HAW HAMBURG

BACHELOR THESIS
Elina Juliane Eickstädt

Impacts of the Java Security Manager Removal: An Analysis of it Features, Limitations, and Prototypical Alternatives Implementations

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Elina Juliane Eickstädt

Impacts of the Java Security Manager Removal: An Analysis of it Features, Limitations, and Prototypical Alternatives Implementations

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Angewandte Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Stefan Sarstedt
Supervisor: Prof. Dr. Klaus Peter Kossakowski

Submitted on: 6. Dezember 2024

Elina Juliane Eickstädt

Title of Thesis

Impacts of the Java Security Manager Removal: An Analysis of it Features, Limitations, and Prototypical Alternatives Implementations

Keywords

Java Security, JVM, Client Side Security

Abstract

This thesis delivers a detailed impact analysis of the Java Security Manager's removal, focusing on alternative approaches and prototypical example development to close the resulting gap of relevant security features.

It provides a thorough examination of the Java Security Manager, delving into its layers of security, assessing its applicability while providing an overview of its practical and academic critique. This includes a detailed analysis of the irreplaceable components of the Security Manager, the relevancy in modern server applications and historically exploited attack vectors, that could be integrated into a streamlined version of the JVM.

An essential contribution is the evaluation and example implementation of certain features previously provided by the Java Security Manager.

Note of Thanks

Thanks to Alexander for answering all my questions regarding the use of the Security Manager in practice. And a big thanks to Simon for being my rubber duck whenenver the project went sideways.

Elina Juliane Eickstädt

Thema der Arbeit

Auswirkungen der Entfernung des Java Security Manager: Eine Analyse seiner Funktionen, Einschränkungen und prototypischen alternativen Implementierungen

Stichworte

Java, IT-Sicherheit, JVM

Kurzzusammenfassung

Diese Arbeit liefert eine detaillierte Analyse der Auswirkungen der Entfernung des Java Security Managers und konzentriert sich dabei auf alternative Ansätze und die Entwicklung prototypischer Beispiele, um die daraus resultierende Lücke relevanter Sicherheitsfunktionen zu schließen.
Sie bietet eine gründliche Untersuchung des Java Security Managers, die sich mit der Architektur und seine Anwendbarkeit befasst. Die Arbeit liefert eine detaillierte Analyse der wichtigsten Komponenten des Security Managers sowie eine Bewertung der Relevanz für modernen Serveranwendungen und der historisch ausgenutzten Angriffsvektoren, die in eine abgespeckte Version der JVM integriert werden könnten.
Ein wesentlicher Beitrag ist die Evaluierung und Beispielimplementierung bestimmter Funktionen, die bisher vom Java Security Manager bereitgestellt wurden.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Goal of This Thesis

The current software landscape is driven by dependencies, which allow developers to build on existing projects and reuse code that solves common problems. While using dependencies was once a complex endeavor, nowadays, dependency managers like Maven[32] for Java make it easy to use even small libraries. [20] With this ease, the risks introduced by unknown software are often overlooked. Installing and using libraries without testing, analyzing, or monitoring them assumes that the library developer properly maintains and secures their project. However, every library will have security vulnerabilities, be deprecated at some point, or even implement functionalities that are unknown to the user and can be exploited.[8]
A bandwidth of tools focuses on dependency management, ranging from detecting, monitoring, and patching. Mirhosseini and Parnin studied the use of dependency analysis tools by software developers. The authors quantitatively analyzed whether tools that supply automated pull requests lead to better update circles for dependencies. They found that even though automated pull requests led to better-update circles, developers did not perform almost two-thirds of the updates due to fear of breaking changes and not understanding the implications of changes or migration efforts. This quantitative study shows that even though automated tooling exists, developers often neglect updates due to a lack of knowledge of the used dependencies.
There are fewer concepts and strategies that focus on avoiding unwanted behavior of used dependencies. However, the unwanted behavior of dependencies has proven to be the origin of exploits like Log4Shell [29]. Ensuring that only the intended functionalities of a dependency is being used requires developers to thoroughly analyze and verify the functionalities of each dependency. This would be an extensive and complicated task, therefore the problem of running untrusted and unknown code remains. Additionally the general practice of running untrusted code at runtime is still a valid case. Every

application that allows users to run dynamic scripts for customized usage allow running untrusted code. While offering this kind of customization is a standard feature it is also a potential attack vector, making it a strong need for isolating those kind of scripts.[37] An early solution to the problem was put forward in 1993 by Wahbe et al., called sandboxing. The concept of a sandbox is based on encapsulating untrusted code and enforcing security policies via the sandbox. Instead of thoroughly analyzing used libraries, only the sandbox needs to be verified and trusted.

The Java 2 Platform Standard Edition (J2SE) implements the concept of sandboxing with the Security Manager. Allowing developers to block specific calls during runtime. Because of its complexity, it is hardly used by any project and was marked for removal and is deprecated in Java 17 [25]. Moreover, it will be blocked in Java 18 unless explicitly allowed by its user. In releases after Java 18, it will be fully degraded with limited to no functionalities. The problem of controlling the execution of critical functions on the run-time level remains.

In response, this thesis aims to develop prototypical alternatives to the Security Managers sandboxing capabilities controlling the execution of critical functions during runtime. The goal is to develop a prototype that delivers key functionalities like network, file level access, and process execution blocking while reducing the complexity by leaving fine grained object access control and thread model security out of scope. The prototype's design is based on an analysis of the Security Manager's critique and applicability. It analyzes practical and academic critique to derive usability requirements and better understand what led to its deprecation. Furthermore, it examines the Security Manager's applicability to the security requirements of modern server-side applications and historically exploited attack vectors to identify features that deliver an important layer of security that can not be covered otherwise.

1.2 Overview of the Java Security Architecture and existing Security tools

The Java Security Developer's Guide divides the Java Security Architecture into two areas: functionalities provided by the platform to run Java applications securely as well as security tools and services implemented in the Java programming language that allows the development of security-sensitive applications.[6]

Java was designed to be type-safe and offers automatic memory management, garbage collection, range checking on strings and arrays, and many other features that enable more secure coding. Furthermore, the compiler and the bytecode verifyer ensure that only legitimate bytecode can be executed in the Java Virtual Machine (JVM). With the JVM mediating access to critical system resources and the Java Security Manager as an option to check access in advance, restricting it to the bare minimum.

Since JDK 1.0, the Java Security Architecture has evolved significantly. Initially, it was based on a sandbox model, dividing code into trusted (local) and untrusted (remote) code. The sandbox provides a heavily restricted environment for running untrusted code, a necessity when shipping code as applets. With JDK 2.0, a first version of a more fine-grained access control model based on policies and permissions extending the sandbox model was introduced. This architecture was constantly developed with every JDK. Over time a, namespace-specific and highly granular permission model has been established.

1.2.1 Java Architecture

In order to understand the Java Security Architecture the general architecture of Java must be understood. It contains three main components: The Java Development Kit (JDK), the Java Run-Time Enviroment, and the Java Virtual Machine. While the JDK provides the compiler, standard libraries, and programs, needed to develop and build a Java application; the JRE links all runtime components together and initiates the Java Virtual Machine (JVM). The Java Virtual Machine is the key component of the Java Platform, enabling hardware and operating system independence. It performs the loading, verification, and execution and provides a runtime environment.[28] Thus, most security features are closely connected to the JVM.

Java Virtual Machine

The JVM is entirely detached from the Java programming language and solely operates as an platform independent execution engine that transforms Java byte code into executable code at runtime. Java byte code acts as an intermediate representation produced by the JDKs Java compiler allowing the Java Language to evolve over time implementing new language features. The following description is based on "The Java Virtual Machine Specification Java SE 22 Edition"[28].

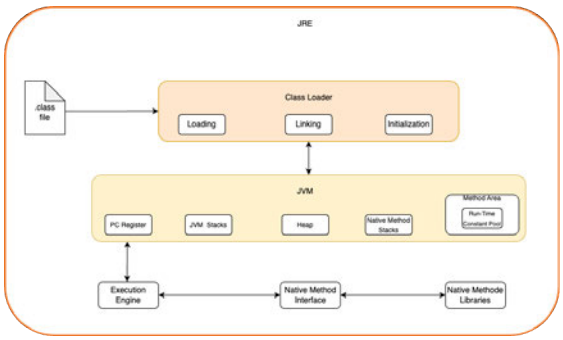

Figure 1.1: JVM Architecture adapted from the JVM Specification

Memory

The JVM has various areas of memory also called run-time areas. Some of them exist from start-up until exit of the JVM, others are created per thread and are destroyed subsequently.

Program Counter (PC) Register

The program counter in general stores the current execution instructions. As the JVM supports multi threading the PC register holds one PC per thread. One thread always executes the code of a single method.

Java Virtual Machine Stacks

In general, stacks store local variables and partial results. In the JVM, each thread has a private stack which is created when the thread is started, the stack is never directly manipulated and can be either of a fixed size or dynamically extended.

Heap

The memory of all class instances is allocated from the heap which is shared by all threads. The heap is created during the JVM start-up. The heap storage of objects is

automatically reclaimed by the *garbage collector*. If a program requires more heap than available a *OutOfMemoryError* is thrown.

Method Area
The Method Area has functionalities similar to the compiled code storage area in other programming languages. It stores the run-time constant pool, field and method data, and the code for methods and constructors.

Run-Time Constant Pool
The run-time constant pool functions like a symbol table with a much wider range of data. It is allocated from the Method area and created when the JVM creates a class or interface.

Native Method Stack
The Native Method Stack stores method information per thread. It is created when a new thread is created.

Class Loader
The class loader is responsible for loading class information, linking them, and initializing accordingly. Understanding class loaders and their hierarchy will be important to understand the Security Manager and the developed prototype.

The Java run-time contains different types of class loaders, which are divided into JVM and user supplied class loaders. They can be used to load classes dynamically from custom sources and allow for class isolation within the JVM runtime. Classloaders use the delegation model. Thus, each class loader instance has a parent loader, when loading a Class, one first delegates the class lookup to its parent before loading to prevent loading classes multiple times within one hierarchy.

The built-in JVM class loaders are the Bootstrap class loader,which loads the basic runtime classes provided by the JVM. It has no parent class loaders and is set to null. The platform class lloader which is responsible for loading all platform classes including J2SE platform APIs. The System/Application classl oaderL which loads from the application class path and loads everything else by delegating it to the platform class lloader. User defined class loader enables users to change the way the JVM loads and thus creates classes. This can be used to load classes from different sources, or change the behavior of certain classes. (This is were the JSM (Java Security Manager, Glossar add link) as well as the later presented prototypical solution comes in.)

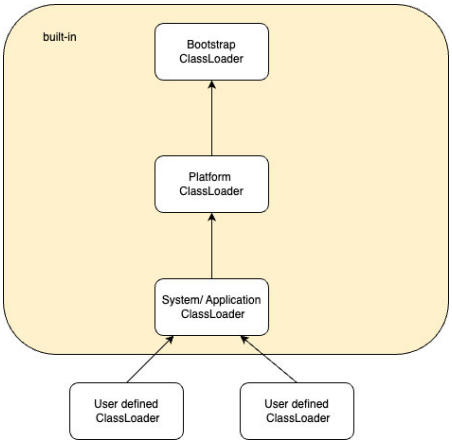

Figure 1.2: Class Loader Hirachy

Classloading

During the JVM startup, it creates an initial class using the bootstrap class loader. It starts by linking, initializing, and invoking the public class method void main(String[]). The invocation of this class drives all further execution. Subsequently, the loading, linking, and initialization of all other classes start in the same manner. In the loading process, the class loaders are used according to the hierarchy described above. During the loading process, the fully qualified class name, parent classes, and any relations to Class, Interfaces, and Method information are saved to the method area. An object of the type Class is also created and saved to the heap memory. In practice, those steps enable developers to get all information (class, parent name, method, and variable information) with the getClass() method.

Linking, Resolving, Initalizing

During the linking process, the bytecode verifier ensures that the .class file and, thus, the binary representation of a class or interface are structurally correct. If an error is detected, a VerifyError is thrown. After the loading process, the linking process starts. First, the bytecode verifier checks if the .class file is properly generated. If an error occurs, a run-time exception is thrown. Afterwards, the preparation process starts. During this process, memory for static class variables is allocated and initialized using default values. Until this step, the JVM only works with symbolic references, which contain just enough information to uniquely identify the class or interface. During the linking process, those references are resolved, which means identifying the class and determining the concrete values from the run-time pool. Following the linking comes the initialization process,

which executes the classes or the interface's initialization method. In this phase, all static variables are assigned to the defined values.

1.2.2 Security Architecture

The main goal of security architectures is to avoid security issues and limit damange in the case of an unexpected event, a software bug or unintended behavior. The Java Security Architecture is based on the sandbox model[6]. In earlier Java versions, only untrusted code was run in the sandbox, which provided a limited environment with no access to critical resources like network or file system access. The main driver for this separation was the widespread use of applets, a technology that downloaded code executing it on the local machine instead of running solely in the browser. With new browser technologies, applets became obsolete, and Java applications were deployed differently. Yet, the sandbox model is still used in the evolved Java Security Architecture; however, it no longer uses the concept of trusted and untrusted code, instead establishes a domain-specific and highly granular permission model. The following description is based on the current JDK version 22.0 and its security architecture specification. [6]

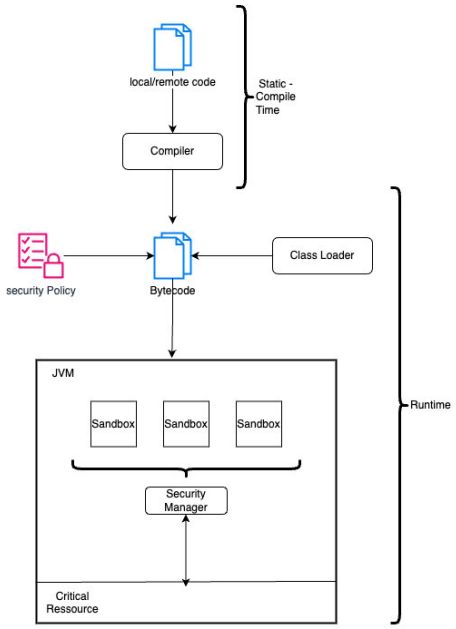

Figure 1.3: Java Security Architecture

1.2.3 Static Analysis

Compile time describes the process of translating a programm from source code to machine-readable code. In the case of the JVM, code is compiled into bytecode readable by the JVM. The compiler checks the source code for syntax, type-checking, and semantic errors. If no errors are detected, a bytecode (.class) file is generated; otherwise, a compile-time error is thrown. In addition to the Java compiler, third-party static analysis tools can be used to enforce code style or check for common bug patterns. Mainly, they are not used as a compiler extension but rather built into the build process using Gradle, Maven, or other CI/CD tools. For example, *Error Prone*[12] delivers a framework to spot common programing errors. It's an open-source project built and maintained by Google, maintaining a library of common programming mistakes like missing or wrong implementation of *ArrayToString* methods, serialization issues, or wrongfully configured tests that are never run. In addition to common programming mistakes, the use of dangerous or outdated APIs is a significant source of vulnerabilities in software development. *Forbidden APIs*[15] is a static analysis tool that addresses this issue by allowing developers to detect and prevent the usage of risky or deprecated APIs at compile time. Integrated with build tools such as Maven and Gradle, Forbidden APIs scans the bytecode of compiled class files to match against predefined or custom API lists, ensuring that insecure, outdated, or inefficient methods are not used in the codebase.

Runtime

After compilation, the JVM loads and executeds the generated bytecode. Security features enforced while runtime follow the architecture of the JVM and build up a sandbox by granular setting the permissions. The *Security Manager* is the instance that enforces and controls all set permissions and policies. In order to set a security context and thus define a sandbox environment, the class loader first defines the *code cource*. The *code cource* partitions the program into different components by security levels defined in the *Protection Domain Class*. The access to system resources (permissions) is defined via the *Permissions Class*. All other permissions to resources are defined with the *Policy Class*. With all permissions/policies defined, each access request to a resource is checked, managed, and enforced by the *Security Manager*.

1.3 Overview of this thesis structure

Chapter 2: Foundation of the Java Security Manager The goal of this chapter is to gain an understanding of the different capabilities of the Security Manager as well as the critique to derive key functionalities that a prototypical alternative should implement to function as a modern run-time level sandbox. This chapter describes the architecture of the Security Manager in detail and gives application examples regarding it's sandbox capabilities. Furthermore, it analyzes the integration complexity and critique put forward by the language developers and academic research. In order to understand architectural and conceptually problems of the Security Manager, it identifies issues that should be considered when designing an alternative solution.

Chapter 3: Security requirements in the J2SE This chapter aims to identify features of the Security Manager that defend against common attacks on server-side applications. It derives a baseline of vulnerabilities and attack paths from common security frameworks and analyzes the Security Manager's applicability. Furthermore, it examines how the Security Manager is used in practice by studying open-source projects interaction with the Security Manager and the impact of Log4Shell.

Chapter 4: Development of alternatives to the Java Security Manager This chapter describes the architecture of the alternative prototype using the thorough analysis of the previous chapters. It describes the prototypes intended scope, its architecture, and its functionalities. Furthermore, it evaluates tools like seccomp, ePBF and GraalVM that might be suited to cover functionalities that were previously covered by the Security Manager.

Chapter 5: Evaluation The Evaluation Chapter analyzes the developed prototype with regards to its practical effectiveness and theoretical applicability. It includes a detailed argumentative analysis comparing the prototype with the Security Manager.

Chapter 6: Conclusion and future work The final chapter concludes the analysis and implementation, summarizing key findings and contributions. It also outlines additional features and potential future research based on the prototype and findings of the thesis.

2 Foundation of the Java Security Manager

The goal of this chapter is to gain an understanding of the different capabilities of the Security Manager as well as its critique in order to derive key functionalities that a prototypical alternative should implement in order to function as a modern run-time level sandbox. This chapter describes the architecture of the Security Manager in detail and gives application examples regarding it's sandbox capabilities. Furthermore, it analyzes the integration complexity and critique put forward by the language developers and academic research. In order to understand architectural and conceptually problems of the Security Manager, it identifies issues that should be considered when designing an alternative solution.

2.1 Architecture and concepts

The Security Manager has existed since the release of JDK 1.0 [42]. It is part of the initial Java sandbox model and was built as a security mechanism for controlling untrusted code by restricting access to critical system resources. In Java 1.2, the Security Manager was redesigned to enforce the least privilege principle. It's architecture evolved constantly with every JDK version until the removal of Java applets in JDK version 9.0 [22], rendering the initial use case of the Security Manager obsolete. However, the security architecture of a domain-specific and highly granular permission model remained and was developed around the Security Manager. The following description of the architecture and permission model is based on the long-term support version of the JDK, JDK 21, security developers' guide. [6].

The Security Manager is based on the principle of least privilege enforced through protection domains, permissions, and policies. It is deeply embedded into the JVM and loads once the JVM is started. During runtime a *class loader* assigns a *code source* to

every loaded class, specifying the class' origin. Each *code source* is subsequently linked to a *protection domain* via its defined *permissions*. Permissions are user-defined via a provided .policy file or programmatically at runtime. Depending on the architecture, a policy file is created for the application and each dependency. The Security Manager's configuration is a whitelisting model; each permission is granted per method and lazily evaluated (when the method is called) during runtime. In practice, implementing the Security Manager requires not only testing the code but fully exercising it with all dependencies. Using the Security Manager can often expose unknown code paths that need to be considered to ensure that all permissions are correctly granted.

2.1.1 Enforcement of Policies and Permissions

When the Security Manager is enabled, the call of any method triggers its *checkPermission* method which ensures that the caller of the method has the correct permissions. In order to identify the caller and its permissions, the Security Manager checks if every method on the current call stack has permission to execute the called method. This mechanism ensures that less privileged domains cannot gain additional permission by calling or being called by a more privileged domain. However, the Security Manager offers the flexibility to avoid this behavior in order to enable developers to grant permission to a specific method or code base exclusively by *"cutting off"* the call stack and therefore, avoiding extensive permission checks by the Security Manager. This can be done using the *doPrivilidged* method which is part of the *java.security.AccessController API* .

2.1.2 Application examples

While the Security Manager can be used for a wide range of granular permissions, a key functionality is managing access to critical resources. Therefore, this application example uses only a subset of features to illustrate a possible Security Manager application:

- write/read access to files, which could lead to the corruption of files
- opening a network connection, which could be used to contact command and control servers
- calling System.exec and therefore being able to spawn a System process

- Calling System.exit. This is specific to the JVM and is called when unrecoverable errors occur. If System.exit() is called due to an error in dependencies, it terminates the entire JVM, thus quitting the whole application.

This example uses elasticsearch and methods used in the context of elasticsearch as it is the most prominent project that extensively uses the Security Manager in production. Elasticsearch is a widely adopted distributed search engine built on Apache Lucene [10]. The examples and figures are adapted from the blog post "elasticsearch - Securing A Search Engine While Maintaining Usability" [40].

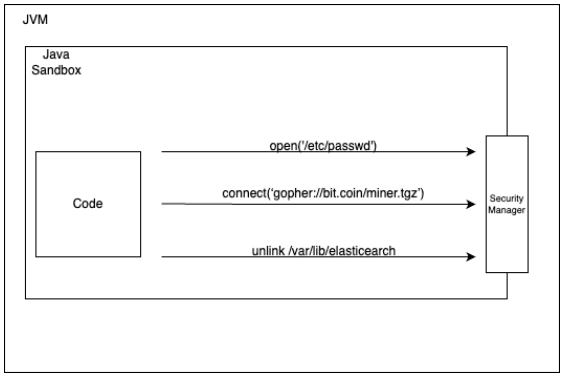

Figure 2.1: Java Sandbox

2.1 shows an example of different application methods that the Security Manager handles. Initially, the application tries to *open('/etc/passwd')* in the context of elasticsearch, there is never a need to open any files in */etc/* except its own configuration. For that reason, a policy file would only set permission for reading its *config file*; therefore, the Security Manager would block the call. Subsequently, the application tries to initiate a network connection to *gopher://bit.coin/miner.tgz*; while initiating a network connection is common to the use of elasticsearch due to its distributed nature, connecting to a potential bitcoin miner is not a valid action. Thus, permissions in the policy file should be set only for connecting to other elasticsearch nodes but not randomly initializing connections. (Note: In practice, elastic's netty jar is allowed to connect everywhere as Clusters/IP addresses change during runtime) The application calls *unlink /var/lib/elasticearch*, which would clean out all of elasticsearch data. Thus, permissions in the Security Manager would prevent that, too.

2.2 Security Manager integration complexity and critique

This section discusses the integration complexity and ongoing critique of the Security Manager. It discusses the elaborate critique put forward in the JDK Enhancement Proposal related to the Security Managers request for removal [25] and its sequential permanent disablement [24] as well as additional technical critique put forward in Java Blog posts regarding the Security Manager. Furthermore, it evaluates the scientific landscape regarding the Security Manager and elaborates on the current state of research.

2.2.1 Reasons for removal

The deprecation of the Security Manager was formally proposed in JEP 411 [25], with initial steps for its deprecation beginning in Java 17. As of Java 18, dynamic installation of the Security Manager is prohibited unless explicitly permitted. In conclusion, the request for removal argued that the Security Manager should be removed due to its lack of adoption, its complexity, and the high maintenance costs.

High Maintenance Costs

The Security Manager works with a highly granular permission model, which requires that the relevant permissions are not only granted to the called method but also the entire caller stacks leading to it's invocation. The request for removal argues that with the growth of the java.* and javax.* packages hundreds of permissions need to be checked and maintained throughout the JDK. Furthermore, every new package must be tested against the Security Manager for the access control model to work as intended, explicitly ensuring that additional permission cannot be gained by wrongful interaction of permissions.

Complexity

JEP 411 [25] criticizes the complex permission model, the implementation, and the Security Manager's poor performance. In order to take advantage of the Security Managers permission model, developers must carefully grant permission that an application or dependencies requires to run.
Due to the way permission checks of the Security Manager work, permissions must be

not only granted to the method that is executed but also to all the operations on the call stack when calling it; thus, in-depth knowledge of all functionalities that require permissions to security sensitive operation of the Java class library is needed.

While from a security perspective, in-depth knowledge of dependencies' interaction with security-sensitive operations of the Java class library might be desirable, the broad exploitation of such functionalities reveals that this isn't the case in reality.
The implementation of the Security Manager requires significant additional effort when writing new libraries as well when integrating dependencies into existing applications. A developer who wants to create a project that can be used with the Security Manager needs to document all permissions their code might need. The developer integrating the library into their application must grant all permissions accordingly. Granting permissions to the whole application that are only meant to be granted to a specific library would violate the principle of least privilege. This behavior can be avoided using the *java.security.AccessController API* specifically the method *doPrivilidged*. This enables a developer to grant permission to a specific method exclusively by cutting off the call stack and therefore avoiding escalated permission checks by the Security Manager. However, this approach assumes that library developers are willing to structure their applications to work with the Security Manager and that developers using these libraries are open to incorporating *doPriviliged* blocks as necessary. In reality, this is almost never the case, and developers frequently resort to granting universal permissions with *AllPermissions*, rendering the Security Manager as obsolete. Furthermore, the functionality of this API is not widely known among developers nor taught in an academic context. In addition to the Security manager's usability, performance plays a significant role for developers when considering implementing security tooling into their projects. JEP 411 argues that due to the complex access-control algorithm, the Security Manager can impose an unacceptable lack of performance. It further states that the lack of performance is a primary reason the Security Manager is disabled by default.

Lack of adoption

JEP 411 argues that the adoption of the Security Manager has been low. Only one project, elasticsearch, has fully implemented the Security Manager and ships with a costume version of the Security Manager in combination with a set of customized and highly specific permissions. Besides elasticsearch, only a few projects, e.g., Tomcat, NetBeans,

and Lucene, ship with either a policy file, thus using the standard or customized Security Manager. A paper published in 2014 strongly supports this argument [5]. The paper quantitatively examines the implementation of the Security Manager in open-source projects. For that purpose, a Dataset of 112 popular open-source Java applications was analyzed, and only 24 of those interacted with the Security Manager; the authors identified 12 additional repositories interacting with the Security Manager. However, further analysis showed that only a fraction of the identified projects used the Security Manager as intended. Which clearly shows that even when the Security Manager was a default feature, it was still rarely used. In addition to the mentioned reasons that led to a lack of adoption the lack of adjusting the permissions to modern development practices like supporting permissions for cloud storage, might play a significant role.

2.2.2 Systematic Analysis of academic critique of the sandbox

To evaluate the academic critique of the Security Manager, a systematic review of papers published in six top security and operating system conferences from 2014-2024 [16](see full list A.1) regarding containing analysis of the Security Manager was conducted. Two papers whose abstracts contained the term Java Sandbox or Security Manager were identified. The first identified paper [5] focuses on using the Security Manager as a sandboxing tool, quantitatively examining its implementation and use in open-source projects, and putting forward a proposal for improvements of the sandbox model based on the identified usability issues. The second identified paper presents an in-depth study of Java exploits [21] and analyzes the inner workings of Java exploits and the vulnerabilities used to exploit them. The paper systematically analyzes a set of 87 publicly available Java exploits to identify their root cause, targeting to propose countermeasures to address those causes. Both papers were published between 2014 and 2016 using data from the previous 10 years, 2004-2014, analyzing Java applications and vulnerabilities that were still focused on the applet model. Therefore, some of the critiques presented are no longer applicable to the current Java version. This section summarizes the critique and identifies valid points that can be used to develop an alternative prototype.

Security Manager as an instrument for running untrusted Code

The paper "Evaluating the Flexibility of the Java Sandbox" [5] analyzes the Security Manager in the context of the original use case of the Security Manager, running un-

trusted code in a sandbox environment. The paper hypothesizes that the Security Manager offers more flexibility than developers need or use, leading to increased complexity without offering additional functionalities. The authors conducted a quantitative study of open-source projects' interactions with the Security Manager. The data provided by the study proves the lack of adoption of the Security Manager; out of 112 open-source projects, only 24 interacted with it. As a result of the study, the authors found that half of the projects interacted with the Security Manager in a non-security way, either using it for unit testing or to cover functionalities that Java does not provide otherwise. Their study confirms that developers either do not know how to interact with the Security Manager at all or a deep misunderstandings regarding configuring policy files and permissions being present. The study's authors derived two rules that, according to their findings, could lead to a more secure use of the Security Manager. The authors used JVM Tool interfaces (JVMTI) to enforce the identified rules. JVMTI allows developers to create dynamic analysis tools called agents that can be used to intercept and react to events like the creation of classes and threads.

Privilege escalation rule The authors define privilege escalation as follows *"This rule is violated when the protection domain of a loaded class implies a permission that is not implied in the protection domain that loaded it."*[5] The proposed solution enforces privilege escalation by only allowing the JRE to load restricted-access packages, meaning packages that belong to the restricted-access sun package like *java.lang.reflect*. But prevent application classes from loading such packages. This prevents payloading attacks as they use application-level classes.

Security Manager rule The authors hypothesize that a benign use of the Security Manager would never change the Security Manager once it has been set up [1]. This rule is intended to catch so-called confusion attacks that target the manipulation of Security Managers' behavior. In order to enforce this behavior, they use an agent to monitor, read, and write in the Security Manager's system class field. It compares the settings with a shadow copy of the most recent security settings and checks for insecure settings.

The proposed changes were tested against Java 7 exploits from Metasploit 4.10.0 and proved to be a valid defense against the tested exploit. This strongly indicates that the Security Manager can be used as a line of defense against a broad set of vulnerabilities if configured correctly.

[1] " The Security Manager cannot be changed if a self-protecting Security Manager has been set by the application." [5]

Overall, the paper finds that a usable/impactful sandbox solution should be simplified, supply more insightful error messages, and deliver a more extensive documentation with examples of use. The suggested additional features of the Security Manager demonstrate that an adjusted Security Manager is an effective line of defense against common vulnerabilities. Generally the suggested permissions are not applicable to the current state of the security architecture. The kind of privilege escalation described is not possible anymore and manipulation of the Security Manager is not often appearing due to its lack of adoption and can be prevented if a custom Security Manager is set during bootstrapping.

Security Manager as Source of vulnerabilities

The paper "An In-Depth Study of More Than Ten Years of Java Exploitation"[21] analyzed and categorized 87 publicly available Java exploits. The exploits were reduced to 61 minimal code implements by removing exploits that were not reproducible and merging similar exploits. The authors derived a set of 9 weaknesses directly connected to the Java platform's features, like unauthorized use of restricted classes, arbitrary class loading, caller sensitivity, and *MethodHandles*. They conclude that many weaknesses result from the Java architecture, which should be redesigned according to the authors. They propose the redesign by introducing narrow permissions checks for sensitive functionalities in restricted classes, removing caller sensitivity, or replacing the functionalities with methods requiring extensive permission checks. The study shows that the Java Sandbox mechanism has been a source of vulnerabilities due to its complex nature, with several security mechanisms coming together to achieve the goal of sandboxing. In the current JDK Version (22), parts of the proposed redesign options have been implemented. Caller sensitivity still exists; however, features like *CallerSensitive* annotation and enhanced JVM checks for caller sensitivity have been introduced to ensure more secure use. Overall, this paper's results give valuable insights into the weaknesses of the Java Security Architecture. However, the vulnerabilities identified are highly specific to previous JDK versions. Thus, only the high-level issues identified (vulnerabilities due to complexity) are of interest to derive requirements for a high-level prototype.

2.2.3 Conclusion

The analysis of the Security Mangers critique shows that while there is a broad bandwidth of opinions from practitioners specifically from the Java Development team itself. There's a lack of research in the academic context in the last ten years potentially due to the lack of relevance of the Security Manager in practice. Overall, both practical and academic critique come to the conclusion that the Security Manager is to complex to implement and maintain. The evidence collected by the studies [5][21] conducted show that the extensive flexibility of the Security Manager leads to insecure or unintended use. And shows that there already was a lack of wide spread adoption when the Security Manager was still used in its original use case securing Java Applets. However, the proposals for the improvement of the Security Manager show that it can be a valuable tool reducing the impact of severe vulnerabilities. In conclusion, an alternative solution should significantly reduce its implementation and configuration complexity. This could potentially be achieved through reducing the flexibility and the functionalities of the Security Manager to a set of functionalities that only cover key features that enable sandbox capabilities. The next chapter will analyze which functionalities are crucial to using the Security Manager as line of defense against common vulnerabilities.

3 Applicability and impact of the Security Manager

This chapter aims to identify features of the Security Manager that defend against common attacks on server-side applications. It derives a baseline of vulnerabilities and attack paths from common security frameworks and analyzes the Security Manager's applicability. Furthermore, it examines how the Security Manager is used in practice by studying open-source projects interaction with the Security Manager and the impact of Log4Shell [29].

3.1 Common weaknesses of server-side applications

Identifying a baseline of security requirements for server-side applications is quite challenging, as the requirements differ depending on the application. While the failure of some applications might endanger human life other applications' failure might not be noticed at all. However the goal remains: a system should be build in a way that it remains dependable even if vulnerabilities are being exploited or programmatic errors have been made. Common security requirements are often focused around the information security triad (confidentiality, integrity, and availability). Complemented with awareness documents/frameworks like the OWASP Top Ten [36] or MITRE 25[34]. While those frameworks deliver insights into the most critical security risks of web applications, they don not focus on general impact mitigation during run-time. The Security Manager as well as the prototype that this thesis aims to develop acts as a layer of security at runtime. However the applicability of the Security Manager as run-time level defense against common attacks offers insights into general use cases. Therefore this thesis uses selected vulnerabilities of the MITRE [34][1] most dangerously rated software weaknesses and analyzes whether the Security Manager is applicable. The MITRE top 25 delivers a

[1]The vulnerabilities where selected according to their relevance to Java based server-side applications

list of the 25 most common vulnerabilities, which, according to MITRE , are potentially easy to find and can lead to data breaches, system takeovers, and rendering applications unusable.[34] For a better overview, they are grouped into three categories: access control, input validation/data processing, and memory safety.

Category	Weakness
Input Validation Data Processing	
	Improper Input Validation
	Unrestricted Upload of File with Dangerous Type
	Deserialization of Untrusted Data
	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
	Improper Control of Generation of Code ('Code Injection')
Access Control	
	Missing Authorization
	Improper Authentication
	Missing Authentication for Critical Function
	Use of Hard-Coded Credentials
	Incorrect Authorization
	Server-Side Request Forgery (SSRF)
Memory Safety	
	Improper Restriction of Operations within the Bounds of a Memory Buffer

Table 3.1: Identified relevant vulnerabilities

3.1.1 Applicability of the Security Manager

The Security Manager was intended to function as a layer of security that catches unintended vulnerabilities and malicious code, thus reducing the impact of vulnerabilities without the needed of extended security awareness by developers. However, it is also

suited to mitigate vulnerabilities before or while being exploited. Therefore, this evaluation evaluates both use cases of the Security Manager in order to analyze its applicability.

Access Control

The section of Access Control related weaknesses contains a bandwidth of weaknesses that can be exploited due to misconfiguration regarding user authorization and authentication when accessing an application or it systems' resources. In the context of the Security Manager access control is based around an application context, caller code origin, and methods. The Security Manager is not applicable to general authentication and authorization requirements. However, when it comes to accessing of critical functions like modifying sensitive data, system or administrative permissions or hardcoded credentials with the Security Manager unwanted behavior can be avoided. For example read access to '/etc/passwd' and thus access to hard coded credentials should never be allowed for server-side applications or through API calls and can be blocked by the Security Manager.

Input Validation/Data Processing

Improper input Validation

Improper input validation occurs when an application does not validate or incorrectly validates input that is processed by the application. This can lead to various ways of exploitation, such as SQL injection or privilege escalation attacks. The Security Manager is generally not designed for input validation as it does not provide any semantic or syntactic checks. In case of exploitation of improper input validation, it may minimize the impact of input that tries to read or write files or execute critical functionalities, which would be caught by the Security Manager's permission model.

Unrestricted Upload of File with Dangerous Type

This vulnerability describes an application that allows the upload of file types that are automatically processed or executed by the application. It can lead to exploits that perform arbitrary code execution. The Security Manager can be a suitable measure against those kinds of vulnerabilities. However, it depends on the specific permission model of the application that is being exploited. For example, if permissions are granted per dependency, an exploit in a dependency could be caught.

Deserialization of Untrusted Data

An application that insufficiently verifies deserialized data can be vulnerable to manipulation through serialization. The Security Manager cannot be used to avoid wrongful deserialization. However, it can be used to reduce the impact of exploitation depending on the permission model.

Improper Neutralization of Input

Improper Neutralization of input can lead to a range of attacks, including cross-site scripting, SQL injections, and OS command injections. The Security Manager is not applicable in this case as it has no built-in neutralization features. Furthermore, in most cases, exploitation is possible due to the architecture allowing specific commands, which would imply that they are permitted through the Security Manager as well. In case of managing SQL access control, the Security Manager comes with build-in features to build a set of permissions, however, it has to be assumed that the permissions would be granted accordingly.

Memory Safety

Exploits regarding Memory safety result in read and write operations outside the buffer's memory which can lead to overwriting critical data or control execution. Java is designed as a memory safe language with automatic memory management. Therefore the Security Manager was not build to protect against this kind of vulnerabilities.

Conclusion

The Security Manager's applicability is limited to managing resource permissions rather than directly preventing or resolving vulnerabilities. While effective in controlling access to critical functions and sensitive data, such as file access, it is not intended for general authentication, authorization, or input validation. Although it can reduce the impact of improper input handling by restricting resource access, its effectiveness against attacks using deserealization or exploiting improper input neutralization highly depends on the permission model and structure of the project.

3.2 Case Study: Log4Shell

This section describes the structure and methodology of the case study on the Security Managers as a defense line against attacks based on the example of Log4Shell [29]. The research question is: Does the Java Security Manager reduce the impact of vulnerable 3rd party dependencies like Log4Shell? The answer to the research question not only provides insights into the Security Managers features that form a robust layer of security against vulnerable dependencies but also offers practical insights for the architecture of an alternative solution.

3.2.1 CVE-2021-44228: Log4Shell Description and Proof of Concept

The Log4Shell vulnerability was publicly disclosed on December 10th, 2021. It was the first of a set of vulnerabilities in the popular logging library Log4j. Log4Shell was similarly widespread as vulnerabilities such as Heartblead[19], Rowhammer([26], and Spectre(2021)[27]. Log4Shell is described by the National Vulnerability Data Base as follows [29]: *"Apache Log4j2 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) JNDI features used in configuration, log messages, and parameters do not protect against attacker controlled LDAP and other JNDI related endpoints. An attacker who can control log messages or log message parameters can execute arbitrary code loaded from LDAP servers when message lookup substitution is enabled."*

```
logger.error("${jndi:ldap://127.0.0.1:" + port +"/matilda-poc}")
```

Listing 1: Log4Shell exploit example

3.2.2 Dataset

In the absence of an existing data set that maps the impact of Log4Shell on applications that use the Security Manager, I updated the data set from the Paper *"Evaluating the Flexibility of the Java Sandbox"* [5] which presents a data set on the usage of the Security Manager in Open-Source projects. The paper combined the relevant open-source projects from the Qualitas Corpus version [39] 20130901 with 12 additional projects that interact with the Security Manager found by the researchers on GitHub. As the Paper was published ten years ago, I updated it by removing inactive repositories, checking for

App Name	Category	Uses Log4J	Impacted by Log4Shell
Apache Batik	non-security context	No	No
Apache Lucene	Removed	Yes	Yes
Apache MyFaces	Removed	No	No
Apache Tomcat	Removed	No	No
Apache Xalan	Uses Sandboxing	No	No
AspectJ Java Extension	Uses Sanboxing	Yes	No
elasticsearch	Uses Sanboxing	Yes	No
IntelliJ IDE Community Edition	Uses Sandboxing	Yes	No
JRuby Ruby Interpreter	Uses Sandboxing	Yes	No
Netbeans IDE	Uses Sanboxing	Yes	No
Spring Boot	Removed	No	No

Table 3.2: Security Manager Log4Shell impact Dataset

any releases/commits in the last year and reduced it to applications that are hosted on GitHub to have one point of truth regarding Issues. I identified relevant applications by searching for the keywords *SecurityManager*, *System.setSecurityManager* in the applications source code. I added elasticsearch to the dataset as I identified it as the only open-source project that uses the Security Manager extensively. Additionally, I analyzed the repositories regarding their use of *Log4J* by searching the repositories for the keyword *Log4J*. I identified all repositories that were impacted by Log4Shell by searching through the repositories issues and their documentation.

3.2.3 Methodology

I performed a manual inspection of the applications in the dataset by grouping them by their interaction with the Security Manager. If an application comes with a policy file or a custom implementation of the Security Manager, I categorized them as *uses Sandboxing*. If Security Manager was removed but used until 2021, I categorized them as *removed*. When an application uses the Security Manager in a non-security context, I categorized them as *non-security context*. In addition to the analysis regarding the general interaction with the Security Manager, I identified which projects use Log4J and whether they were impacted by Log4Shell[29] in order to analyze the role of the Security Managers effectiveness in defending against vulnerabilities introduced through dependencies.

3.2.4 Study Result

In general, the data shows the lack of adoption of the Security Manager. In order to analyze the Security Managers effectiveness against supply chain attacks like Log4Shell, the usage of the Security Manager was mapped to whether the projects where impacted by Log4Shell. Out of the 10 repositories that interact with the Security Manager, six fall in the category "uses sandboxing" and three in the category of "removed". All nine projects used the Security Manager when Log4Shell appeared. However, the three projects that later removed the Security Manager never used Log4J and therefore were not impacted by Log4Shell. Out of the six projects using the Security Manager for sandboxing five used Log4J. However, none of them were impacted by Log4Shell[29]. That proposes that the Security Manager could play a role in the protection against Log4Shell. Out of those projects only elasticsearch officially stated that *"Supported versions of Elasticsearch (6.8.9+, 7.8+) used with recent versions of the JDK (JDK9+) are not susceptible to either remote code execution or information leakage. This is due to elasticsearch's usage of the Java Security Manager."*[11] This shows that the Security Manager is used in practice to defend against Remote Code Execution. Furthermore, the analysis of elasticsearch's use of the Security Manager could deliver insights into which features are valuable and should be implemented into an alternative prototype.

4 Development of alternatives to the Java Security Manager

This chapter describes the alternative prototypes architecture based on the thorough analysis of the previous chapters. It describes the prototypes intended scope, its architecture and its functionalities. Furthermore, it evaluates tools like seccomp, ePBF and GraalVM that might be suited to cover functionalities that were previously covered by the Security Manager.

4.1 Scope

The analysis of the academic and practical critique showed, that the lack of adoption of the Security Manager is primarily due to its maintenance and implementation complexity as well as its extensive flexibility. As a result of this analysis the prototypes scope is limited to sandboxing capabilities that allow controlling critical functionalities at runtime. The goal is to provide a solution that is easier to configure and to implement without the need of extensive testing or patching. Fine grained object access control and Java threading model security is generally out of scope. The analysis revealed that the Security Manager is theoretically suited to reduce the impact of common vulnerabilities but does not serve as first line of defense which is still the responsibility of the application developer. In turn, it implements a protection mechanism for impact minimization in the case of a programming error in the application itself or a 3rd party dependency (3.1.1). The case study found that elasticsearch's use of the Security Manager can be used to derive a set of functionalities that should be implemented in an alternative prototype since it has proved to protect the application from significant programming flaws even in 3rd party dependencies.

4.1.1 Use of the Security Manager in practice

Elasticsearch makes extensive use of the Security Manager using it as an effective measure against common vulnerabilities, while avoiding several problems that where identified in the previous chapters. The most prominent use case of the Security Manager is securing untrusted code in form of dynamic scripting that offers the user a great flexibility but also opens the application up to be vulnerable to executing arbitrary code. In the case of Log4Shell the use of the SocketPermission blocked the exploitation of the vulnerabilities Generally, Elasticsearch uses the following functions of the Security Manager [40]:

- FilePermission (read, write)
- SocketPermission (connect, listen, accept)
- block System.exit (via java.lang.Runtime)
- block System.exec (via java.lang.ProcessBuilder)

Primarily the Security Manager cannot be used if a project is not properly encapsulated packing all dependencies in one jar. With this practices, all permissions must be granted to all classes in the project or setting of *doPrivileged* in multiple places which can become complex and error prone fast. Elasticsearch circumvented this issue early on by building their own module architecture which encapsulates not just 3rd party dependencies but also its own domains like network access properly in order to allow granular permission setting for every dependency and internal module. This practice requires a lot of engineering effort, building an architecture around a security tool. Such an effort seems unlikely to be deployed in homegrown applications or even in the most available open source applications. In addition to the Security Manager elasticsearch uses seccomp due to the assumption that the Security Manager could fail. seccomp works as second layer of security blocking forking of any processes.

Summary

Elasticsearch uses the Security Manager to block System.exit, System.exec, Network Connection and read/write access to files outside of its home directory. Elasticsearch's Security model heavily relies on the Security Manager. Due to the Security Managers removal it can be assumed that they will build an alternative which will be tailored to their project and thus not usable for other projects. However, as this thesis shows there

is a valid need for an alternative that is usable for projects in general. The prototype delivers a sandboxing solution with the ability to block System.exit, System.exec, Network Connection and propose an architecture for controlling read/write access. This is due to the reason, that control over file access can be achieved by implementing a custom filesystem via the Filesystem API. This approach is well known, while the implementation of the other sandboxing capabilities needs a more complex solutions that make use of relatively new JDK APIs closely working on a lower JVM level.

4.1.2 Requirements

From the analysis of the previous chapter, the following requirements were derived:

1. Prototype should be configurable only at JVM startup
2. Prototype should be easy to implement, no changes in the projects using it needed
3. Prototype should implement Sandboxing features for: System.exit, System.exec and Network Connection
4. Prototype should be simple, focused and minimize flexibility to enforce safety and misconfiguration

In addition to the requirements derived from the analysis, security requirements must be considered as the solution operates on a deep technical level in the JVM and manipulates loaded classes at runtime. Therefore, the architecture and coding adhere to the Java Secure Coding guidelines [44], were ever possible. Not all guidelines are applicable since the prototype operates on a low technical level. Heavy use of ClassFile API [24]

4.2 Alternative Tools

Before developing the prototype from scratch, tools that provide sandbox capabilities for Java applications were evaluated.

4.2.1 Seccomp

The secure computing mode (seccomp) is a security feature of the Linux kernel that can be used to restrict system calls of processes interacting with the operating system. Even though seccomp provides sandboxing capabilities, it does not provide the level of granularity of the Security Manager as well as its platform independence. However, it can be used as a layer of the security for the case that the Security Manager or an alternative fails to block system calls equal to elasticsearch's use of seccomp [40]. Similar tools exist to implement such capabilities in macOS [43] and on Windows[1].

4.2.2 ePBF

The extended Berkley Packetfilter (ePBF) provides sandboxing capabilities that allow running application in a privileged context without the need of changing kernel source code or load kernel modules.[9] While eBPF allows running applications on a privileged level, it does not come with the functionalities to granular grant permissions to the sandboxed application but rather operates as an observability tool. It can be used to monitor system performance, application behavior as well as anomaly detection.

4.2.3 GraalVM

GraalVM is a virtual machine solution developed by Oracle. It is intended to improve application performance as well as compatibility with other programming languages.[17] While GraalVM generally provides sandboxing capabilities for scripting languages like Javascript, it does not provide capabilities to sandbox untrusted Java code at runtime.[18] Thus, it is not a suitable alternative for the security manager.

4.2.4 Summary

Existing tools that provide sandbox capabilities operate on the OS level and are therefore not suited as an alternative of the Security Manager. Additionally, there are partially depended on Operating Systems which contradicts the idea of running Java everywhere independent from the platform. Additionally, their deployment overhead makes it extremely difficult to package them with an application. However, they can be used to reinforce the Security Manager to catch malicious system calls if the Security Manager fails in addition to the applications build-in security measures.

4.3 Architecture

This description of the architecture refers to release 0.1 of the prototype [31]. The architectural design of the prototype is based on the so called interceptor pattern. Generally, architecture patterns describe successful solutions to common software problems. According to Avgeriou and Zdun [2] the use of architectural patterns offer *"well-established solutions to architectural problems, help to document the architectural design decisions, facilitate communication between stakeholders through a common vocabulary, and describe the quality attributes of a software system as forces."*
This is essential in order to ensure a solution that will be maintainable without too much complexity. Avgeriou and Zdun define interceptor pattern as a mechanism that allows augmenting of an existing service in response to incoming events by using reflection in order to retrieve necessary information for processing. Implementing sandboxing capabilities on JVM level works with this mechanism. Classes that are being executed need to be intercepted, the caller need to be checked for permission and be either blocked or forwarded accordingly. In the prototype, the logic of the interceptor is implemented via the *MathildaAgent*, the "service" that is augmented is the class that is currently loaded. (Figure 4.1)

The technical implementation of the interceptor pattern is based on java modularization. Modules provide two guarantees at runtime that are crucial for the maintainability and security of an application. First, reliable configuration which guarantees that only one instance of each class exists and that they are the same as those that are used during build-time. This is important due to the way the VM scans the classpath which can easily lead to confusing behavior when having similar jars in different versions on the

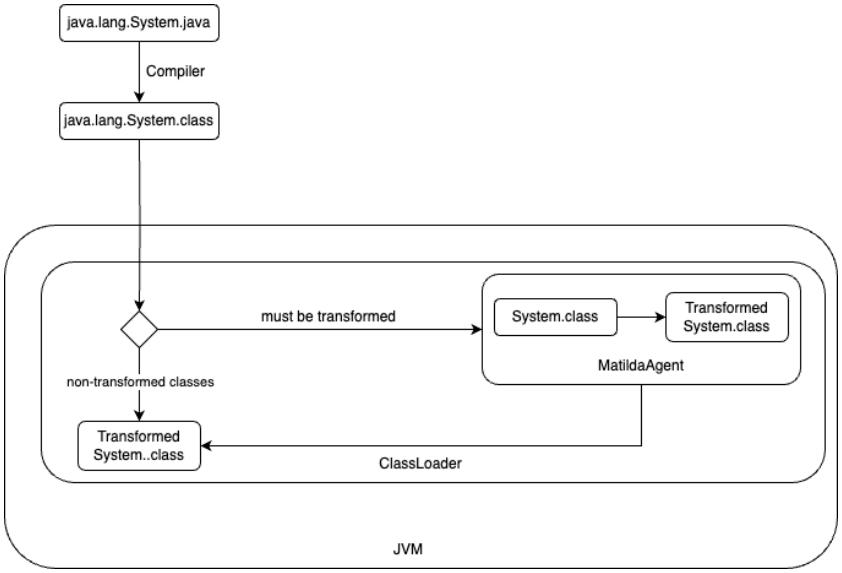

Figure 4.1: Matilda Agent - JVM Interaction

classpath as the VM uses the jar that occurs first during scanning. [38] The second and more security critical guarantee is strong encapsulation for classes and interfaces. Therefore code can only be accessed directly or through reflections if explicitly declared in the module definition. [38] This guarantees that classes and interfaces are only used via their defined API. Furthermore, using modules is generally recommended by the Java Secure Coding Guidelines [44]. In practices, most projects can be modularized easily, however for big old code bases it might be an issue. This is an disadvantage that is consciously accepted in order to get the advantages of security, maintainability, and reduced complexity.

The core module implements the interceptor and its underlying logic. The *bootstrap.module* provides access control configuration capabilities. These two modules are completely separated from one another and have no internal dependencies, this implements the concept of separation of concerns.

4.3.1 Core Module

The *matilda.core* module implements the Java Agent and the logic of the class transformation that is used to allow blocking of *System.exit*, *System.exec* and Network connection.

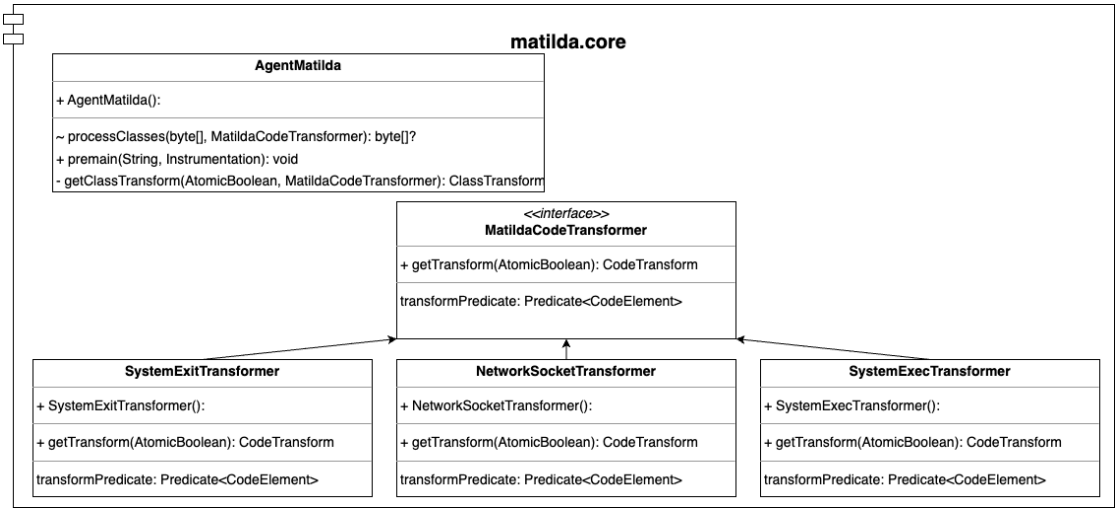

Figure 4.2: Class Diagram - Matilda Core Module

4.3.2 Agents

The prototype should provide sandboxing capabilities that allow critical functionalities to be controlled. It needs to work on a level between the application and the JVM to build this functionality. This is achieved by using the *java.lang.instrument* package, specifically the *Agent class*. A Java Agent allows a developer to either attach to the JVM before the application starts via the *premain* method or attaches the Agent after the JVM has started to run via the *Agent.main()* method. The prototype uses the first option as it targets to transform every class once it is loaded. The *premain* method takes the Agent as a string argument and allows to pass an instance of Instrumentation additionally. This instance provides an interface to low-level JVM functionalities. Control over the loaded classes is needed to control critical functionalities. In the case of the Security Manager, every class needs to check if the Security Manager was enabled and then run *SecurityManager.checkPermission()* before any restricted logic is executed, wit checks if the caller has permission to call the method or access certain internal functionality. This design leads to complexity as every class needs to be adjusted to use the Security

Manager. To prevent this flaw from being reintroduced, the prototype manipulates the target classes during class loading so they check the permission of their caller every time they are being called. Setting up the agent with the *premain* method allows accessing even JDK internal classes before the application is started and implements methods that manipulate classes while they are loaded. The *ClassFileTransformer* class which is triggered for every class being loaded provides the capability to process or even replace the loaded bytecode with an altered bytecode before any application code can utilize it.

4.3.3 Class Transformation

The prototype needs to be able to adjust the behavior of all classes being loaded in order to enforce permissions accordingly. With the *ClassFileTransformer* providing the bytecode representation of every class a tool to manipulate this code is needed. In addition, it needs to be decided when and how the permission check is being conducted and enforced.

Bytecode manipulation

A few libraries can be used to do bytecode manipulation, most prominently the ASM library. During early stages of the prototype development, ASM seemed to be the most promising way to do bytecode manipulation. Developing the first iteration, it became clear that using ASM and its visitor pattern quickly leads to a complexity that is hard to maintain. Furthermore, considering the robustness that is needed to create a security-related prototype is endangered by using non-jdk libraries, as those libraries are always behind the newest JDK release because they can only be updated to the newest version once the JDK is released. Furthermore, according to JEP 484 [23]
"[..]a significant problem for class-file libraries is that the class-file format is evolving more quickly than in the past, due to the six-month release cadence of the JDK. In recent years, the class-file format has evolved to support Java language features such as sealed classes and to expose JVM features such as dynamic constants and nestmates. This trend will continue with forthcoming features such as value classes and generic method specialization."
Therefore, the prototype uses the new JDK Class-File API, it is still a preview feature on JDK 23 but will be finalized with JDK 24. In addition to solving the above

mentioned issues, this decision also leads to improved security model. Only using JDK API makes the application independent from external dependencies and thus reduces the risk of supply chain attacks. The next step following the requirements would be the class transformation provided by the *ClassFileTransformer*. As stated above, different classes need to be transformed depending on what method needs to be blocked. For this purpose, the prototype offers a customized interface called *MatildaCodeTransformer* with two methods *getTransformPredicate* and *getTransform*, which are implemented in customized transformers for the methods that should be blocked.

Identifying classes that should be modified

Before manipulating a method, the right piece of code needs to be identified. Manipulating methods into the desired behavior can be achieved on different levels but also comes with some pitfalls.

Rewriting the invoking method

One possibility is to identify the invoking method (the caller of the API matilda tries do protect) and transform it. However, this leaves the actual executed method untouched, making the prototype vulnerable to attacks using the java reflection API. A malicious caller could still access code via the reflection API and execute methods that are supposed to be protected. Additionally, it would require a significant amount of byte code manipulation even if the majority of the calling code is never executed.

Rewriting the method itself

In order to rewrite the protected method itself, the method body needs to be identified and transformed. The prototype achieves this by using the *getTransformPredicate*. It matches *MethodModel* against characteristics specific to the method that needs to be blocked and can be used to test to identify the method that needs to be transformed. A *MethodModel* generally models a method's characteristics and can be traversed with a stream. The *getTransformPredicate* checks if the method has the correct parent class and matches against the method name and type.

Transformation

getTransform implements the customized transformer to transform the class (Example Code A.5). This transformer does decide the final behavior of the method; it manipulates the class, so it invokes the *MatildaAccessControll* when it is being called. It first checks via the *getTransformPredicate* if the method needs to be transformed. If not, it returns

```
public void exit(int status) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkExit(status);
        }
        Shutdown.exit(status);
    }
```

Listing 2: Runtime.exit Before Transformation

the *CodeElement*. Otherwise, it transforms and returns the transformed class accordingly. See A.4 for a bytecode example before and after the transformation.

```
public void exit(int status) {
        MatildaAccessControl.checkPermission("System.exit");
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkExit(status);
        }
        Shutdown.exit(status);
    }
```

Listing 3: Runtime.exit After Transformation

4.3.4 Bootstrap Module

The Bootstrap Module implements the access controller that checks each method call for its permissions and implements the configuration logic. The module is build as a separate jar which is added to the bootstrap class path. It is only loaded once called by the manipulated class. It is not only packaged separately in its own jar for security reasons but also to prevent class loading issues since *matilda* mainly manipulates JDK internal classes like *java.lang.Runtime*. For this to operate correctly *MatildaAccessControl* and all its dependencies must be available to the platform classloader hence it needs to be available on the bootstrap classpath.

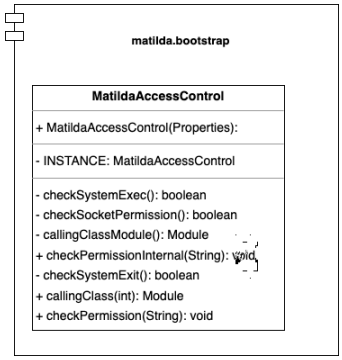

Figure 4.3: Class Diagram - Matilda Bootstrap Module

4.3.5 Access Control

The *MatildaAgent* transforms each class that should be protected such that that it calls *MatildaAccessControl* before the critical section of the code is accessed, which checks the permission and decides the final permission of the methods behavior. The *MatildaAccessControl* class uses a singleton pattern to ensure that only one instance of the access controller exists in the JVM. Singletons are instantiated exactly once and are often used to represent system components like file systems.[3]. Similar to the Security Manager the *MatildaAccessControl* uses a whitelisting approach. The access control (MatildaAccessControl) uses a module-based whitelisting approach. Every module that should be allowed to execute *System.exit*, *System.exec*, and Network connection needs to be whitelisted.

Configuration of Permissions

As stated in the scope the prototype should be configurable but not offer a brought flexibility like the Security Manager. In order to make the prototype configurable it uses system properties which can be easily passed via the command line. Those properties are represented as a simple key-value store (like a hash table), so their values can be accessed through their keys. In case of the *MatildaAccessControl* the key value pair follow the scheme:

matilda.<function>.allow:module <module name>[,module <module name>]

Listing 4: Property Format

The configuration will be checked for spelling and syntax errors by the *MatildaAccessControl* and throw an error accordingly. Afterwards the properties will be added to sets (*systemExitAllowPermissions*, *systemExecAllowPermissions*, *networkConnectAllowPermissions*) which will be used for permission checking.

Identification of Caller Class

In order to identify the caller and its permissions the Security Manager checks if every method on the current call stack has permission to execute the called method. This mechanism ensures that less privileged domains cannot gain additional permission by calling or being called by a more privileged domain. The prototype implements a similar approach. Before checking if the calling class belongs to a module with the necessary permission, the calling class needs to be identified. The call stack is a stack of program counters representing instructions that are being executed. In order to find the calling class the implementation optimized away the first few elements since they are known to be part of the prototypes architecture, however it is depended of the architecture of the solution. If an additional method is implemented into the prototype the frames to skip need to be adjusted

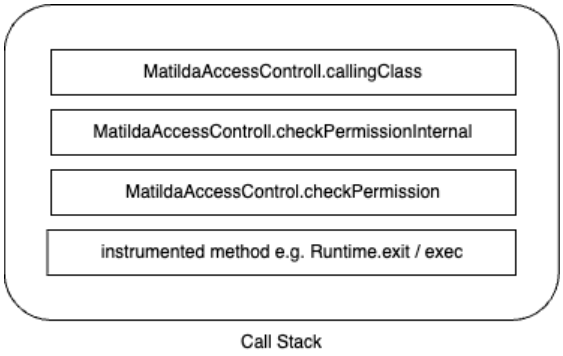

Figure 4.4: Call Stack Example

Enforcement of Permission

The permissions are enforced once the access control has identified the first class in the call stack that does not belong to the platform core module or in other words the first class that is not part of the java core class library and checks if the module (caller module)

it belongs to is in the allowed set of modules. If the caller module is not allowed to access this particular protected method a *RuntimeException* is thrown. Otherwise the code executes the actual code of the transformed method.

4.4 Testing

A number of unit and integration tests were implemented to ensure the correctness of the prototype and the proper interaction of the developed modules. Due to the architecture, tests were conducted per module. While unit testing could be achieved by a standard test setup, the integration tests needed additional helper classes to reach full test coverage.

4.4.1 Agent Matilda

The goal of testing the Agent Matilda module is to generally check if the transformation of classes works correctly by checking if it calls for *System.exit*, *System.exec*, and *Socket.connect* are correctly blocked. Per default the prototype blocks *System.exit*, *System.exec*, and *Socket.connect* which is expected behavior but lead to the issue that the gradle module will not be able to end the testing process once testing is finished due to its purposeful call to *System.exit*. Therefore the module *gradle.worker* requires specific permission to run *System.exit*. A.3 While this is an implementation detail of the testing used, it serves as a real-world integration test for the agents capabilities. Taking this configuration steps into account, unit test covering the case of successfully blocking method calls can be run. Additionally, this configuration is a test cased as well as it inherently tests if the access controller properly works by granting the the gradle worker sufficient permission. In addition to this unit-level testing, the capability of the module to protect against reflection attacks needs to be considered as well. In the past, the reflection API has been used to bypass Security Manager controls [41]; therefore, testing the prototype for its robustness against reflection is a significant concern. The use of reflection can be done by calling the method that should be blocked by using the *getClass* and *getMethod* methods of the reflection API and invoking it. In the first iteration, this test yielded a negative result as the prototype transformed just the invoking method, not the method body itself. Due to this test, the prototype was adjusted so that the method body is correctly rewritten and the application is protected against such attacks. Yet, the prototype also needs to verify that the the actual protected or transformed code is still working

correctly. In order to do that tests deploy a self-contained server that listens to a random port while the test with sufficient permission tries to communicate with it. This should trigger the blocking of the Socket.connect() method, which is called every time a socket is opened.

Integration testing

To test if the Matilda Agent and the Access Control module work together properly the case of setting permission for a specific module via the command line needs to be covered as well. For this reason a proxy object that belongs to another module is necessary whether the test can grant permission to execute specific calls. For this purpose the *ModuleProxy* class was created within the *matilda.core* module. It proxies the invocation of a method and returns object accordingly or the initialization of an object by invoking a certain method or a constructor within the context of the *matilda.core* module. In future iterations of this prototype, this should be moved out into its own module to keep any non-production code outside of the core module. As a workaround, this class will not be packaged into the jar files that are used to run the agent and therefore are only available to the testing infrastructure. In the integration tests the *ModuleProxy* class is used to call the methods protected by the Matilda Agent. In order for this test to be successful, the permissions for the module need to be set in accompanying gradle configuration A.3

4.4.2 Matilda Access Control

The testing of the Matilda Access Control covers standard unit tests. Testing both: successful blocking as well as successful setting of permissions.

4.5 Usage of the Prototype

In order to use the prototype, developers need to either download or build two jars. First the *matilda-agent.jar* and secondly the *matilda-bootstrap.jar* . The jars need to be passed via the command line when starting the application that should use the project. The *matilda-agent.jar* contains the matilda core module with the code relating to the agent it needs to be passed with the java agent parameter. In addition the *matilda.bootstrap.jar* needs to be added to the bootstrap path to avoid class loading conflicts. Furthermore

permission can be configured for *System.exit*, *System.exec* and *Socket.connect*. As the prototype makes use of a preview features, preview features need to be enable via the commandline.

4.5.1 Use of Prototype with tomcat

In order to demonstrate the easy integration of the prototype it was implemented into a tomcat server in order to run the server with the prototype the *matilda-agent.jar* and *matilda-bootstrap.jar* need to be added to the project and passed through the command line this can be done exporting the *CATALINA_OPTS* [1] with the following configuration.

```
export CATALINA_OPTS="--enable-preview
-javaagent:./path/matilda-agent-1.0-SNAPSHOT.jar
-Dmatilda.bootstrap.jar=./path/matilda-bootstrap-1.0-SNAPSHOT.jar"
```

Listing 5: Configuration tomcat

As well as adding the *enable-preview* to avoid any warnings related to the usage of the *ClassFile* API.

In order to test if blocking the critical functions works properly a test servlet was created that calls *System.exec.*(see code example A.2)

The server output shows that *System.exec* has been blocked successfully. Also note that this simple servlet does not have a module name which is the case if Java Code does not use modularization. Calls from an unnamed module will always be blocked by the agent.

[1] command line options that are used by the tomcat server and directly passed to the JVM in addition to it's own used parameters

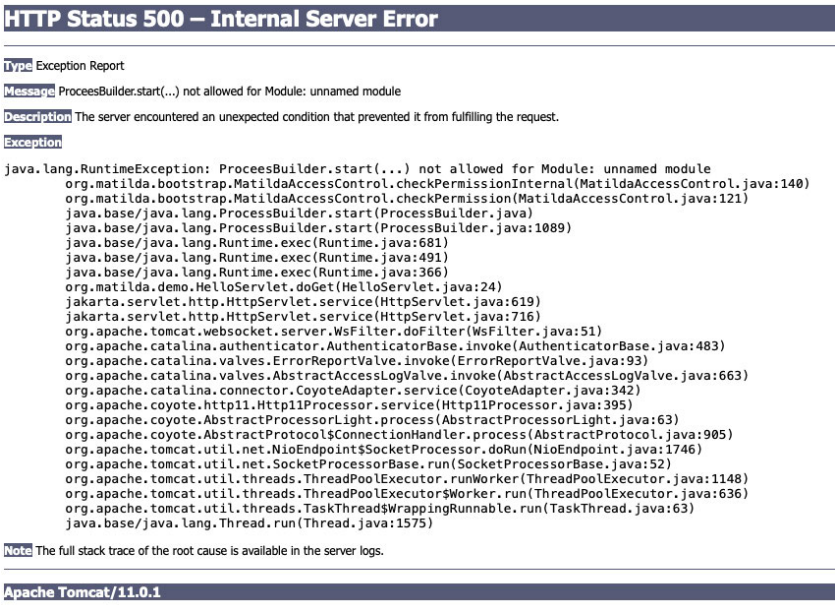

Figure 4.5: Server Error

4.6 File System Access - Architecture Proposal

While controls over the filesystem access like *read*, *write*, and *delete* could be achieved through the developed prototype, using the existing filesystem abstraction provided by the JDK is more suitable. In general, the usage of Java Agents and bytecode manipulation should be a last resort in order to achieve customization as it is provided by the prototype.

4.6.1 Filesystem Access and the JVM

The Java Virtual Machines (JVM) access to the filesystem is managed by a *FileSystemProvider*[14]. If no *FilesystemProvider* is defined, access to the filesystem will be provided by a system-default provider that creates a default filesystem.[4] The *FileSystems* API allows the creation of custom filesystems that can be used to manage the JVMs access to the underlying OS dependent file system. This API is better suited to control read, write, and delete actions as it already provides the necessary abstractions to manage such actions.

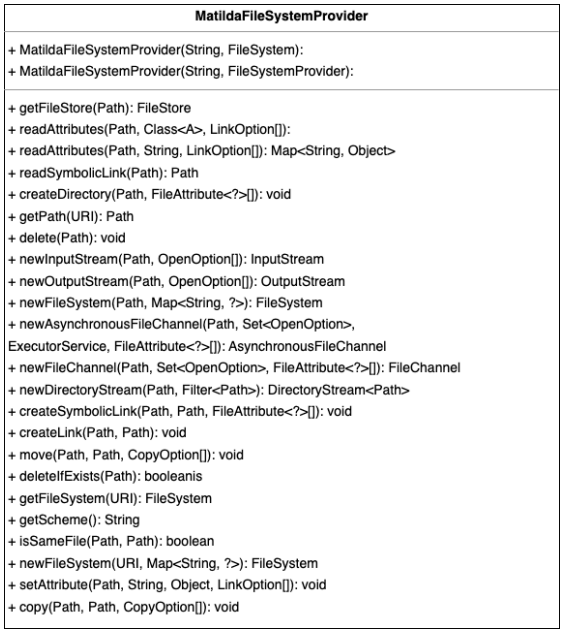

Figure 4.6: Class Diagram: MatildaFileSystemprovider

4.6.2 Architecture Proposal

In order to implement controls over read, write, and delete operations on the filesystem a customized *FilesystemProvider* is needed. In order to identify which method needs additional checks through the *MatildaAccessController*, a review of the *FileSystemProvider* class has been conducted, identifying all methods that contain permissions checks by the Security Manager. The Documentation of the *FilesSystemProvider* also gives insight whether a read, write, or delete check is needed. 4.6 shows a simplified class diagram with all methods that need to be overwritten. In order to properly implement a custom *FileSystemProvider* [14] the classes Path, *FileSystem* [4], *FileStore* [13], *DirectoryStream<Path>* [7] needs to be overwritten as well due to internal dependencies. Additionally the *MatildaAccessController* needs to be extended with.

5 Evaluation

5.1 Objectives and Scope

The objective of this evaluation is a detailed and nuanced analysis of the developed prototype with regards to its effectiveness as an defense tool that is similar to the Security Manager. The analysis is bisected into two distinct approaches: a theoretical analysis focusing on the prototypes applicability, its improvements compared to the Security Manager, and general considerations regarding performance implications. Followed by a practical analysis focusing on the prototype's effectiveness as a defense tool against common vulnerabilities. The scope of this evaluation encompasses a comprehensive study of the prototype's general effectiveness and applicability. Specifically by delving into the prototypes effectiveness of protecting against vulnerabilities similar to Log4Shell [29]. However, the evaluation will not extend to a broader study of effectiveness with regards to the wide field of vulnerabilities as long term studies in widely used projects would be necessary. Furthermore, the practical performance or usability is out of scope. This focused approach ensures that the evaluation remains aligned with the primary capabilities of the prototype and stays in scope with this thesis.

5.2 Theoretical Analysis

5.2.1 Theoretical Applicability

The prototype is partially based on the security manager's applicability analysis, and its theoretical applicability is very similar to that of the Security Manager.

Access Control

The prototype's access control capabilities are based on the modularization[1] of the application.

It is not applicable to general authentication and authorization requirements. Due to its missing implementation of controlling file system access it currently holds no capabilities to block critical functions like modifying sensitive data, system or administrative permissions or hard-coded credentials

Input Validation/Data Processing

The prototype is generally not designed for input validation and in its current version does not support the granular control of filesystem access. Therefore, its applicability to exploit which is based on insufficient input validation is limited. However, it comes with the capability of blocking any execution of spawning new process (*System.exe*), exiting the process (*System.exit*), opening a network connection (*Socket.connect*) and actively preventing attacks like *Log4Shell* by mitigating *Socket.connect*.

Memory Safety

Memory Safety can be disregarded as Java is designed as a memory safe language with automatic memory management. Therefore, protection against memory safety exploits is not in scope of the prototype.

Conclusion

The prototype's applicability is limited to managing resource permissions rather than directly preventing or resolving vulnerabilities. Compared to the Security Manager, it currently allows only managing a reduced set of resource permissions: (*System.exec*, *System.exit*, *Socket.connect*). However, the architecture allows the extension of these functionalities if needed. This comes with the trade-off of possibly creating a more complex application. Similar to the Security Manager, the prototype's effectiveness is

[1] Ron Pressler of the Java team at Oracle defines Java Modules as *" a set of packages that declares which of them form an API accessible to other modules and which are internal and encapsulated — similar to how a class defines the visibility of its members."*[38]

based on the permission model and structure of the project. However, the dependency is reduced by employing a less granular permission model and enforcing modularization which eliminates the issue of granting all permissions but rather adjusting to secure architecture guidelines.

5.2.2 Improvements

The Security Manager was deprecated due to its high maintenance costs and integration complexity which led to a lack of adoption. However, studies have shown that it can be used as an effective second line of defense. The prototype is already reduced in complexity due to its focus on sandboxing capabilities with a limited set of functionalities. The Security Manager's high maintenance cost is rooted in the extensive code path and additional checks that had to be implemented in order to use it. The prototype uses a different approach while the Security Manager needs adding in *checkPermission* methods into every class that should be used with the Security Manager. The prototype adds permission checks during class loading by manipulations the bytecode of the class directly. The only user interaction that is needed for usage is the attachment of the *MatildaAgent* and the loading of the *MatildaBootstrap.jar*. This not only reduces the maintenance costs but also the integration complexity. The only constraint is the proper use of modules which is a security best practice recommended by the Java Secure Coding Guideline [44].

Critique regarding use Java Agents and Bytecode manipulation

The prototype is based on Java Agents making heavy use of Bytecode manipulation using the *ClassFile* API while this is the only way to currently implement those kind of sandboxing capabilities aside of changing the JDK code itself. The usage of Java Agents and bytecode manipulation comes with imminent risk as it is present root kit like functionalities and the potential risk of missing implementation details of certain JVMs like method overrides not taken into consideration. Building a sandboxing solution that is based on a similar level of abstraction as building a filesystem would be preferable. However, this solution would come with the extensive need of overriding and adjusting code which could present a similar integration complexity and challenges as the Security Manager revealed in the past.

5.2.3 Performance

When evaluating security tools, its performance is often a major concern as it can impact the tools abilities to detect and mitigate attacks. However, this is primarily true when it comes to real-time threat detection tools. Even though the prototype serves as a security tool at runtime its performance is not linked to its effectiveness. Firstly, the prototype is intended as a second line of defense reducing the impact of errors that have been made during the development process. Secondly, the transformation of the class, which would be intentionally identified as a root of performance issues, happens during class loading which takes place before the application is even started. Therefore, it has no implication for the applications performance. However, the prototype allows run time checking of permissions which could lead to performance issues. Therefore, the permission checking itself needs to be considered for any performance evaluation. The frequency in which the permission checks are conducted is highly dependent of the specific application using the prototype. Only excessive calling of permission checks could lead to some performance implication. Compared to the time needed to execute *System.exe* or performing a network connect via *Socket.connect*, these checks merely add noise then anything else. Unless a method is considered *"hot"* and therefore considered eligible for JIT compilation permissions checks are not expected to contribute to the execution time of a method in any significant way. The methods that perform these checks today are not expected to be called repeatedly or in a context where those checks could impact performance.

5.3 Practical Analysis

To make this analysis comparable, it will test the prototype's effectiveness against the *Log4Shell* vulnerability, as the case study showed that the Security Manager can be used as an effective measure of this kind of attack. *Log4Shell* is still a representative vulnerability, according to Sonatype's latest "State of the Software Supply Chain" report, 13% of the Log4J downloads remain vulnerable to *Log4Shell*.[30].

5.3.1 Test setup

Initially, the testing was planned as part of the regular integration testing. However, testing *Log4Shell* within the unit test showed some class loading issues. Therefore, the pro-

totype repository contains a separate project that demonstrates the effective use against the *Log4Shell*. It is described by the National Vulnerability Data Base as follows: *"Apache Log4j2 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) JNDI features used in configuration, log messages, and parameters do not protect against attacker controlled LDAP and other JNDI related endpoints. An attacker who can control log messages or log message parameters can execute arbitrary code loaded from LDAP servers when message lookup substitution is enabled."*[29]
The first step of the exploit is to call a server through an LDAP instruction, which can be done via log messages, configuration, or parameter passing.

```
logger.error("${jndi:ldap://127.0.0.1:" + port +"/matilda-poc}")
```

Listing 6: Log4Shell expoloit example

If this call is successful, the exploit works, and the application is vulnerable. In order to prove that the prototype is effective against such an attack, it needs to be demonstrated that it successfully blocks such a call. The test setup consists of a simple socket server and a logger that uses a vulnerable version of *Log4J* (2.14.1)[29]. While the server listens for any input, the logger subsequently tries to inject an LDAP instruction to connect to the server via a *log4j* logging statement. If the call succeeds, this method will fail with an exception (see listing: 8); otherwise, it logs that the LDAP call was successfully blocked. The test setup offers to run the POC with and without the prototype in order to show that the application would be otherwise vulnerable. It can be run with *gradle run* or *gradle runNoAgent*.

```
> Task :runNoAgent FAILED
14:06:22.372 [Thread-0] ERROR org.matilda.ServerLog - start server on port
    57074 address: 127.0.0.1
14:06:22.372 [Thread-0] ERROR org.matilda.ServerLog - starting to listen
14:06:22.378 [Thread-0] ERROR org.matilda.ServerLog - accept
14:06:22.372 [main] ERROR org.matilda.ServerLog - ${jndi:ldap
    ://127.0.0.1:57074/matilda-poc}
Exception in thread "main" org.opentest4j.AssertionFailedError: LogForShell was
    not blocked by matilda ==> expected: <false> but was: <true>
```

Listing 7: Output Test without Matilda Agent

```
> Task :run
12:14:00.930 [Thread-0] ERROR org.matilda.ServerLog - start server on port
    49563 address: 127.0.0.1
12:14:00.931 [Thread-0] ERROR org.matilda.ServerLog - starting to listen
12:14:00.931 [main] ERROR org.matilda.ServerLog - ${jndi:ldap
    ://127.0.0.1:49563/matilda-poc}
12:14:00.948 [main] ERROR org.matilda.ServerLog - Matilda has successfully
    blocked log4shell
```

Listing 8: Output Test with Matilda Agent

5.3.2 Evaluation

Testing the prototype with the test setup showed that the prototype effectively prevents an application from being vulnerable to the *Log4Shell* vulnerability. The test setup allowed the application to run with and without the prototype. When the prototype is activated, malicious LDAP instructions are prevented from being executed by blocking any network connection. While the prototype was deactivated, the vulnerability was exploited. This underlines the prototype's efficiency in mitigating the attack, similar to capabilities of the the Security Manager.

6 Conclusion and future work

6.1 Conclusion

Through this thesis considerable progress has been achieved in creating alternative solutions for the security manager. Key achievements include, a systematic analysis of the Security Manager's critique both in a practical dimension and on an academic level to identify what features and improvements need to be implemented in an alternative prototype. Based on the analysis is the development of an alternative to the Security Manager that implements sandboxing capabilities at runtime. The thorough analysis of the Security Manager is crucial for several reasons. The case study conducted showed that the Security Manager can be used as an effective measure reducing the impact of common vulnerabilities. However, the analyzed critiqued shows that the adoption and secure use of the Security Manager is almost non-existent as well as the high maintenance and implementation costs which finally led to the deprecation of the Security Manager. The module and simple nature of the prototype delivers an easy to implement robust security solution that allows blocking of critical functionalities at runtime. This solution delivers another line of defense against the currently rising supply chain attacks. Furthermore, it exists and operates entirely on top of an existing application and is opaque to the application developer. It allows to secure and sandbox 3rd party applications without any code modification as long as relevant parts of the application use modularization. This allows individuals and organizations to deploy an extra layer of defense against unknown software bugs, supply chain attacks and malicious code maintainers.

6.2 Future Work

Building upon the analysis and the prototype developed in this thesis, several possibilities for further research and development have been identified:

1. **Evaluation of the impact of language Level Sandboxing** The thesis already delivers a thorough analysis of Java language-based sandboxing and a systematic analysis of the academic discourse. However, a full-picture analysis is needed to determine whether language-level sandboxing increases security protection against supply chain attacks and other common vulnerabilities. The implementation of a language-level sandbox by node.js proposes that there is a practical need for such concepts.[35]

2. **Validation Framework** A challenge identified both by the practical and academic analysis is the use of the Security Manager and the prototype to set the correct permissions and validate them to avoid false use of the solution. An additional feature to the prototype is a solution that first analyzes which configuration is needed and validates any insecure configuration once set.

3. **Implementation of Filesystem Access** The thesis puts forward an architectural proposal to implement sandbox capabilities for Filesystem access that could be implemented in another iteration of the prototype.

4. **Improvements Regarding Usability** The main reason for the lack of adoption of the Security Manager was its complexity accompanied by poor usability. In order to present an improved prototype, it should be scrutinized against usability best practices.

Bibliography

[1] *Active Process Limit.* https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-jobobject_basic_limit_informationredirectedfrom=MSDN. – Acessed: 25.11.2024

[2] Avgeriou, Paris ; Zdun, Uwe: Architectural Patterns Revisited - A Pattern Language. In: *EuroPLoP' 2005, Tenth European Conference on Pattern Languages of Programs, Irsee, Germany, July 6-10, 2005* Bd. 81, 01 2005, S. 431–470

[3] Bloch, Joshua: *Effective Java Programming Language Guide.* Addison-Wesley, 2001

[4] *FileSystem Class.* https://download.java.net/java/early_access/valhalla/docs/api/java.base/java/nio/file/FileSystem.html. – Accessed: 01.10.2024

[5] Coker, Zack ; Maass, Michael ; Ding, Tianyuan ; Le Goues, Claire ; Sunshine, Joshua: Evaluating the Flexibility of the Java Sandbox. In: *Proceedings of the 31st Annual Computer Security Applications Conference.* New York, NY, USA : Association for Computing Machinery, 2015 (ACSAC '15), S. 1–10. – ISBN 9781450336826

[6] *Java Platform, Standard Edition,Security Developer's Guide.* https://docs.oracle.com/en/java/javase/21/security/java-security-overview1.html. 2024. – Accessed: 07.06.2024)

[7] *DirectoryStream.* https://docs.oracle.com/javase/8/docs/api/java/nio/file/DirectoryStream.html. – Accessed: 01.10.2024

[8] Düsing, Johannes ; Hermann, Ben: Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories. In: *Digital Threats* 3 (2022), feb, Nr. 4

[9] *eBPF Documentation.* https://ebpf.io/what-is-ebpf/. 2024. – Accessed: 01.11.2024

[10] *The heart of the free and open Elastic Stack.* https://www.elastic.co/elasticsearch. 2024. – Accessed: 19.09.2024)

[11] *Subject: Apache Log4j2 Vulnerability - CVE-2021-44228, CVE-2021-45046, CVE-2021-45105, CVE-2021-44832 - ESA-2021-31.* https://discuss.elastic.co/t/apache-log4j2-remote-code-execution-rce-vulnerability-cve-2021-44228-esa-2021-31/291476?ultron=log4js-exploit&blade=announcement&hulk=email&mkt_tok=ODEzLU1BTS0zOTIAAAGBU8N1ZQBlVujs1JEM0Czxtag0JMHHD7tRaN5a--Tzhstpotlft0x7E5zeQYNYq9cQFrY37aUlhm5yI6BTYpaRzsuK4b_Z_GJUxTCux1bZF23HS-RI. 2021. – Accessed: 30.08.2024)

[12] *ErrorProne.* https://errorprone.info. – Accessed: 28.11.2024

[13] *FileSystem Class.* https://download.java.net/java/early_access/valhalla/docs/api/java.base/java/nio/file/FileStore.html. – Accessed: 01.10.2024

[14] *Class FileSystem Provider.* https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/nio/file/spi/FileSystemProvider.html. – Accessed: 29.11.2024

[15] *Policeman's Forbidden API checker.* https://github.com/policeman-tools/forbidden-apis/wiki. – Accessed: 28.11.2024

[16] *Google scholar top publications.* https://scholar.google.co.in/citationsview_op=top_venues&hl=en&vq=eng. 2024. – Accessed: 19.09.2024)

[17] *Introduction to GraalVM.* https://www.graalvm.org/latest/introduction/. 2024. – Accessed: 01.11.2024

[18] *GraalVM Security Guide.* https://www.graalvm.org/latest/security-guide/. 2024. – Accessed: 01.11.2024

[19] *CVE-2014-0160 Detail.* https://nvd.nist.gov/vuln/detail/cve-2014-0160. 2014. – Accessed: 11.11.2024

[20] Heinemann, Lars ; Deissenboeck, Florian ; Gleirscher, Mario ; Hummel, Benjamin ; Irlbeck, Maximilian: On the extent and nature of software reuse in open source Java projects. In: *Proceedings of the 12th International Conference on*

Top Productivity through Software Reuse. Berlin, Heidelberg : Springer-Verlag, 2011 (ICSR'11), S. 207–222. – ISBN 9783642213465

[21] Holzinger, Philipp ; Triller, Stefan ; Bartel, Alexandre ; Bodden, Eric: An In-Depth Study of More Than Ten Years of Java Exploitation. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA : Association for Computing Machinery, 2016 (CCS '16), S. 779–790. – ISBN 9781450341394

[22] *JDK 9 Release Notes - Deprecated APIs, Features, and Options*. https://www.oracle.com/java/technologies/javase/9-deprecated-features.html. 2017. – Accessed: 30.05.2024

[23] *JEP 484: Class-File API*. https://openjdk.org/jeps/484. 2024. – Accessed: 01.10.2024)

[24] *JEP 486: Permanently Disable the Security Manager*. https://openjdk.org/jeps/486. 2022. – Accessed: 30.08.2024)

[25] *JEP 411: Deprecate the Security Manager for Removal*. https://openjdk.org/jeps/411. 2022. – Accessed: 24.06.2024)

[26] Kim, Yoongu ; Daly, Ross ; Kim, Jeremie ; Fallin, Chris ; Lee, Ji H. ; Lee, Donghyuk ; Wilkerson, Chris ; Lai, Konrad ; Mutlu, Onur: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, S. 361–372

[27] Kocher, Paul ; Horn, Jann ; Fogh, Anders ; ; Genkin, Daniel ; Gruss, Daniel ; Haas, Werner ; Hamburg, Mike ; Lipp, Moritz ; Mangard, Stefan ; Prescher, Thomas ; Schwarz, Michael ; Yarom, Yuval: Spectre Attacks: Exploiting Speculative Execution. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019

[28] Lindholm, Tim ; Yellin, Frank ; Bracha, Gilad ; Buckley, Alex ; Smith, Daniel: *The Java® Virtual Machine Specification Java SE 22 Edition*. https://docs.oracle.com/javase/specs/jvms/se22/jvms22.pdf. 2024. – Accessed: 07.06.2024)

[29] *CVE-2021-44228 Detail*. https://nvd.nist.gov/vuln/detail/CVE-2021-44228. 2021. – Accessed: 30.08.2024)

[30] *State of the Software Supply Chain*. https://www.sonatype.com/state-of-the-software-supply-chain/introduction. 2024. – Accessed: 11.11.2024

[31] *Matilda Release 0.1*. https://github.com/khaleesicodes/Matilda/tree/branch_v0.1. – Accessed; 29.11.2024

[32] *Maven - Introduction*. https://maven.apache.org/what-is-maven.html. 2024. – Accessed: 30.09.2024

[33] Mirhosseini, Samim ; Parnin, Chris: Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, S. 84–94

[34] *2023 CWE Top 25 Most Dangerous Software Weaknesses*. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. 2023. – Accessed: 30.05.2024)

[35] *Process-based permissions*. https://nodejs.org/api/permissions.html#process-based-permissions. – Accessed: 20.11.2024

[36] *OWASP Top Ten*. https://owasp.org/www-project-top-ten/. – Accessed: 28.11.2024

[37] https://www.elastic.co/blog/painless-a-new-scripting-language. – Accessed: 29.11.2024

[38] Pressler, Ron: *What Modules Are About*. https://inside.java/2021/09/10/what-are-modules-about/. 2021. – Accessed: 28.11.2024

[39] *Qualitas Corpus*. http://qualitascorpus.com. – Accessed; 10.8.2024

[40] Reelsen, Alexander: *2020 Elasticsearch - Securing A Search Engine While Maintaining Usability*. https://spinscale.de/posts/2020-04-07-elasticsearch-securing-a-search-engine-while-maintaining-usability.htmll. 2020. – Accessed: 30.05.2024)

[41] *CVE-2013-0422*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0422. – Accessed: 25.10.2024

[42] *12 Revision History*. https://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-spec.doc12.html. 2002. – Accessed: 11.06.2024)

[43] *OSX Sandboxing Design.* https://www.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design/. – Acessed: 25.11.2024

[44] *Secure Coding Guidelines for Java SE.* https://www.oracle.com/java/technologies/javase/seccodeguide.html. 2023. – Accessed: 01.09.2024

[45] Wahbe, Robert ; Lucco, Steven ; Anderson, Thomas E. ; Graham, Susan L.: Efficient software-based fault isolation. In: *ACM SIGOPS Operating Systems Review* 27 (1993), dec, Nr. 5, S. 203–216. – URL https://doi.org/10.1145/173668.168635. – ISSN 0163-5980

A Appendix

A.1 Top Security and operating System conferences

1. IEEE Symposium on Security and Privacy (Oakland)
2. Usenix Security
3. ACM Conference on Computer and Communications Security (CCS)
4. ACM Symposium on Operating System Principles (SOSP)
5. Usenix Symposium on Operating System Design and Implementation (OSDI)
6. Annual Computer Security Applications Conference (ACSAC)

A.2 tomcat example

```
@WebServlet(name = "helloServlet", value = "/demo-servlet")
public class MatildaDemoServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
     ↪  HttpServletResponse response) throws IOException {
        //testing matilda
        Runtime.getRuntime().exec("echo demo");
    }
}
```

A.3 gradle test setup

```
test {
    jvmArgs += [
    "--enable-preview","-javaagent:${project.rootDir}/build/libs/matilda-agent
        -0.1.jar", "Dmatilda.bootstrap.jar=${project.rootDir}/build/libs/matilda-
        bootstrap-0.1.jar",
    // needs to be allowed so gradle worker can exit and negative Test cases work
    "-Dmatilda.runtime.exit.allow=module gradle.worker",
    "-Dmatilda.system.exec.allow=module matilda.core",
    "-Dmatilda.network.connect.allow=module matilda.core"]
    useJUnitPlatform()
    testLogging {
        exceptionFormat = 'full'
    }
}
```

Listing 9: gradle test setup

A.4 Bytecode Example

```
Load[OP=ALOAD_0, slot=0]
Load[OP=ALOAD_1, slot=1]
UnboundIntrinsicConstantInstruction[op=ICONST_0]
Invoke[OP=INVOKEVIRTUAL, m=java/net/Socket.connect(Ljava/net/SocketAddress
    ;I)V]
Return[OP=RETURN]
```

Listing 10: Class Bytecode before transformation

```
Invoke[OP=INVOKESTATIC, m=org/matilda/bootstrap/MatildaAccessControl.
    checkPermission(Ljava/lang/String;)V]
Load[OP=ALOAD_0, slot=0]
Load[OP=ALOAD_1, slot=1]
UnboundIntrinsicConstantInstruction[op=ICONST_0]
Invoke[OP=INVOKEVIRTUAL, m=java/net/Socket.connect(Ljava/net/SocketAddress
    ;I)V]
Return[OP=RETURN]
```

Listing 11: Class Bytecode after transformation

A.5 Transformer Example

```
@SuppressWarnings("preview")
public class NetworkSocketTransformer implements MatildaCodeTransformer{

private final AtomicBoolean hasRun = new AtomicBoolean(false);

@Override
public CodeTransform getTransform() {
    return (codeBuilder, codeElement) -> {
        if (!hasRun.getAndSet(true)) { // this must only be run / added once on
            top of the method
            var accessControl = ClassDesc.of("org.matilda.bootstrap.
                MatildaAccessControl");
            var methodTypeDesc = MethodTypeDesc.ofDescriptor("(Ljava/lang/
                String;)V");
            codeBuilder
                    // Needs to be hard coded in order to not run into classpath
                        issues when using MatildaAccessControl, as it is not
                        loaded yet
                    .ldc("Socket.connect")
                    .invokestatic(accessControl, "checkPermission",
                        methodTypeDesc)
                    .with(codeElement);
        } else {
            codeBuilder.with(codeElement);
        }
    };
}


@Override
public Predicate<MethodModel> getModelPredicate() {
    return methodElements -> {
        // Get class method is an element of
```

```
            String internalName = methodElements.parent().get().thisClass().
                asInternalName();
            // Check if its parent is the Socket Class
            return internalName.equals("java/net/Socket")
                    // Matches Methode
                    && "connect".equals(methodElements.methodName().stringValue
                        ())
                    // Matches Method Type
                    && "(Ljava/net/SocketAddress;)V".equals(methodElements.
                        methodType().stringValue());
        };
    }

}
```

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original