also be used. For example, the „-l“ option limits the output to only show the file paths and not the matching lines. The path to the file path contains the TSID value („00000X“) which was shown in earlier examples in this section, however, it is shown in a slightly different format here („00/00/0X“). Using this value, an analyst can view or replay logs as described above.

```
1 [trimmed for brevity]
2 /var/log/sudo-io/00/00/0X/ttyout:ESC]0;user1@debian: ~^GESC[01;32
  ↪ muser1@debianESC[00m:ESC[01;34m~ESC[00m$ for i in {0..30}; do (while :;
  ↪ do wget --quiet -O/dev/null http://192.168.4.247/; done ) & done
3 /var/log/sudo-io/00/00/0X/ttyin:for i in {0..30}; do (while :; do wget --
  ↪ quiet -O/dev/null http://192.168.4.247/; done ) & done^Mhtop
  ↪ ^?^?^?^?^?^?^?^?^?^?^?^?^?
```

Listing 6.4: Example output of „sala search 192.168.4.247“, which shows matches in log files that contain the search pattern. In this case the pattern is the IP address of the web server which was attacked.

6.3.3 Performance Evaluation

This section compares several performance characteristics between a user that has activity audit logging enabled and one where the sala logging code is not used. The second case is used to measure a base-line performance of the setup so that the difference between a system with and without audit logging can be determined.

Throughput The throughput of the SSH connection describes how much data can be transferred if bulk data transfers are performed. An example for this are file transfers of large files over the SSH connection. For the purpose of this test, the SSH client application is run on the host to avoid being limited by networking hardware. The test transfers 1GiB of zeros, read form the special file „/dev/zero“. Data is transferred only in one direction at the time. When downloading data from the virtual machine to the host, the baseline is 172MiB/s, while with activity logging enabled this value drops to 116MiB/s. When uploading data from the host to the virtual machine, the baseline is 175MiB/s, while it drops to 102MiB/s with logging enabled. Each test was performed 3 times and the best (highest) result is reported here.

Latency The latency is the time between sending data to the server and receiving it back. This is interesting during interactive usage of an interactive shell since the remote shell shows the user which keys they entered and which command they are about to execute. To measure the effect of the sala activity auditing solution, a script has been created that mimics this process and measures how long it takes until the data comes back. The script uses an SSH connection

with the „cat“ application as the remote command. The „cat“ application reads data from the standard input and writes it back out on standard output. The script writes the string „ping“ to the connection and reports how long it takes to be read back. This process is repeated 2000 times and the average, median, min, and max values are shown in Table 6.1. Using sala increases the latency by approximately 30 microseconds on average.

Table 6.1: Comparison between the input/output latency of an SSH connection with and without being recorded by the sala activity auditing solution.

	Without sala	With sala	Difference
Mean	0.09528ms	0.12492ms	+30µs
Median	0.09336ms	0.12109ms	+28µs
Min	0.07438ms	0.09690ms	+23µs
Max	0.30068ms	0.43250ms	+132µs

Disk Usage Debian's „sudo“ package is built without compression support[2] at the time of this evaluation. Therefore, to investigate the effect of compression on the log file size, this test is performed on the host system, which uses a different Linux distribution where sudo is built with compression support. Table 6.2 shows the results for the usage scenarios shown below:

- Manpage: View the „sudo“ manpage and scroll down page by page by using the „Page Down“ key until the end of the file.
- Dmesg: Call dmesg 10 times to view the kernel's log messages.
- Vi: Use the „vi“ text editor to create a file that contains the text „Hello World!“.
- Echo: Run the command „echo test“.

Each scenario is performed once in a terminal sized to fit 100 columns and 30 lines. As shown in Table 6.2, the compression efficiency differs greatly between different usage scenarios. Compression works especially well for repeated output, such as the repeated dmesg, and large amounts of text, such as in the scrolling manpage. Note that the table only refers to the data size of each file. It does not consider how much space these files actually occupy on disk due to file system and metadata overhead.

Table 6.2: Comparison of compression efficiency for activity auditing log files of different usage scenarios. „Total Log Size“ includes the size of the recorded input/output data, as well as the timing information needed for session replay.

Scenario Name	Total Log Size Compressed	Total Log Size Uncompressed	Compressed Size
Dmesg	211KiB	1537KiB	13.7%
Manpage	13KiB	45KiB	28.9%
VI	828B	1.5KiB	53.9%
Echo	427B	540B	79.1%

[2] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=846077 (visited on 2019-05-02)

7 Conclusion and Outlook

As described in Chapter 1, this thesis develops an activity auditing concept for a security exercise environment. Towards this goal, it first describes relevant foundations of IT security, which include a definition of terms such as confidentiality, integrity, availability, dependability, and authenticity. It highlights that some of these terms are interconnected, and in practice it may be impossible to fully address each of these aspects. A solution towards this problem is presented in the form of threat modelling and associated risk management. This approach allows selecting appropriate security measures for a specific context and thereby addressing potential conflicts between security requirements. The thesis also discusses security aspects of open source software and common pitfalls, such as believing that every project's security is being reviewed by others. It also explains how the SSH protocol works and which potential security issues should be considered when using it.

Based on these foundations, the thesis continues to discuss the basics of logging. Logging can range from simple locally written log files to complex, networked systems with aggregation and processing steps in-between the log source and the log storage. Log sources may be of various types, each of which have their own benefits and drawbacks. For example, application logs have a unique view of application specific data, but it is also often necessary to properly protect this data, especially if it contains personal information. If such data is logged, it becomes necessary to protect the transmission, processing, storage, access, and destruction of this log data.

Building upon normal logging, activity auditing solutions and approaches are investigated. Sources that can provide auditing data include general log data sources or more specialised variants, such as the linux audit framework and LSM. The thesis then discusses analysis approaches of audit data to discover and prove misuse of protected systems. Being able to provide a convincing proof places several requirements on the audit data generation and storage. For example, it may be necessary to use tamper-evident storage systems so that audit data cannot be modified by administrative personnel.

Finally, the case example and a concept tailored towards it are presented. This concept builds upon the foundations and aspects described in this thesis and combines them into a working solution for the example context. To prove this claim, the concept is implemented and this implementation is evaluated and tested.

The introduction in Section 1.3 also posed three research questions. With the concept and its implementation and testing results, these can now be answered as follows:

- **Research Question 1**: What are the requirements for and the threat model of an auditing solution in the context of a security lecture exercise environment?

  The threat model is described in Section 5.2 and includes various forms of DoS attacks as well as network attacks, such as ARP spoofing and traffic tunneling. The requirements of the case example are described in Section 5.3 and are mainly concerned with the content of the log files and the security properties of the system.

- **Research Question 2**: Is it possible to define a technical concept that satisfies all requirements and can be implemented efficiently?

It has been shown that a technical solution can be implemented and operated with justifiable effort. A concept that satisfies the requirements is provided in Section 6.1. Section 6.1.5 explains why and how this concept satisfies all requirements. An overview of the implementation is shown and explained in Section 6.2.1. This implementation is evaluated along with an example scenario in Section 6.3.

- **Research Question 3**: How can the audit data be evaluated to detect misuse of the exercise environment and is it possible to integrate automatic auditing solutions into this concept considering the requirements?

  The collected audit data can be evaluated, as described in Section 6.1.3, by viewing and replaying relevant log files, and searching for known patterns. Section 6.3.2 shows that this can indeed be used to detect misuse. Different automated analysis solutions have been discussed in Section 4.4. However, as described in Section 6.1.2, their usage results in a requirements conflict with regards to dependability. Based on a risk based evaluation they have, therefore, not been used as part of this concept.

While the presented concept satisfies the requirements, there is still room for potential improvement, especially if a similar solution shall be implemented in different environments. Section 6.1.6 explains how an attacker might be able to prepare their attack in advance and execute it later. By hiding their attack like this, an analyst may need to analyse a much longer time frame, which could be especially problematic on bigger systems that are used for a longer time frame than a single semester (i.e. 4 months). In the context of the exercise environment, the course entry VMs can be decommissioned after the course is over, but in other environments, systems may be online for many years. Thus, logs can accumulate over many years and analysing all of the accumulated logs may be impractical. Therefore, future research should investigate if this problem can be addressed in such a way that the resulting log files can be analysed more efficiently.

Section 6.2.1 explains that activity audit log data is synchronised to the secondary log storage host either when a session terminates or when a timer expires. This means that there is a window of time where an attacker can perform actions that are not immediately synchronised to the secondary log storage host. If the attacker manages to obtain administrative privileges in that time, they may be able to disable the secondary system's access before the log data is transfered. Since access is denied for the secondary storage system, the log that contains the attack cannot be transfered. Future research should explore possibilities to synchronize log data to the secondary storage host online, instead of performing scheduled synchronisation.

Bibliography

References

[1] A. S. Abed, C. Clancy, and D. S. Levy. „Intrusion Detection System for Applications Using Linux Containers“. In: *Security and Trust Management*. Springer International Publishing, 2015, pp. 123–135. DOI: 10.1007/978-3-319-24858-5_8.

[2] J. Anderson. „Why we need a new definition of information security“. In: *Computers & Security* 22.4 (2003), pp. 308–313.

[4] A. Avizienis et al. „Basic concepts and taxonomy of dependable and secure computing“. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[5] M. Balduzzi et al. „A Security Analysis of Amazon's Elastic Compute Cloud Service“. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy: ACM, 2012, pp. 1427–1434. ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2232005.

[6] A. Barisani. *tenshi Manual*. tenshi version 0.17. 2017.

[8] E. Barker. *Recommendation for Key Management*. Tech. rep. National Institute of Standards and Technology (NIST), Jan. 2016. DOI: 10.6028/NIST.SP.800-57pt1r4.

[9] M. Bauer. „Paranoid Penguin: Linux Filesystem Security, Part II“. In: *Linux Journal* 2004.127 (Nov. 2004), p. 11. ISSN: 1075-3583. URL: https://dl.acm.org/citation.cfm?id=1029015.1029026 (visited on 2019-05-02).

[10] W. Becker et al. *Projektrisikomanagement im Mittelstand*. Springer, 2015. DOI: 10.1007/978-3-658-05316-1.

[13] H. Bernstein. „Risikoanalyse und deren Beurteilung“. In: *Sicherheits- und Antriebstechnik*. Ed. by T. Zipsner. Springer Fachmedien Wiesbaden, 2016. Chap. 2, pp. 33–81. DOI: 10.1007/978-3-658-12934-7_2.

[14] I. Beschastnikh et al. „Debugging Distributed Systems“. In: *Queue* 14.2 (Mar. 2016), 50:91–50:110. ISSN: 1542-7730. DOI: 10.1145/2927299.2940294.

[15] J. Bethencourt, A. Sahai, and B. Waters. „Ciphertext-Policy Attribute-Based Encryption“. In: *2007 IEEE Symposium on Security and Privacy*. IEEE, May 2007. DOI: 10.1109/sp.2007.11.

[16] J. Biskup and U. Flegel. „Transaction-Based Pseudonyms in Audit Data for Privacy Respecting Intrusion Detection“. In: *Recent Advances in Intrusion Detection*. Ed. by H. Debar, L. Mé, and S. F. Wu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 28–48. ISBN: 978-3-540-39945-2.

[17] H.-C. Brauweiler. *Risikomanagement in Unternehmen*. Springer-Verlag, 2014. DOI: 10.1007/978-3-658-07721-1.

[18] D. Chang et al. „Rig: A Simple, Secure and Flexible Design for Password Hashing“. In: *Information Security and Cryptology*. Ed. by D. Lin, M. Yung, and J. Zhou. Cham: Springer International Publishing, 2015, pp. 361–381. ISBN: 978-3-319-16745-9. DOI: 10.1007/978-3-319-16745-9_20.

[19] G. Chang et al. „A novel approach to automated, secure, reliable, & distributed backup of mer tactical data on clouds“. In: *2012 IEEE Aerospace Conference*. IEEE, Mar. 2012. DOI: 10.1109/AERO.2012.6187355.

[20] T. Chomsiri. „Sniffing packets on LAN without ARP spoofing“. In: *2008 Third International Conference on Convergence and Hybrid Information Technology*. Vol. 2. IEEE. IEEE, Nov. 2008, pp. 472–477. DOI: 10.1109/iccit.2008.318.

[21] A. Chuvakin and G. Peterson. „How to Do Application Logging Right“. In: *IEEE Security Privacy* 8.4 (July 2010), pp. 82–85. ISSN: 1540-7993. DOI: 10.1109/MSP.2010.127.

[22] A. A. Chuvakin and K. J. Schmidt. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*. Ed. by P. Moulder. 1st. Syngress Publishing, Nov. 29, 2012. 460 pp. ISBN: 9781597496353.

[23] M. Dagenais et al. „Software Performance Analysis“. In: *CoRR* abs/cs/0507073 (2005). URL: https://arxiv.org/abs/cs/0507073 (visited on 2019-05-02).

[24] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. Internet Requests for Comments. RFC. Aug. 2008. DOI: 10.17487/RFC5246.

[25] N. V. Dijkhuizen and J. V. D. Ham. „A Survey of Network Traffic Anonymisation Techniques and Implementations“. In: *ACM Comput. Surv.* 51.3 (May 2018), 52:1–52:27. ISSN: 0360-0300. DOI: 10.1145/3182660.

[26] G. Dionne. „Risk Management: History, Definition, and Critique“. In: *Risk Management and Insurance Review* 16.2 (Sept. 2013), pp. 147–166. DOI: 10.1111/rmir.12016.

[27] H. Doubleday, L. Maglaras, and H. Janicke. „SSH Honeypot: Building, Deploying and Analysis“. In: *International Journal of Advanced Computer Science and Applications* 7.5 (2016). DOI: 10.14569/ijacsa.2016.070518.

[28] R. Duncan and M. Whittington. „Creating an Immutable Database for Secure Cloud Audit Trail and System Logging“. In: *Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017-23 February 2017, Athens, Greece*. 2017, pp. 54–59.

[30] J. G. Elerath and M. Pecht. „Enhanced Reliability Modeling of RAID Storage Systems“. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. IEEE, June 2007, pp. 175–184. DOI: 10.1109/dsn.2007.41.

[31] European Union. „Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)“. In: *Official Journal of the European Union* L 119 (2016). URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=OJ:L:2016:119:FULL&from=EN (visited on 2019-05-02).

[32] F. Fankhauser, C. Schanes, and C. Brem. „Softwaretechnik - Mit Fallbeispielen aus realen Entwicklungsprojekten“. In: 1st ed. München: Pearson Studium, 2009. Chap. 13, pp. 589–646. ISBN: 978-3-86894-007-7.

[33] U. Flegel. „Pseudonymizing Unix Log Files“. In: *Infrastructure Security*. Ed. by G. Davida, Y. Frankel, and O. Rees. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 162–179. ISBN: 978-3-540-45831-9.

[34] S. Garfinkel, A. Schwartz, and G. Spafford. *Practical UNIX and Internet Security, 3rd Edition*. O'Reilly, 2003. ISBN: 0596003234.

[35] S. Garfinkel and G. Spafford. *Practical UNIX and Internet Security, 2nd Edition*. O'Reilly, 1996. ISBN: 1-56592-148-8.

[36] R. Gerhards. *The Syslog Protocol*. RFC 5424. Mar. 2009. DOI: 10.17487/RFC5424.

[38] A. Gkortzis, D. Mitropoulos, and D. Spinellis. „VulinOSS: A Dataset of Security Vulnerabilities in Open-source Systems“. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: ACM, 2018, pp. 18–21. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196454.

[39] V. Goyal et al. „Attribute-based Encryption for Fine-grained Access Control of Encrypted Data“. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. Alexandria, Virginia, USA: ACM, 2006, pp. 89–98. ISBN: 1-59593-518-5. DOI: 10.1145/1180405.1180418.

[40] S. Grubb. *auditctl Manual*. audit version 2.8.4. 2017.

[41] B. Güntert. „Konzepte und Ansätze des Risikomanagements in Gesundheitseinrichtungen“. In: *Gesundheitsqualität als Aufgabe* (2006), p. 125.

[42] B. Guttman and E. Roback. *An introduction to computer security: the NIST handbook*. Tech. rep. National Institute of Standards and Technology (NIST), 1995. DOI: 10.6028/NIST.SP.800-12.

[43] A. Haeberlen, P. Kouznetsov, and P. Druschel. „PeerReview: Practical Accountability for Distributed Systems“. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 175–188. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294279.

[44] Z. Han et al. „Risk assessment of digital library information security: a case study“. In: *The Electronic Library* 34.3 (June 2016), pp. 471–487. DOI: 10.1108/el-09-2014-0158.

[45] P. He et al. „Towards Automated Log Parsing for Large-Scale Log Data Analysis“. In: *IEEE Transactions on Dependable and Secure Computing* (2017). ISSN: 1545-5971. DOI: 10.1109/TDSC.2017.2762673.

[46] S. He et al. „Experience Report: System Log Analysis for Anomaly Detection“. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. Oct. 2016, pp. 207–218. DOI: 10.1109/ISSRE.2016.21.

[47] I. Heckmann, T. Comes, and S. Nickel. „A critical review on supply chain risk – Definition, measure and modeling“. In: *Omega* 52 (Apr. 2015), pp. 119–132. DOI: 10.1016/j.omega.2014.10.004.

[48] J.-H. Hoepman and B. Jacobs. „Increased security through open source“. In: *Communications of the ACM* 50.1 (Jan. 2007), pp. 79–83. DOI: 10.1145/1188913.1188921.

[49] C. Humphries et al. „ELVIS: Extensible Log VISualization“. In: *Proceedings of the Tenth Workshop on Visualization for Cyber Security*. VizSec '13. Atlanta, Georgia, USA: ACM, 2013, pp. 9–16. ISBN: 978-1-4503-2173-0. DOI: 10.1145/2517957.2517959.

[50] ISO/IEC. *ISO/IEC 27000 - Information technology — Security techniques — Information security management systems — Overview and vocabulary*. Tech. rep. ISO, 2018.

[51] M. Jahoda et al. *Red Hat Enterprise Linux 6 Security Guide*. Red Hat, Inc., 2018. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/pdf/security_guide/Red_Hat_Enterprise_Linux-6-Security_Guide-en-US.pdf (visited on 2019-05-02).

[52] J. Keniston et al. „Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps“. In: *Proceedings of the Linux Symposium*. 2007, pp. 215–224.

[53] K. Kent and M. Souppaya. *Guide to Computer Security Log Management*. Tech. rep. National Institute of Standards and Technology (NIST), 2006. DOI: 10.6028/NIST.SP.800-92.

[54] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style, 2nd Edition*. McGraw-Hill, 1978. ISBN: 0070342075.

[55] E. Kiciman et al. „Mining Web Logs to Debug Distant Connectivity Problems“. In: *Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data*. MineNet '06. Pisa, Italy: ACM, 2006, pp. 287–292. ISBN: 1-59593-569-X. DOI: 10.1145/1162678.1162680.

[56] H.-D. Kochs. *System Dependability Evaluation Including S-dependency and Uncertainty*. Cham: Springer International Publishing, 2018. ISBN: 9783319649917. DOI: 10.1007/978-3-319-64991-7.

[57] I. Koniaris, G. Papadimitriou, and P. Nicopolitidis. „Analysis and visualization of SSH attacks using honeypots“. In: *Eurocon 2013*. July 2013, pp. 65–72. DOI: 10.1109/EUROCON.2013.6624967.

[58] C. Lim, N. Singh, and S. Yajnik. „A log mining approach to failure analysis of enterprise telephony systems“. In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. June 2008, pp. 398–403. DOI: 10.1109/DSN.2008.4630109.

[59] Q. Lin et al. „Log Clustering Based Problem Identification for Online Service Systems“. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. Austin, Texas: ACM, 2016, pp. 102–111. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889232.

[60] S. Lins, S. Schneider, and A. Sunyaev. „Trust is Good, Control is Better: Creating Secure Clouds by Continuous Auditing“. In: *IEEE Transactions on Cloud Computing* PP.99 (2017), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2016.2522411.

[61] G. Lowe. „A hierarchy of authentication specifications“. In: *Proceedings 10th Computer Security Foundations Workshop*. June 1997, pp. 31–43. DOI: 10.1109/CSFW.1997.596782.

[62] R. Marty. „Cloud Application Logging for Forensics“. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: ACM, 2011, pp. 178–184. ISBN: 978-1-4503-0113-8. DOI: 10.1145/1982185.1982226.

[63] V. Mateljan, K. Peter, and V. Juričić. „An Optimization of Command History Search“. In: *Digital Resources and Knowledge Sharing*. Ed. by D. Bawden et al. The Future of Information Sciences. Department of Information Sciences, Faculty of Humanities and Social Sciences, University of Zagreb, 2009, pp. 299–307.

[64] J. McDermid et al. „Experience with the application of HAZOP to computer-based systems“. In: *COMPASS '95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'*. IEEE, 1995. DOI: 10.1109/cmpass.1995.521885.

[65] D. Mellado and D. Rosado. „An overview of current information systems security challenges and innovations J. UCS Special Issue“. In: *Journal of Universal Computer Science* 18.12 (2012), pp. 1598–1607. DOI: 10.3217/jucs-018-12.

[66] T. C. Miller. *sudoers Manual*. sudo version 1.8.25p1. 2018.

[67] J. Morris, S. Smalley, and G. Kroah-Hartman. „Linux security modules: General security support for the linux kernel“. In: *USENIX Security Symposium*. 2002.

[68] S. Myagmar, A. J. Lee, and W. Yurcik. „Threat modeling as a basis for security requirements“. In: *Symposium on requirements engineering for information security (SREIS)*. Vol. 2005. 2005, pp. 1–8.

[69] B. Nagpal et al. „Tool based implementation of SQL injection for penetration testing“. In: *International Conference on Computing, Communication Automation*. May 2015, pp. 746–749. DOI: 10.1109/CCAA.2015.7148509.

[70] M. Nieles, K. Dempsey, and V. Y. Pillitteri. *An introduction to information security*. Tech. rep. National Institute of Standards and Technology (NIST), June 2017. DOI: 10.6028/nist.sp.800-12r1.

[71] H. F. Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616.

[72] A. Oliner, A. Ganapathi, and W. Xu. „Advances and Challenges in Log Analysis“. In: *Communications of the ACM* 55.2 (Feb. 2012), pp. 55–61. ISSN: 0001-0782. DOI: 10.1145/2076450.2076466.

[73] OpenSSH Authors. *OpenSSH SSH daemon Manual*. OpenSSH version 7.9p1. 2018.

[74] OpenSSH Authors. *ssh_config - OpenSSH SSH client configuration files*. 2018.

[75] A. Oprea et al. „Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data“. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. June 2015, pp. 45–56. DOI: 10.1109/DSN.2015.14.

[76] Oracle. *MySQL Reference Manual*. MySQL version 8.0. 2018.

[77] A. Oram and J. Viega. *Beautiful Security, 1st Edition*. O'Reilly Media, Inc., 2009. ISBN: 0-596-80178-5.

[78] D. B. Parker. „Toward a New Framework for Information Security?“ In: *Computer Security Handbook*. Ed. by S. Bosworth, M. E. Kabay, and E. Whyne. John Wiley & Sons, Inc., Sept. 2012, pp. 31–323. DOI: 10.1002/9781118851678.ch3.

[79] T. Pasquier et al. „Practical Whole-system Provenance Capture“. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: ACM, 2017, pp. 405–418. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3129249.

[80] D. A. Patterson, G. Gibson, and R. H. Katz. „A case for redundant arrays of inexpensive disks (RAID)“. In: *ACM SIGMOD Record* 17.3 (June 1988), pp. 109–116. DOI: 10.1145/971701.50214.

[81] H. Perl et al. „VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits“. In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 426–437. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813604.

[82] J. Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[83] C. Preston. *Backup & recovery: inexpensive backup solutions for open systems*. O'Reilly Media, Inc., 2007. ISBN: 978-0596102463.

[84] D. Ramsbrock, R. Berthier, and M. Cukier. „Profiling Attacker Behavior Following SSH Compromises“. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. June 2007, pp. 119–124. DOI: 10.1109/DSN.2007.76.

[85] I. Ray et al. „Secure Logging as a Service—Delegating Log Management to the Cloud“. In: *IEEE Systems Journal* 7.2 (June 2013), pp. 323–334. ISSN: 1932-8184. DOI: 10.1109/JSYST.2012.2221958.

[86] M. Roesch. „Snort - Lightweight Intrusion Detection for Networks“. In: *Proceedings of the 13th USENIX Conference on System Administration*. LISA '99. Seattle, Washington: USENIX Association, 1999, pp. 229–238. URL: https://www.usenix.org/legacy/event/lisa99/full_papers/roesch/roesch.pdf (visited on 2019-05-02).

[87] P. Rogaway and T. Shrimpton. „Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance“. In: *Fast Software Encryption*. Ed. by B. Roy and W. Meier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388. ISBN: 978-3-540-25937-4. DOI: 10.1007/978-3-540-25937-4_24.

[89] E. Ruijters and M. Stoelinga. „Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools“. In: *Computer Science Review* 15-16 (Feb. 2015), pp. 29–62. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2015.03.001.

[90] C. Sanders. *Practical Packet Analysis: Using Wireshark to Solve Real-World Network Problems*. Ed. by S. Yang. 3rd. San Francisco, CA, USA: No Starch Press, 2017. ISBN: 1593278020, 9781593278021.

[91] Y. Sasaki and K. Aoki. „Finding Preimages in Full MD5 Faster Than Exhaustive Search“. In: *Advances in Cryptology - EUROCRYPT 2009*. Springer, 2009, pp. 134–152. DOI: 10.1007/978-3-642-01001-9_8.

[92] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 2015. ISBN: 978-0-4711-2845-8.

[93] B. Schneier. *Secrets and Lies*. Wiley Publishing, Inc., Oct. 2015. DOI: 10.1002/9781119183631.

[94] E. Schubert et al. „DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN“. In: *ACM Trans. Database Syst.* 42.3 (July 2017), 19:1–19:21. ISSN: 0362-5915. DOI: 10.1145/3068335.

[95] A. Shamir. „How to Share a Secret“. In: *Communications of the ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176.

[96] A. Shostack. *Threat Modeling: Designing for Security*. Ed. by C. Wysopal. John Wiley & Sons, Inc., 2014. ISBN: 978-1-118-80999-0.

[97] A. Smith and R. Mahajan. „National critical incident reporting: improving patient safety“. In: *British Journal of Anaesthesia* 103.5 (Nov. 2009), pp. 623–625. DOI: 10.1093/bja/aep273.

[98] J. Smith and W. Davison. *rrsync Source Code*. 2004.

[99] G. Stringhini, C. Kruegel, and G. Vigna. „Shady Paths: Leveraging Surfing Crowds to Detect Malicious Web Pages“. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: ACM, 2013, pp. 133–144. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516682.

[100] SUSE LLC. *SUSE Linux Enterprise Server 15 Security Guide*. Ed. by SUSE LLC. SUSE LLC, 2018. URL: https://www.suse.com/documentation/sles-15/pdfdoc/book_security/book_security.pdf (visited on 2019-05-02).

[101] A. Sutton and R. Samavi. „Blockchain Enabled Privacy Audit Logs“. In: *The Semantic Web – ISWC 2017*. Ed. by C. d'Amato et al. Cham: Springer International Publishing, 2017, pp. 645–660. ISBN: 978-3-319-68288-4. DOI: 10.1007/978-3-319-68288-4_38.

[104] C. Thompson and D. Wagner. „A Large-Scale Study of Modern Code Review and Security in Open Source Projects“. In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE. Toronto, Canada: ACM, 2017, pp. 83–92. ISBN: 978-1-4503-5305-2. DOI: 10.1145/3127005.3127014.

[105] P. Torr. „Demystifying the threat modeling process“. In: *IEEE Security Privacy* 3.5 (Sept. 2005), pp. 66–70. ISSN: 1540-7993. DOI: 10.1109/MSP.2005.119.

[106] S. Turner and L. Chen. *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*. Internet Requests for Comments. RFC. Mar. 2011. DOI: 10.17487/RFC6151.

[107] X. Wang et al. „Performance evaluation of Attribute-Based Encryption: Toward data privacy in the IoT“. In: *2014 IEEE International Conference on Communications (ICC)*. June 2014, pp. 725–730. DOI: 10.1109/ICC.2014.6883405.

[108] B. R. Waters et al. „Building an Encrypted and Searchable Audit Log“. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society, 2004. ISBN: 1-891562-18-5. URL: https://crypto.stanford.edu/~bwaters/publications/papers/audit_log.pdf (visited on 2019-05-02).

[109] C. Wickramage et al. „Challenges for Log Based Detection of Privacy Violations during Healthcare Emergencies“. In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. Dec. 2017, pp. 1–6. DOI: 10.1109/GLOCOM.2017.8254433.

[110] W. Xu et al. „Detecting Large-scale System Problems by Mining Console Logs“. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 117–132. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629587.

[111] J. Ya et al. „An automatic approach to extract the formats of network and security log messages“. In: *MILCOM 2015 - 2015 IEEE Military Communications Conference*. Oct. 2015, pp. 1542–1547. DOI: 10.1109/MILCOM.2015.7357664.

[112] D. Yaga et al. *Blockchain technology overview*. Tech. rep. National Institute of Standards and Technology (NIST), Oct. 2018. DOI: 10.6028/NIST.IR.8202.

[113] T. Ylonen and C. M. Lonvick. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. Jan. 2006. DOI: 10.17487/RFC4252.

[114] T. Ylonen and C. M. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Jan. 2006. DOI: 10.17487/RFC4251.

[115] T. Ylonen et al. *Security of Interactive and Automated Access Management Using Secure Shell (SSH)*. Tech. rep. National Institute of Standards and Technology (NIST), Oct. 2015. DOI: 10.6028/NIST.IR.7966.

[116] L. Zeng, Y. Xiao, and H. Chen. „Linux auditing: Overhead and adaptation“. In: *2015 IEEE International Conference on Communications (ICC)*. June 2015, pp. 7168–7173. DOI: 10.1109/ICC.2015.7249470.

Online References

[3] Austrian Standards. *ONR 49000 - Risikomanagement für Organisationen und Systeme*. 2014. URL: https://www.austrian-standards.at/fileadmin/user/bilder/downloads-produkte-und-leistungen/fachinformation06_risikomanagement.pdf (visited on 2019-05-02).

[7] A. Barisani. *Tenshi Website*. 2018. URL: https://github.com/inversepath/tenshi/blob/master/README.md (visited on 2019-05-02).

[11] D. Berman. *5 Logstash Filter Plugins You Need to Know About*. 2017. URL: https://logz.io/blog/5-logstash-filter-plugins/ (visited on 2019-05-02).

[12] D. Berman. *The Complete Guide to the ELK Stack – 2018*. 2018. URL: https://logz.io/learn/complete-guide-elk-stack/ (visited on 2019-05-02).

[29] Elasticsearch BV. *Elasticsearch Reference - Basic Concepts*. 2018. URL: https://www.elastic.co/guide/en/elasticsearch/reference/6.4/getting-started-concepts.html (visited on 2019-05-02).

[37] German Federal Office for Information Security (BSI). *Hinweise zur räumlichen Entfernung zwischen redundanten Rechenzentren*. 2006. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Hilfsmittel/Doku/RZ-Abstand.pdf?__blob=publicationFile (visited on 2019-05-02).

[88] C. Rohmann. *Elasticsearch, Logstash & Kibana*. 2016. URL: http://www.linux-magazin.de/ausgaben/2016/02/elk-stack (visited on 2019-05-02).

[102] The MITRE Corporation. *About CVE*. 2018. URL: https://cve.mitre.org/about/index.html (visited on 2019-05-02).

[103] The MITRE Corporation. *CVE and NVD Relationship*. 2018. URL: https://cve.mitre.org/about/cve_and_nvd_relationship.html (visited on 2019-05-02).