iJRASET
International Journal For Research in Applied Science and Engineering Technology

# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

**Volume:** 11 **Issue:** X **Month of publication:** October 2023

**DOI:** https://doi.org/10.22214/ijraset.2023.56166

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
*ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538*
*Volume 11 Issue X Oct 2023- Available at www.ijraset.com*

# JVM Security Mechanism

Dr. K. Bargavi[1], Bandaru Kranthi Kumar Varma[2], Addula Ajay Kumar[3], Chegoori Sai Kiran Mudiraj[4], Bollemani Vishal Nagaraj[5]
[1]*Professor, Teegala Krishna Reddy Engineering College, Hyderabad*
[1, 2, 3, 4, 5]*Student, Teegala Krishna Reddy Engineering College, Hyderabad*

***Abstract:*** ***This research paper explores the Java Virtual Machine's (JVM) security framework and investigates strategies to achieve comparable or enhanced security within the code. Examining core JVM security components such as classloaders, bytecode verification, and the security manager, we assess their limitations and capabilities in providing comprehensive security. We also delve into contemporary coding practices, including security libraries, secure coding principles, and industry-standard security frameworks, empowering developers to integrate security directly into their code. By comparing JVM security with in-code security strategies, this study aims to provide actionable insights for developers, security practitioners, and decision-makers, bridging the gap between runtime JVM security and proactive code-level security. The objective is to advocate for a more adaptable and robust approach to secure Java applications in today's evolving threat landscape.***
***Keywords: JVM, security, Java***

## I. INTRODUCTION

In an age where software permeates nearly every aspect of our lives, the security of software applications has become a matter of paramount concern. Among the many programming languages and platforms in use today, Java has stood the test of time as a stalwart choice for building robust and secure applications. A cornerstone of Java's security architecture is the Java Virtual Machine (JVM), renowned for its ability to provide a protective shield against malicious code execution. The JVM accomplishes this through a suite of security mechanisms, including classloaders, bytecode verification, and the security manager. However, the ever-evolving threat landscape demands a proactive and adaptable approach to security. This research embarks on a journey to not only dissect the security provisions offered by the JVM but also to explore the feasibility of replicating and, perhaps, surpassing these security measures within the code itself.

Our study begins by dissecting the inner workings of the JVM's security features, evaluating their effectiveness, and identifying their limitations. As we navigate through the intricacies of the JVM's security model, we will uncover scenarios where it may fall short in providing comprehensive protection. Concurrently, we will delve into contemporary coding practices, such as leveraging security libraries, adhering to secure coding principles, and adopting industry-standard security frameworks. These practices empower developers to embed security directly into their code, reducing reliance on the JVM's protective envelope.

By conducting this research, we aim to bridge the gap between runtime security, traditionally provided by the JVM, and proactive security measures at the source code level. This investigation will offer valuable insights for developers, security practitioners, and decision-makers, enabling them to make informed choices about security paradigms in the face of evolving threats. Ultimately, our goal is to advocate for a more resilient, adaptable, and comprehensive approach to securing Java applications in an era where software security is not merely a preference but a necessity.

## II. CODE WITH SECURITY

```
import java.security.Permission;

public class SecureJVMExample {

    public static void main(String[] args) {
        // Set a custom security manager to control access to sensitive resources
        System.setSecurityManager(new CustomSecurityManager());

        // Attempt to access a sensitive resource
        try {
```

```
        accessSensitiveResource();
    } catch (SecurityException e) {
        System.out.println("Security Exception: Access to sensitive resource denied.");
    }
}

static void accessSensitiveResource() {
    // Simulate an operation that requires special permissions
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkPermission(new CustomPermission("sensitiveResource"));
    }

    // Access the sensitive resource here
    System.out.println("Accessing the sensitive resource...");
}

static class CustomSecurityManager extends SecurityManager {
    @Override
    public void checkPermission(Permission perm) {
        if (perm instanceof CustomPermission && perm.getName().equals("sensitiveResource")) {
            // No need for isPermissionGranted(), you can implement custom logic here
            // to determine if the permission is granted
            // If not, throw a SecurityException as shown below:
            throw new SecurityException("Access to sensitive resource denied.");
        }
    }
}

static class CustomPermission extends Permission {
    private String name;

    CustomPermission(String name) {
        super(name);
        this.name = name;
    }

    @Override
    public boolean implies(Permission permission) {
        // Implement logic to determine if this permission implies the given permission
        // You can define more sophisticated logic here
        return false;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof CustomPermission) {
            return name.equals(((CustomPermission) obj).getName());
        }
        return false;
```

*ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538*
*Volume 11 Issue X Oct 2023- Available at www.ijraset.com*

```
        }

        @Override
        public int hashCode() {
           return name.hashCode();
        }

        @Override
        public String getActions() {
           return "";
        }
     }
  }
```

*1) Output*

Security exception: Access to sensitive resource denied.

*2) Explaination*

1. public class SecureJVMExample { ... }:

- This is the main class of the Java program, encapsulating the entire application. It contains the mainmethod, which serves as the entry point for the program.

2. static void accessSensitiveResource() { ... }:

- This method simulates an operation that requires special permissions to access a sensitive resource.
- It checks if a security manager (security) is set using System.getSecurityManager(). If one is set, it invokes security.checkPermission(new CustomPermission("sensitiveResource"))to check for the custom permission "sensitiveResource."
- If the permission is not granted, it throws a SecurityExceptionto indicate that access to the sensitive resource is denied.

4. static class CustomSecurityManager extends SecurityManager { ... }:

- CustomSecurityManageris a custom implementation of the Java SecurityManagerclass, which is responsible for controlling access to system resources.
- It overrides the checkPermission(Permission perm)method. This method is called by the JVM whenever a security check is required.
- In this example, CustomSecurityManagerchecks if the passed permission (perm) is an instance of CustomPermissionand if its name is "sensitiveResource." If these conditions are met, it calls isPermissionGranted()(a custom method you can implement) to determine whether the permission is granted. If not, it throws a SecurityExceptionto deny access.

5. static class CustomPermission extends Permission { ... }:

- CustomPermissionis a custom implementation of the Java Permissionclass, which is used to represent and manage permissions.
- It takes a nameas a parameter in its constructor and calls the superclass constructor with this name.
- The implies(Permission permission)method is left unimplemented in this example. It should typically contain custom logic to determine whether this permission implies the given permission.
- The equals(Object obj)method checks if two CustomPermissionobjects are equal by comparing their names.
- The hashCode()method returns a hash code based on the name of the permission.
- The getActions() method is left unimplemented in this example. It is used to specify actions associated with the permission.

These predefined methods and classes work together to create a custom security framework within the Java application, allowing you to define and enforce access control policies for sensitive resources based on your specific security requirements.

**III.CODE WITHOUT SECURITY**

```
import java.security.Permission;

public class SecureJVMExample {

    public static void main(String[] args) {
        // Set a custom security manager to control access to sensitive resources
        System.setSecurityManager(new CustomSecurityManager());

        // Attempt to access a sensitive resource
        try {
            accessSensitiveResource();
        } catch (SecurityException e) {
            System.out.println("Security Exception: Access to sensitive resource denied.");
        }
    }

    static void accessSensitiveResource() {
        // Simulate an operation that requires special permissions
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkPermission(new CustomPermission("sensitiveResource"));
        }

        // Access the sensitive resource here
        System.out.println("Accessing the sensitive resource...");
    }

    static class CustomSecurityManager extends SecurityManager {
        @Override
        public void checkPermission(Permission perm) {
            if (perm instanceof CustomPermission && perm.getName().equals("sensitiveResource")) {
                // No custom permission checking logic, simply allow the permission
            }
        }
    }

    static class CustomPermission extends Permission {
        private final String name;

        CustomPermission(String name) {
            super(name);
            this.name = name;
        }

        @Override
        public boolean implies(Permission permission) {
            if (permission instanceof CustomPermission) {
                // Implement logic to determine if this permission implies the given permission
                // You can define more sophisticated logic here if needed
```

```
            return this.name.equals(permission.getName());
        }
        return false;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof CustomPermission) {
            return name.equals(((CustomPermission) obj).getName());
        }
        return false;
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }

    @Override
    public String getActions() {
        return "";
    }
  }
}
```

1) Output

Accessing the sensitive resource. . .

## IV. DIFFERENCE BETWEEN SECURED AND UNSECURED CODE

The two code examples you provided demonstrate different approaches to security in a Java application:

*A. First Code (Custom Security Manager Approach):*

1) *Custom Security Manager:* This code uses a custom security manager CustomSecurityManager and custom permissions (CustomPermission) to enforce security checks.
2) *Security Logic:* Security checks are primarily implemented in the CustomSecurityManagerclass, which checks permissions and throws SecurityExceptionif necessary.
3) *Advantage:* It allows you to define custom security policies and checks specific to your application's needs.
4) *Disadvantage:* It relies on the JVM to execute these security checks, and the security logic is tightly coupled with the application. It may not be as versatile as other security mechanisms for complex scenarios.

*B. Second Code (Predefined Security Mechanisms Approach)*

1) *Predefined Security Mechanisms:* In this approach, the code relies on the Java Virtual Machine (JVM) to provide security through its built-in security features.
2) *Security Logic:* The JVM handles security features like classloaders, bytecode verification, and permissions. Your code does not explicitly implement custom security checks.
3) *Advantage:* It leverages established, proven security mechanisms provided by the JVM, offering a robust and standardized security framework.
4) *Disadvantage:* It may be less flexible for highly customized security requirements, and developers have limited control over security checks.

## V. APPLICATIONS OF SECURING CODE IN CODE ITSELF

1) Highly Customized Security Requirements: When your application has unique security needs that cannot be adequately addressed by standard JVM security mechanisms, custom security checks in code may be necessary.
2) Fine-Grained Access Control: Custom code-based security can allow for fine-grained access control, enabling you to implement specific security policies tailored to your application's requirements.

## VI. DISADVANTAGES OF SECURING CODE IN CODE ITSELF

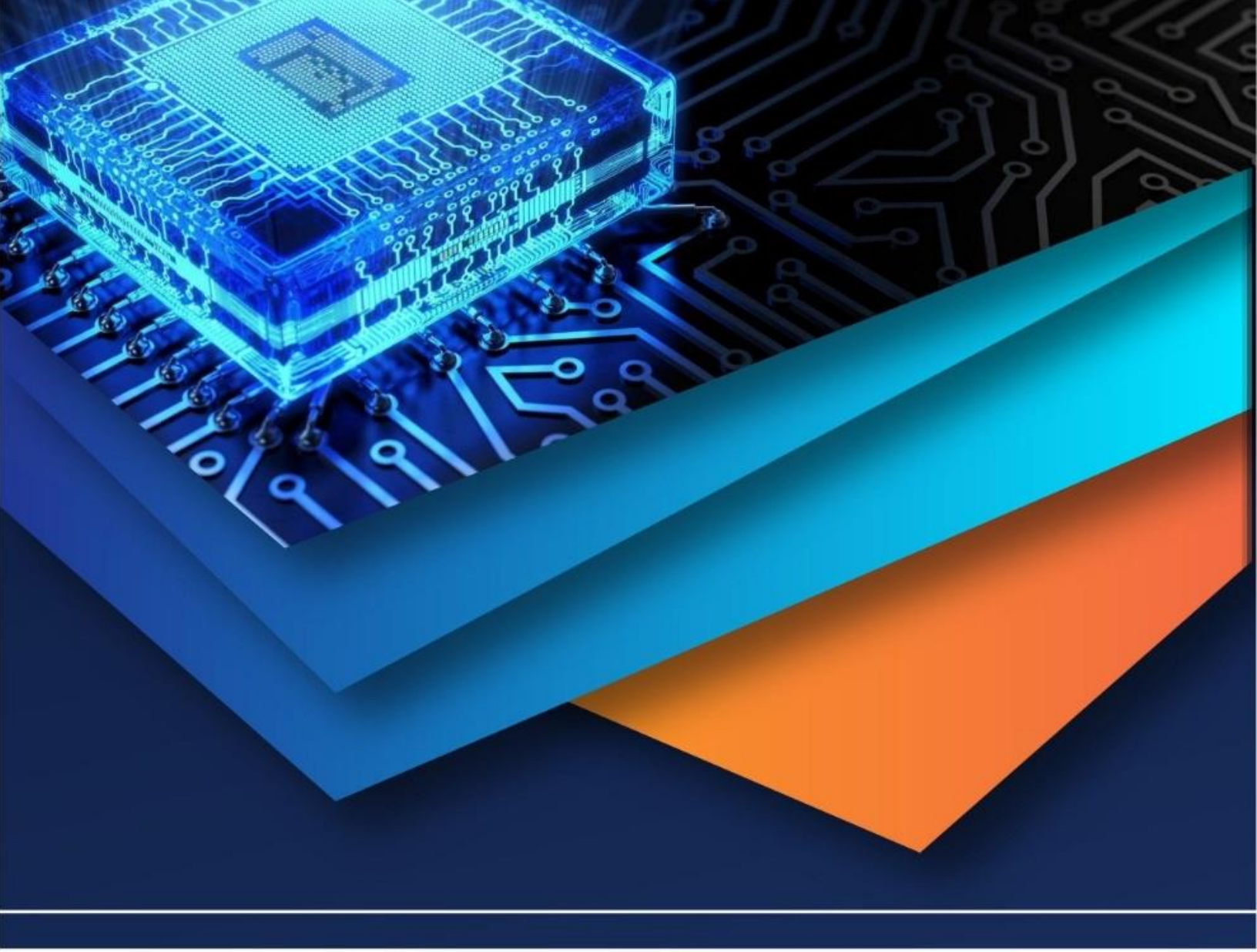
1) *Complexity:* Implementing custom security measures can add complexity to your code, making it harder to maintain and potentially introducing security vulnerabilities if not implemented correctly.
2) *Maintenance Overhead:* Custom security code requires ongoing maintenance, including updates for new security threats and vulnerabilities.
3) *Compatibility:* Custom security measures may not be compatible with all JVM implementations or may require adjustments when migrating to new JVM versions.
4) *Lack of Standardization:* Custom security implementations lack the standardization and auditing that come with built-in JVM security mechanisms.

## VII. CONCLUSION

In summary, securing code in code itself can be beneficial when you have specific security needs that aren't adequately met by standard JVM security features. However, it comes with added complexity and maintenance overhead. Leveraging JVM's built-in security mechanisms provides a standardized and proven approach to security but may be less flexible for highly customized security requirements. The choice between these approaches depends on your application's specific security needs and trade-offs

## REFERENCES

[1] Lin Deng, Bingyang Wei (2021). Securing Sensitive Data in Java Virtual Machines published in IEEE
[2] https://www.ijirt.org/master/publishedpaper/IJIRT142762_PAPER
[3] Reasoning about safety properties in a JVM-like environment by Philip W.L. Fongc institution or the target publication.
[4] Anindya Banerjee, David A. Naumann, Using access control for secure information flow in a Java-like language, in: Proceedings of the 16th IEEE Computer Security Foundations Workshop, CSFW'03, Pacific Grove, CA, USA, June 2003.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY