Breaking Encapsulation in Java-Methods, Security Implications and Mitigation Strategies

Mohammad Adeel[1], Shafiqul-Abidin[2], Hashir Sayed[3], Neha Yadav[4], Faisal Rais[4], Mohd. Izhar[4*]

[1] Department of CSE, Dr. Akhilesh Dass Gupta Instituite of Professional Studies
(Approved by AICTE and Affiliated to GGSIPU), Delhi, India
2 Department of Computer Science, Aligarh Muslim University, Uttar Pradesh, INDIA
[3] Department of AIML, Dr. Akhilesh Dass Gupta Instituite of Professional Studies
(Approved by AICTE and Affiliated to GGSIPU), Delhi, India
[4] Dr. Akhilesh Dass Gupta Instituite of Professional Studies
(Approved by AICTE and Affiliated to GGSIPU), Delhi, India

* Corresponding author: mohd.izhar.delhi@gmail.com
doi: https://doi.org/10.21467/proceedings.7.6.63

ABSTRACT

Encapsulation is a fundamental principle of Object-Oriented Programming (OOP) that ensures data security by restricting direct access to class members. In Java, private variables are designed to be accessible only within their defining class, safeguarding the internal state of an object. However, various techniques allow indirect access to these private variables, thereby weakening the guarantees provided by encapsulation. There are the methods through which private variables can be accessed indirectly, including reflection, inner classes, serialization, and method handles. This paper explores the various ways encapsulation can be bypassed in Java, analyze the implications of these techniques on software security and maintainability, and propose mitigation strategies, while discussing best practices to mitigate risks. Additionally, we present security recommendations to safeguard encapsulation in Java, including restricting reflection, using immutability, minimizing getters, and leveraging Java records and design patterns to reinforce data integrity and security.

Keywords: *Encapsulation, Object-Oriented Programming, data security*

I INTRODUCTION

Encapsulation is one of the core tenets of OOP, ensuring that an object's internal state remains hidden from external entities. Java enforces encapsulation through access modifiers, restricting access to class members. Among these, the private modifier offers the highest level of restriction, limiting access to the enclosing class. Despite Java's strict access control, private variables are not entirely secure. Several mechanisms allow them to be accessed indirectly, violating encapsulation principles. While these mechanisms are useful in scenarios such as debugging, testing, and framework development, they also introduce potential security risks when misused. This paper investigates different techniques for indirectly accessing private variables in Java, explores their security implications, and provides best practices to ensure safe and secure coding.

II LITERATURE REVIEW

2.1 Encapsulation and Private Variables

Encapsulation and private data protection remain core principles of secure software design. Java's architecture embeds multiple layers of security to preserve data integrity and enforce controlled access. Early distributed systems such as remote procedure calls [1,2] inspired Java's sandbox model to isolate processes and restrict unauthorized access. K. Bargavi [3] elaborated on JVM-level access control and bytecode verification, while L. Gong [4] offered a ten-year review of Java's evolving security landscape. L. Koved [5] detailed the architectural growth of Java's security manager, and A.Sterbenz [6] analyzed

Proceedings DOI: 10.21467/proceedings.7.6; Series: AIJR Proceedings; ISSN: 2582-3922; ISBN: 978-81-989164-9-5

limitations in policy enforcement. Foundational works such as Bloch [7] and Boyarsky and Selikoff [8] emphasize encapsulation through disciplined API design, defining how access modifiers (private, protected, public) safeguard internal class states. Practical guidelines by Bauer [9] and Silvestre and Ling [10] highlight design patterns ensuring persistence safety and modular access. M. Avalle, A. Pironti, R. Sisto and D. Pozza [11] and J. Liu [12] warned that Java's reflection API can break encapsulation by exposing hidden members, a challenge reinforced by Landman [13], who empirically demonstrated reflection's interference with static analysis. N. Meng [14] discussed how insecure reflection and unchecked access permissions create critical attack vectors, motivating automated detection methods such as Ghorbani 's Darcy tool [15] for resolving architectural inconsistencies. Z. Tang [16] further proposed an encapsulation analysis framework to detect poorly encapsulated classes.

Security in distributed and enterprise Java systems parallels concerns in other computing domains. Studies on secure network architectures [17], blockchain-enhanced IoT [18], and quantum cryptography in mobile cloud computing [19] illustrate comparable confidentiality and integrity needs. Research in wireless and sensor networks [20-23] aligns with Java's emphasis on authentication and freshness within communication protocols. Encapsulation's principles also extend into data-centric and neural computing contexts: Erfanian and Gerivany [24], Fanelli [25] and Abidin [22] address signal classification and model validation where privacy and modular parameter access are essential. Similarly, lightweight architectures in agriculture and medical analytics [1,2,11,14] demonstrate cross-domain applications of secure encapsulated systems for reliability and precision.

2.2Understanding Encapsulation

Encapsulation in Java restricts direct manipulation of internal object states and exposes behavior only through controlled interfaces. According to Bloch [7] and Boyarsky and Selikoff [8], it ensures data hiding, modularity, and maintainability. The four access levels—private, default, protected, and public—determine scope and visibility. Studies such as Bauer [9] and Koved [5] illustrate how layered access structures contribute to scalable and secure enterprise development. Reflection-related vulnerabilities [7,9,15,22] highlight that violating encapsulation can lead to privilege escalation, while automated verification tools [21] strengthen compliance. Broader security models developed for wireless [23] and IoT [22] based environments echo the same need for strict encapsulation boundaries to avoid unauthorized interference.

2.3 Private Variables in Java

Private variables should be inaccessible outside their defining class; however, Java features such as reflection [1, 13], and serialization may unintentionally expose them. Developers must apply principles from Effective Java [7] and secure coding research [14] to prevent privacy breaches. Static analysis frameworks [15,21] help identify weak encapsulation, while runtime enforcement [2, 16,17] ensures proper sandbox operation. Cross-disciplinary methods in neural computing [20,24] similarly benefit from encapsulated parameter control, enhancing reliability and reducing side-channel risks.

III INDIRECT ACCESS TO PRIVATE VARIABLES

3.1 Using Getter Methods

One of the most common ways to access private variables indirectly is through getter methods. While this method is intentional and widely used, it still allows external code to access private data.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

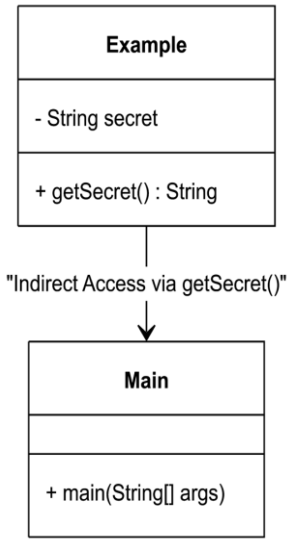

Figure 1: indirect Access via get

Example: Using a Getter Method

```
“class Example {
   private String secret = "Hidden";
   public String getSecret() {
      return secret;
   }
}
public class Main {
   public static void main(String[] args) {
      Example obj = new Example();
      System.out.println(obj.getSecret()); // Indirect access to private variable
   }
}”
```

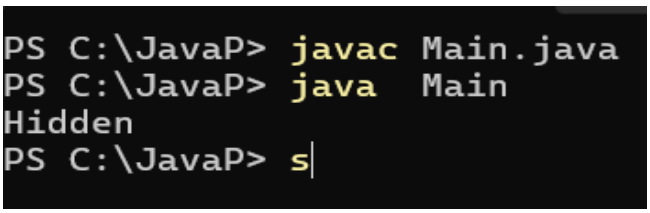

Figure 2: Output showing Indirectly Accessing of Private Variables

Implication: While this method does not break encapsulation explicitly, it exposes private data, making it susceptible to unintended modification.

Figure 1 illustrates the concept of indirect access to a private variable in Java through a public getter method, ensuring encapsulation. Figure 2 demonstrates the program output confirming controlled access without direct exposure of the private field.Table 1 summarises recommended security practices, including immutability, defensive copying, and restricted access mechanisms, to enhance data protection while allowing controlled access.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

TABLE 1: SECURITY RECOMMENDATIONS, ALLOWING CONTROLLED ACCESS

Method	Protection
Make variable final	Prevents modification after initialization
Return defensive copy	Prevents external modification of mutable objects
Restrict Reflection	Prevents unauthorized access via reflection
Avoid direct getters	Controls access using authorization checks
Use Encryption	Secures sensitive data from direct exposure
Use Java Records	Enforces immutability for encapsulation

3.2 Using Java Reflection API

Java's Reflection API allows dynamic access to class members at runtime, including private variables. By modifying access permissions, reflection bypasses encapsulation, allowing access to private fields.

Example: Accessing a Private Variable Using Reflection

```
"import java.lang.reflect.Field;
class Example {
   private String secret = "Hidden";
}
public class ReflectionAccess {
   public static void main(String[] args) throws Exception {
      Example obj = new Example();
      Field field = Example.class.getDeclaredField("secret");
      field.setAccessible(true); // Bypassing private access control
      System.out.println(field.get(obj)); // Prints "Hidden"
   }
}"
```

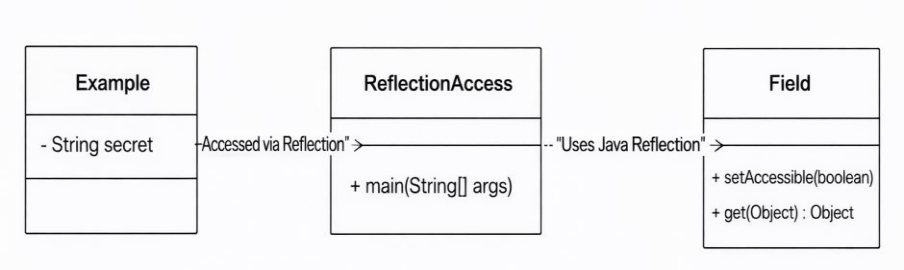

Figure 3: Accessed via reflection

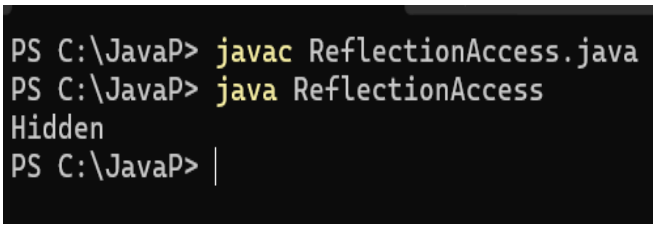

Figure 4: Output using Reflections

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

Implications:

- Reflection enables unauthorized access to private fields, leading to potential security vulnerabilities.
- It can break encapsulation, making the code difficult to maintain.
- Reflection-based access is computationally expensive, impacting performance.

Figure 3 depicts how Java Reflection can bypass encapsulation by accessing a private variable using the Field API. Figure 4 shows the successful runtime output, highlighting the security risk of reflection-based access. Table 2 presents effective countermeasures, including the Java Module System and custom JVM policies, to mitigate reflection attacks across different Java versions.

TABLE 2: SOLUTIONS TO PREVENT REFLECTION ATTACKS

Method	Protection	Applicable Versions
SecurityManager	Deprecated, but useful	Java 8 - 16
Module System	Recommended	Java 9+
Final + Transient Fields	Good practice	Java 7+
Unsafe Blocking	Can be overridden	Java 8+
Custom JVM Policy	Very effective	Java 11+
Overriding setAccessible	Prevents modifications	Java 8 - 16

3.3 Using Inner Classes

Inner classes in Java have unrestricted access to all private members of their enclosing class. This means that private variables can be accessed indirectly through inner class methods.

Example: Accessing a Private Variable via Inner Class

```
“class Outer {
    private String secret = "Hidden";
    class Inner {
        void reveal() {
            System.out.println(secret); // Accessing private variable
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer().new Inner();
        inner.reveal(); // Prints "Hidden"
    }
}
”
```

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

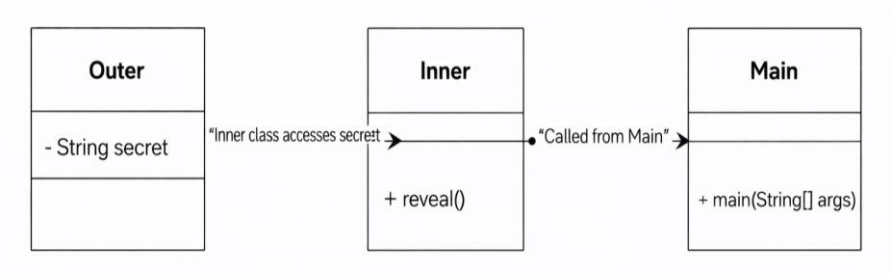

Figure 5: Inner Class accesses

```
PS C:\JavaP> javac Main1.java
PS C:\JavaP> java  Main1
via Inner-Class-show of Hidden
PS C:\JavaP>
```

Figure 6: Accessing a Private Variable via Inner Class

Implication:

- This method does not break Java rules but provides a loophole to access private data.
- The use of inner classes should be carefully considered in security-sensitive applications.

Figure 5 illustrates how a non-static inner class in Java can directly access private members of its outer class, demonstrating a language-level encapsulation exception. Figure 6 confirms this behaviour through program output, where the private variable is accessed via an inner class method. Table 3 summarises recommended design practices, including the use of static inner classes, immutability, and controlled access patterns, to prevent unintended exposure of private data.

TABLE 3: PREVENT INNER CLASSES FROM ACCESSING PRIVATE VARIABLES

Method	Protection	Applicable Version
Use final modifier	Simple & effective	Prevent modifications
Private getter with role check	Recommended	Restrict unauthorized access
Static inner class	Most secure	Prevent implicit access
Encryption	Protect sensitive data	Security-critical apps
Java Records	Immutable	Best for structured data
Factory Pattern	Restricts object creation	Enforce controlled access
Use final modifier	Simple & effective	Prevent modifications

3.4 Using Serialization and Deserialization

Java's serialization mechanism allows objects to be written to a stream and reconstructed later. Private fields are serialized and can be accessed indirectly after deserialization.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

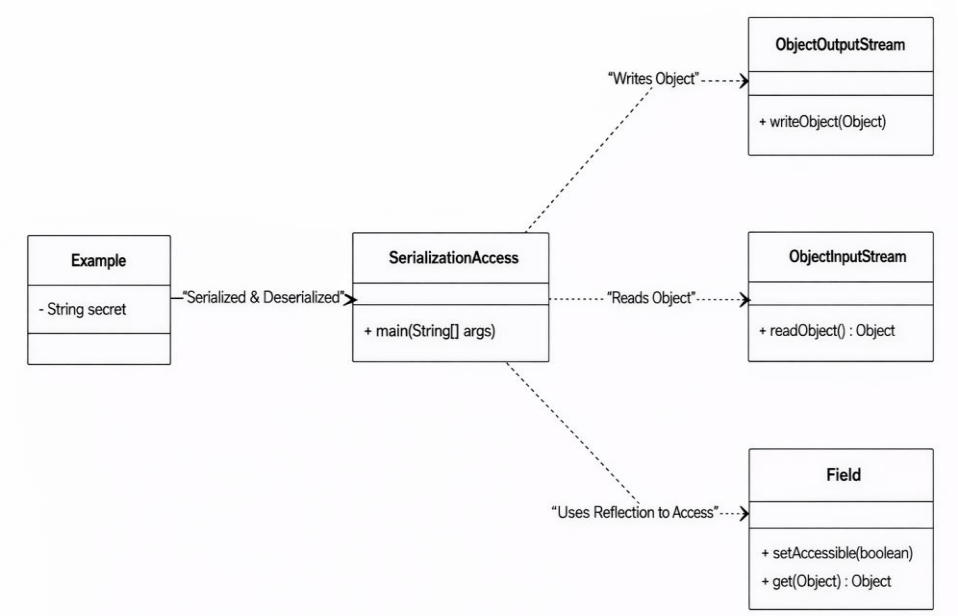

Figure 7: Solutions to Prevent Serialization-Based Attacks

```
PS C:\JavaP> javac SerializationAccess.java
PS C:\JavaP> java  SerializationAccess
Showing via Serialization: Hidden
PS C:\JavaP>
```

Figure 8: Results of Accessing via Serialization

Example: Accessing a Private Variable via Serialization

```
"import java.io.*;
import java.lang.reflect.Field;
class Example implements Serializable {
   private String secret = "Hidden";
}
public class SerializationAccess {
   public static void main(String[] args) throws Exception {
      Example obj = new Example();
      // Serialize
      ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("obj.ser"));
      oos.writeObject(obj);
      oos.close();
      // Deserialize
      ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.ser"));
      Example deserialized = (Example) ois.readObject();
      ois.close();
      Field field = Example.class.getDeclaredField("secret");
      field.setAccessible(true);
      System.out.println(field.get(deserialized)); // Prints "Hidden"
} }"
```

Implication:

- Serialized objects expose private fields unless explicitly marked as transient.
- Serialization-based attacks can extract sensitive data.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

Figure 7 illustrates how Java serialization and deserialization can expose private variables, potentially combined with reflection to bypass encapsulation. Figure 8 presents the program output confirming that sensitive data can be accessed during serialization if not properly secured. Table 4 highlights robust countermeasures—such as the transient keyword, custom serialization methods, and the serialization proxy pattern—to prevent serialization-based attacks.

TABLE 4: DEFENSIVE MECHANISMS AGAINST UNAUTHORIZED ACCESS VIA SERIALIZATION

Solution	Effectiveness	Best Use Case
Use transient keyword	Simple & effective	Prevents serialization
Override writeObject/readObject()	Secure	Controls serialization behavior
Use readResolve()	Safe instance	Blocks unauthorized access
Serialization Proxy Pattern	Best for security	Replaces sensitive data
Restrict Reflection Access	Prevents deep reflection	Java 9+
Use transient keyword	Simple & effective	Prevents serialization

3.5 Using Method Handles (Java 8+)

The MethodHandles API provides a way to manipulate private fields dynamically.

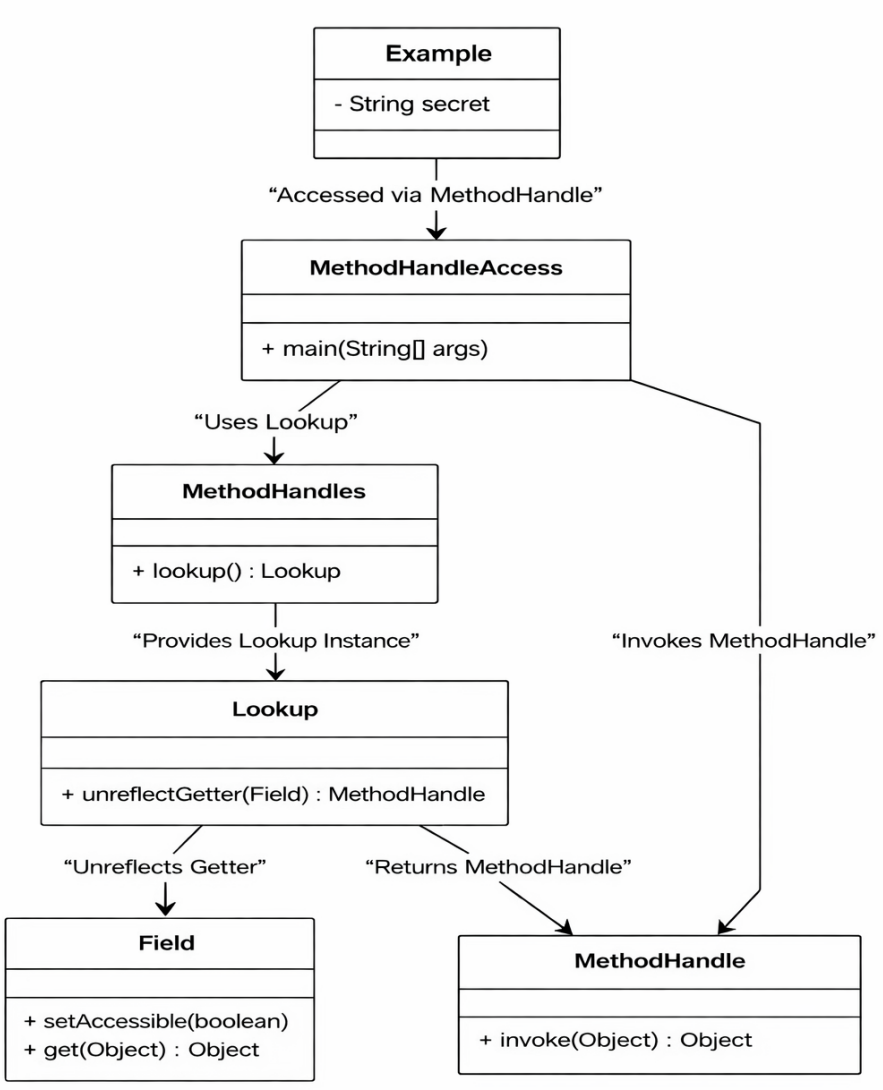

Figure 7: Method Handles to Access Private Variables

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

```
PS C:\JavaP> javac MethodHandleAccess.java
PS C:\JavaP> java MethodHandleAccess
shown through MethodHandleAccess:Hidden
PS C:\JavaP>
```

Figure 10: Result Using Method Handles to Access Private Variables

Figure 9 illustrates the use of Java *Method Handles* as a low-level mechanism to access private variables through controlled lookups, highlighting an advanced reflection-like capability. Figure 10 presents the execution output, confirming successful access to the private field via method handles. Together, these figures demonstrate that although method handles are designed for performance and controlled access, improper configuration can still lead to encapsulation bypass if security boundaries are not enforced.

Example: Using Method Handles to Access Private Variables

```
"import java.lang.invoke.*;
class Example {
   private String secret = "Hidden";
}
public class MethodHandleAccess {
   public static void main(String[] args) throws Throwable {
      Example obj = new Example();
      MethodHandles.Lookup lookup = MethodHandles.lookup();
      Field field = Example.class.getDeclaredField("secret");
      field.setAccessible(true);
      MethodHandle getter = lookup.unreflectGetter(field);
      System.out.println(getter.invoke(obj)); // Prints "Hidden"
   }
}"
```

Implication:

- This approach is efficient but can still violate encapsulation principles.
- Java's security policies may restrict its use in certain environments.

Recommended Ways to Prevent Method Handle Exploits

- Use Java Module System to prevent deep reflection.
- Declare sensitive variables final and transient.
- Use privateLookupIn(Class<?>) to limit MethodHandles scope.
- Block reflection and setAccessible(true) using a security policy.

IV. Security Implications and Best Practices

4.1 Security Risks of Indirect Access

Indirect access to private variables in Java can lead to several security vulnerabilities, undermining the principles of encapsulation and data integrity. Private data can be exposed through techniques like reflection, serialization, and method handles, potentially leading to data breaches. Bypassing encapsulation weakens code maintainability, making it harder to enforce security policies and refactor the code safely. Attackers can exploit reflection, deserialization, or inner classes to manipulate private fields, leading to security breaches such as privilege escalation and unauthorized modifications.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

4.2 Best Practices to Prevent Unauthorized Access

To mitigate the risks associated with indirect access to private variables, the following best practices should be adopted:

Restrict Reflection Usage: Avoid using reflection in production code unless absolutely necessary, as it bypasses Java's access controls.

Utilize Immutable Objects: Declare private fields as final to prevent unauthorized modifications and enforce immutability where applicable.

Limit Public Access to Sensitive Data: Avoid exposing sensitive information through public getter methods, and use controlled access mechanisms where needed.

Enforce Security Policies: Utilize Java Security Managers or custom security frameworks to restrict unauthorized access to critical operations.

Leverage Secure Design Patterns: Utilize Java Records, encapsulation-friendly design patterns (e.g., Builder Pattern), and security-aware coding practices to enhance data protection.

By following these best practices, developers can strengthen encapsulation, minimize security risks, and ensure robust access control in Java applications.

V. CONCLUSION

Java's encapsulation mechanism is fundamental to maintaining data security and integrity. However, indirect access techniques such as reflection, serialization, and method handles can expose private variables, leading to security vulnerabilities. These methods, while beneficial in controlled scenarios like debugging and testing, can be exploited to bypass access restrictions, compromising both security and maintainability. To mitigate these risks, developers must adopt best practices, including restricting reflection usage, enforcing immutability, limiting access to sensitive data, and leveraging secure design patterns. By implementing these measures, Java applications can uphold strong encapsulation, minimize unauthorized access, and enhance overall security.

ACKNOWLEDGMENT

The authors would like to express their gratitude to their organization for providing the necessary resources and support for the research presented in this paper.

CONFLICTS OF INTEREST

The authors declare no conflicts of interest and affirm that this work was conducted independently, upholding research integrity and transparency

REFERENCES

[1] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[2] A. Adeel, M. A. Khan, M. Sharif, F. Azam, J. H. Shah, and T. Umer, "Diagnosis and recognition of grape leaf diseases: An automated system based on a novel saliency approach and canonical correlation analysis based multiple features fusion," *Sustain. Comput.: Inform. Syst.*, vol. 24, 2019, Art. no. 100349, doi: 10.1016/j.suscom.2019.08.002.

[3] B. K. K. Varma, K. Bargavi, A. A. Kumar, C. S. K. Mudiraj, and B. V. Nagaraj, "JVM security mechanism," *Int. J. Res. Appl. Sci. Eng. Technol. (IJRASET)*, vol. 11, no. 5, pp. 123–130, May 2023.

[4] L. Gong, "Java security: A ten year retrospective," in *Proc. 2009 Annual Computer Security Applications Conference*, Honolulu, HI, USA, 2009, pp. 395–405, doi: 10.1109/ACSAC.2009.44.

[5] L. Koved, A. J. Nadalin, D. Neal, and T. Lawson, "The evolution of Java security," *IBM Syst. J.*, vol. 37, no. 3, pp. 349–364, 1998, doi: 10.1147/sj.373.0349.

[6] A. Sterbenz, "An evaluation of the Java security model," in *Proc. 12th Annual Computer Security Applications Conference*, San Diego, CA, USA, 1996, pp. 2–14, doi: 10.1109/CSAC.1996.569664.

[7] J. Bloch, *Effective Java*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2018.

[8] J. Boyarsky and S. Selikoff, “Methods and encapsulation,” in OCP Oracle Certified Professional Java SE 11 Programmer I Study Guide, Hoboken, NJ, USA: Wiley, 2020, ch. 6, pp. 247–295.

[9] C. Bauer, G. King, and G. Gregory, *Java Persistence with Hibernate*, 2nd ed. Greenwich, CT, USA: Manning, 2015.

[10] M. R. Silvestre and L. L. Ling, “Statistical evaluation of pruning methods applied in hidden neurons of the MLP neural network,” *IEEE Latin Am. Trans.*, vol. 4, no. 4, pp. 249–256, 2006, doi: 10.1109/TLA.2006.284156.

[11] M. Avalle, A. Pironti, R. Sisto and D. Pozza, "The Java SPI Framework for Security Protocol Implementation," *2011 Sixth International Conference on Availability, Reliability and Security*, Vienna, Austria, 2011, pp. 746-751, doi: 10.1109/ARES.2011.117.

[12] J. Liu, Y. Li, T. Tan, and J. Xue, “Reflection analysis for Java: Uncovering more reflective targets precisely,” in *Proc. 2017 IEEE 28th Int. Symp. Software Reliability Engineering (ISSRE)*, Toulouse, France, 2017, pp. 12–23, doi: 10.1109/ISSRE.2017.36.

[13] D. Landman, A. Serebrenik, and J. J. Vinju, “Challenges for static analysis of Java reflection—literature review and empirical study,” in *Proc. 2017 IEEE/ACM 39th Int. Conf. Software Engineering (ICSE)*, Buenos Aires, Argentina, 2017, pp. 507–518, doi: 10.1109/ICSE.2017.53.

[14] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, “Secure coding practices in Java: Challenges and vulnerabilities,” in *Proc. 2018 IEEE/ACM 40th Int. Conf. Software Engineering (ICSE)*, Gothenburg, Sweden, 2018, pp. 372–383, doi: 10.1145/3180155.3180201.

[15] N. Ghorbani, T. Singh, J. Garcia, and S. Malek, “Darcy: Automatic architectural inconsistency resolution in Java,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 6, pp. 1639–1657, Jun. 2024, doi: 10.1109/TSE.2024.3396433.

[16] Z. Tang, J. Zhai, B. Li, and J. Zhao, “Are your classes well-encapsulated? Encapsulation analysis for Java,” in *Proc. 2017 IEEE Int. Conf. Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic, 2017, pp. 208–215, doi: 10.1109/QRS.2017.31.

[17] A. Biradar, P. S. Akram, and S. Abidin, “Massive-MIMO wireless solutions in backhaul for the 5G networks,” *Wirel. Commun. Mobile Comput.*, 2022, doi: 10.1155/2022/3813610.

[18] M. Ayasha, S. G., and S. Abidin, “B-IoT (Block Chain – Internet of Things): A way to enhance IoT security via Block Chain against various possible attacks,” in *Proc. 2nd IEEE Int. Conf. Intelligent Computing Instrumentation and Control Technologies (ICICICT)*, 2019, pp. 1100–1104, doi: 10.1109/ICICICT.2019.8993144.

[19] S. Abidin, A. Swami, W. Ramirez-Asis, J. Alvarado-Tolentino, R. K. Maurya, and N. Hussain, “Quantum cryptography technique: a way to improve security challenges in mobile cloud computing (MCC),” *Mater. Today: Proc.*, 2012, pp. 508–514, doi: 10.1016/j.matpr.2012.05.110.

[20] S. Abidin, V. R. Vadi, and A. Rana, “On confidentiality, integrity, authenticity and freshness (CIAF) in WSN,” in *Proc. 4th Springer Int. Conf. Computer, Communication and Computational Sciences (IC4S)*, 2019, pp. 87–97, doi: 10.1007/978-3-030-23696-8_9.

[21] V. R. Vadi, N. Kumar, and S. Abidin, “Classifying time-bound hierarchical key agreement schemes,” in *Proc. 4th Springer Int. Conf. Computer, Communication and Computational Sciences (IC4S)*, 2019, pp. 111–119, doi: 10.1007/978-3-030-23696-8_11.

[22] V. R. Vadi, S. Abidin, A. Khan, and M. Izhar, “Enhanced Elman spike neural network fostered blockchain framework espoused intrusion detection for securing Internet of Things network,” *Trans. Emerg. Telecommun. Technol.*, 2022, doi: 10.1002/ett.4634.

[23] Y. Sucharitha, S. Vinothkumar, V. R. Vadi, S. Abidin, and N. Kumar, “Wireless communication without the need for pre-shared secrets is consummate via the use of spread spectrum technology,” *J. Nucl. Sci. Power Gener. Technol.*, vol. 10, no. 2, pp. 119–124, 2021, doi: 10.37532/jnspt.2021.10(2).119.

[24] A. Erfanian and M. Gerivany, “EEG signals can be used to detect the voluntary hand movements by using an enhanced resource-allocating neural network,” in *Proc. 23rd Annu. IEEE Eng. Med. Biol. Soc. Conf. (EMBC)*, 2001, vol. 1, pp. 721–724, doi: 10.1109/IEMBS.2001.1019042.

[25] S. Fanelli, P. Paparo, and M. Protasi, “Improving performances of Battiti-Shanno’s quasi-Newtonian algorithms for learning in feed-forward neural networks,” in *Proc. Aust. N. Z. Intell. Inf. Syst. Conf. (ANZIIS ’94)*, 1994, pp. 115–119.