Secure Coding 26

Chapter Topics

- An overview of application security
- Identifying security threat categories
- Understanding denial-of-service (DoS) attacks and their mitigation
- Recognizing sensitive information leakages, and applying value obfuscation and data encryption
- Understanding code injection and the importance of input validation
- Understanding code corruption and its mitigation through practices of encapsulation, data integrity, robustness, judicious extensibility, and immutability
- Understanding the basics of Java security policies and how to execute privileged code
- Presenting additional security guidelines on accessing the file system, utilizing deserialization properly, and the importance of good class design practices

Java SE 11 Developer Exam Objectives	
[10.1] Develop code that mitigates security threats such as denial of service, code injection, input validation and ensure data integrity	§26.1, p. 1600 §26.2, p. 1602
[10.2] Secure resource access including filesystems, manage policies and execute privileged code	§26.3, p. 1608 §26.4, p. 1610

The topic of secure coding is marginal compared to the core topics of the Java SE 11 Developer exam. No attempt is made here to cover this important topic comprehensively in a single chapter. Selected topics in secure coding are covered to the extent of their relevance for Java SE 11 certification. We encourage the reader to consult literature on *software security* that encompasses essential

security topics in developing secure and reliable systems, not just secure coding.

26.1 Application Security Overview

This chapter is quite different from other chapters in this book because of the nature of the topic that is covered here. So far this book has covered Java language constructs and APIs that programmers use to implement the business logic of the application. Application designers and architects describe business logic requirements as *functional requirements* (FRs). Such requirements must be implemented to satisfy the business reasons for this application and implement its functionality— in other words, they answer the question of *what* this application should do. However, there is more to application design than that. There is another set of requirements that application designers and architects should consider, known as *nonfunctional requirements* (NFRs). These include things that are not directly related to the question of *what* the application is supposed to do, but rather characterize *how* it is supposed to work—for example, application performance, scalability, user interface ergonomics, maintainability, and security.

Securing and protecting applications is an important NFR and is usually treated as a separate set of concerns in the overall development cycle. NFR-related design decisions are a complex set of considerations that have to be resolved by a fairly large group of participants, including system and application architects, application designers, and even business stakeholders, not just programmers. Programmers usually are responsible for implementing design and architecture decisions that are made for them. However, it is still quite beneficial for a programmer to be able to understand and appreciate the logic behind the NFRs. Therefore, a reasonable level of understanding of the security principles of application design is advised for any practical programmer. Such an appreciation may help programmers reduce cases when code they have written has to be adjusted or reengineered to satisfy NFRs, including security design decisions.

It is important to understand that security enforcement always comes at a price. This could be a direct financial cost of actually investing in designing, coding, and maintaining a secure environment, but also other overheads, such as decreased performance or decreased throughput of the application that could be caused by data encryption and decryption, or verification and validation of values, and so on.

Improving application security can have implications on its usability—for example, making it less convenient for users to access the application or use some of its functionality because security measures may require users to overcome extra hurdles, such as using two-factor authentication or connecting using virtual private network (VPN). Therefore, a cost-benefit analysis should always be performed as part of the application security design process.

The question that needs to be addressed is this: Is extra protection worth the increase in the up-front development investment, potential user inconvenience, increased maintenance costs, and possible adverse impact on performance and scalability? This is not an easy question to answer because it may involve considerations that go far beyond software design concerns. For example, data leaks caused by security breaches can carry financial and reputation risks for a company, or can even put the company in breach of regulatory requirements and thus have legal implications. Overall security design is often a compromise that is a reflection of a precarious and delicate balance between improved protection and the overhead that it entails.

Lastly, it should be noted that the overall security of the application can only be as good as its weakest link. There is no point in overinvesting in addressing a specific security concern if other related areas are not secured at a comparable level. Potential attackers will always look for the weakest link in the overall security model of the application. Don't forget that it is not just software, but also humans that could be the weak link in security. For example, people who are not careful about keeping their password secret could pose a significant security risk depending on their role in the organization, or what information they can access.

Many of these considerations go far beyond the scope of this book because they are considered to be a responsibility of designers and architects rather than programmers. Therefore, this chapter takes a much narrower approach and only discusses issues that have a direct impact on programmers' activities related to security implementation within an application.

It is also worth mentioning that many security concerns raised in this chapter are applicable to much more complex Java applications hosted in the Java Enterprise Edition (EE)/Jakarta and MicroProfile server environments. Thus many implementation details would fall outside the Java SE scope. However, a Java programmer should still have some degree of awareness of these issues

and at least the basics techniques for how to remedy them because Java classes are portable between runtime environments, and thus some of the code written in the context of the Java SE application may eventually end up being used in the Java server environment. This chapter also avoids going into detail about environment management tasks, such as creating and maintaining SSL keys and managing keystore files, and instead keeps the reader focused on programmer-specific tasks related to security implementation. These extra topics may be of interest to developers who not only write code, but also are responsible for maintaining environments were the code will be executed. However, these topics are beyond the scope of Java SE certification.

26.2 Security Threat Categories

Security threats can be grouped into categories based on the nature of the threat and its impact on a system. Each category has its own characteristics that programmers need to be aware of, in order to address these security threats throughout the software lifecycle.

Denial-of-Service (DoS) Attacks

Denial-of-Service (DoS) attacks are a group of threats that are related to the way in which an application manages its resources—specifically, various attempts to exploit the lack of checks and restrictions around resource utilization. For example, a resource could be an open port, or a file, or memory allocation—in other words, something that an application needs for its operational requirements. The idea behind the DoS attack is to make requests to the application for its resources so that their availability for legitimate use is blocked. In DoS attack scenarios, the attacker is trying to exploit the lack of controls and restrictions that an application applies when it is allocating or accessing its resources.

The following scenarios are examples of DoS attacks:

- An attacker providing a very large file or document containing recursive references, causing the application to waste resources trying to parse such a document. This could cause the program to run out of memory, go into an infinite loop trying to handle the recursive execution, or simply slow down as it attempts to read a large document. In any case, this would impede this application's ability to process legitimate data from other sources.

- An attacker opening a connection to a port through which the application performs network interactions. Once this connection is established, the attacker can start sending or receiving information in the slowest possible manner, causing the application to block for read or write operations to complete. The attacker can also spawn a large number of such connections, with the eventual goal of causing this application to run out of its capacity to handle simultaneous connections and denying service from legitimate users.

Countermeasures that can remedy DoS attacks include the following:

- Checking file sizes before starting parsing
- Detecting recursive references and stopping data processing if such recursions occur
- Discarding suspicious documents
- Detecting how many connections a given client has opened concurrently
- Detecting and terminating excessively slow uploads and downloads
- Dropping suspicious connections
- Limiting access to program logic to only authenticated and authorized code

Sometimes it might be the case that a legitimate user struggles with a network connection due to poor network quality. This means that slow connections must still be handled by the program. Consider using asynchronous I/O capabilities to handle these types of slow connections. Asynchronous I/O is supported in Java Enterprise Edition (EE)/Jakarta and MicroProfile servers.

Sensitive Information Leakage

Sensitive information leakage is a group of threats that are related to the way in which an application maintains and transmits its data. This could be internal information storage as well as data exchanges and transmission of values across the network. Reasons behind this type of security vulnerability typically are the lack of value obfuscation, encryption, and information reduction.

For example, the application may have to handle sensitive values, such as personal information or financial details. Such values should be stored in a secure manner, and should not be transmitted through the network in clear text, or maybe not transmitted at all. Objects containing sensitive information should be expunged from memory as soon as possible. Sensitive information should not be written into logs, which includes omitting such information from excep-

tion logging. And of course, it is not recommended to serialize sensitive data. Consider performing selective serialization (§20.5, p. 1261), marking such variables as `transient`, or obfuscating values before serializing.

Nevertheless, sometimes programs do need to write sensitive data, or transmit it through the network. If that is the case, there are several techniques that should be considered to secure such data.

Value Obfuscation

Value obfuscation is a process of scrambling data, producing a digest value that can be used in place of the original data. The purpose of the digest is to act as a value substitution which is not supposed to be unscrambled.

Click here to view code image

```
try {
  String creditCard = "12345678890";
  MessageDigest md = MessageDigest.getInstance("SHA-256");  // (1)
  byte[] digest = md.digest(creditCard.getBytes());         // (2)
  String hash = (new BigInteger(1, digest)).toString(16);   // (3)
} catch (NoSuchAlgorithmException e) {
  e.printStackTrace();
}
```

This example assumes that a credit card number needs to be obfuscated. The numbered comments below correspond to the numbered lines in the code:

1. A `java.security.MessageDigest` object is created using the *SHA-256 digest algorithm*. The SHA algorithm is designed to take a variable-length input and produce a fixed-length digest result. In the case of the SHA-256 algorithm, the number 256 refers to the length of the result in bytes.
2. A digest value in the form of a `byte` array is obtained from the original value.
3. The digest value is represented as a string of fixed length.

Now this digest can be used in place of the original credit card number. It can be stored, or transmitted in the network.

Data Encryption

Another approach is to secure data using encryption. In order to perform encryption and decryption of information, a cipher algorithm and a key have to be selected. This key is then used to perform actual encryption and decryption actions, utilizing an appropriate cipher algorithm. The exact list of supported digest and encryption algorithms varies among different versions and implementations of Java, so developers should consult release-specific Java documentation for a full list of supported algorithms.

Click here to view code image

```
try {
  String creditCard = "12345678890";
  SecretKey key = KeyGenerator.getInstance("AES").generateKey(); // (1)
  Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");       // (2)
  cipher.init(Cipher.ENCRYPT_MODE, key);                         // (3)
  byte[] encryptedValue = cipher.doFinal(creditCard.getBytes()); // (4)
  GCMParameterSpec ps = cipher.getParameters()
                              .getParameterSpec(GCMParameterSpec.class);
  cipher.init(Cipher.DECRYPT_MODE, key, ps);                     // (5)
  byte[] decryptedValue = cipher.doFinal(encryptedValue);        // (6)
} catch (GeneralSecurityException e) {
  e.printStackTrace();
}
```

The numbered comments below correspond to the numbered lines in the code:

1. The key in this example is generated on the fly using the AES algorithm. However, often keys are created in advance and stored in secure SSL wallet files that implement password-protected key storage.
2. A new `javax.crypto.Cipher` object is created that is associated with a cipher algorithm.
3. A cipher is now associated with a key and initialized to be used for encryption.
4. A value is encrypted using the previously initialized cypher. There are several methods provided by the `Cipher` class that can perform the encryption or decryption, such as `update()` and `doFinal()`. In simple cases, a single `doFinal()` method call can be used to perform encryption or decryption actions. In more complex scenarios, such as the need to handle large messages,

multiple-part encryption or decryption mechanics can be used with intermediate actions performed by the `update()` method, before the invocation of the `doFinal()` method.

5. Next, a cipher parameter specification object is acquired, which is used to set up the cipher for the decryption mode. Notice that the same key that was used for encrypting the values is now used to decrypt them.
6. Finally, the cipher is used to decrypt the values.

Notice that the cipher encrypts and decrypts values as `byte` arrays, so any other type values should first be converted into a `byte` array before encryption can happen. After decryption completes, the `byte` array containing the decrypted value can be converted to the required type.

Finally, when exchanging information among different systems, consider only transmitting the minimal amount of data outside an application, if possible.

Code Injection and Input Validation

Code injection is a group of threats that are related to malicious code being injected into the application, instead of legitimate values that the application is supposed to be processing. Code injection is often related to interpreted programming languages, and situations when dynamically supplied code can be executed by the application. Usually, neither of these cases is applicable to a Java application because Java code is fully verified and compiled before execution. However, Java programs can pass executable code disguised as legitimate values that are the source of the code injection. Once again, there is a possibility that malicious executable code can be supplied instead of a regular data value that the application is supposed to handle, causing the application behavior to change when the malicious value is interpreted. Consider validating input values by checking value patterns to ensure that data appears legitimate. Also, consider sanitizing input values by modifying values in a way that prevents their interpretation as executable code.

SQL Injection

Consider the code below, which can be exploited for SQL injection (§24.4, p. 1525). It uses a basic JDBC `Statement` object at (1) to execute a query, but performs a simple concatenation to create the query string passed to the method.

Click here to view code image

```
final String jdbcURL = "jdbc:derby:musicDB";
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement s = connection.createStatement()) {
  String value = "whatever";
  ResultSet rs = s.executeQuery("select * from songs where name like '"
                                + value + "'");                // (1)
} catch (SQLException e) {
  e.printStackTrace();
}
```

The expectation is that this code snippet performs a query. However, this code can be exploited if the user submits a SQL statement as a parameter value, such as `"Beatles'; drop table songs;"`. To avoid this problem, consider using a `Prepared-Statement` object (§24.4, p. 1526), which does not concatenate parameters, or using an `enquoteLiteral(value)` method provided by the `Statement` object to sanitize the parameter value.

Code Corruption

Code corruption is a group of threats that are related to the way in which Java code can be exploited by application developers. There is a possibility that the application logic can be broken because of a lack of encapsulation, or immutability, or loss of semantic cohesion when extending classes. Arguably, these types of issues could be considered general programming malpractice problems rather than security threats per se.

Encapsulation

A field that is declared as `public` allows developers to write code that reads and writes its value directly, which may result in data corruption and loss of integrity, even though relevant safe operations could be provided by the application. The issue is that a developer is not forced to invoke such operations if direct access to data is allowed due to lack of encapsulation.

Click here to view code image

```
public class SyncCounter {
  public int value;                                        // (1)
  public synchronized void increment()  { value++; }
```

```
    public synchronized int  getValue()   { return value; }
}
```

In the preceding code, the variable value at (1) should be `private`; otherwise, no enforcement of memory integrity would be possible. This principle is applicable to any operation—not just those that enforce synchronization (§22.4, p. 1387), but also those that validate values, or transform values, or perform any additional actions besides just setting or getting values.

Data Integrity

Restricting access to data via methods provides additional opportunities to improve overall program integrity. Operations can be used to validate values, and guard against erroneous values, such as number overflow (§2.8, p. 59). Consider using methods that guard against such overflows. It may be better for a program to throw an `ArithmeticException`, rather than silently overflow a numeric value.

Click here to view code image

```
int x = Integer.MAX_VALUE;
int y = x + 1;                          // (1)
int z = Math.addExact(x, 1);            // (2)
```

This example attempts to add 1 to the maximum `int` value. The code at (1) causes a value to overflow and wrap around (§2.8, p. 59), and the code at (2) throws an `ArithmeticException` when overflow occurs.

Similarly, a floating-point operation may result in positive or negative infinity, or result in *Not-a-number* (NaN) when attempting floating-point division by zero. Developers are advised to guard against such erroneous cases using simple `boolean` verification of operation results.

Click here to view code image

```
boolean veryLargeNum = Double.isInfinite(1/Double.MIN_VALUE); // (1)
boolean nan = Double.isNaN(0.1/0);                            // (2)
```

The code at (1) checks if the operation result is an infinitely large number, and the code at (2) checks if the operation result is NaN.

Robustness

Finally, it is worth considering guarding against references with a `null` value using an `Optional` object (§16.6, p. 940).

Click here to view code image

```
Optional<String> o = computeOptionalResult(); // (1)
String s = null;
if (o.isPresent()) {                          // (2)
  s = o.get();
}
s = o.orElse("Default");                      // (3)
s = o.orElseThrow();                          // (4)
```

The method at (1) returns an `Optional` object that can encapsulates an actual value. The caller is forced to check what value is in the `Optional` object and take appropriate action. Here are some example of such actions:

- Check whether the value is present before attempting to retrieve it from the `Optional` object.
- Substitute a default value if a value is not present in the `Optional` object.
- Throw a `NoSuchElementException` if the value is not present in the `Optional` object.

The common requirement for all of these cases to work is proper encapsulation to ensure that relevant validations and value corrections are actually applied by appropriate methods.

Extensibility

An example of a loss of semantic cohesion when extending classes is when a developer extends a class and overrides a method in such a way that the overriding method breaks semantic assumptions imposed by the supertype method (§5.1, p. 196). This may lead to erroneous interpretation of the behavior when using polymorphism. Other developers may expect the invoked method to ad-

here to certain semantics, but instead the invoked method would exhibit unexpected and possibly erroneous behavior.

Immutability

In order to avoid code corruption issues, consider using immutable design, declaring all variables that do not change value as `final` (§5.5, p. 225), as well as declaring classes and methods as `final` to prevent them from being overridden in all cases where extensibility is not required, or if certain method semantics are critical for application functionality.

Click here to view code image

```
public class Authenticator {
  public boolean authenticate() {           // (1)
    /* Perform authentication logic. */
  }
}
```

The method or even the entire class above could be made `final` to ensure that no other class can extend this class or override its methods. This would prevent overriding method implementations that can disable or compromise the actual authentication process. This principle is also applicable in many other cases, such as operations that validate values, ensure memory integrity, and so on.

26.3 Java Security Policies

Security policies are designed to impose restrictions on code execution and on access to resources. Security configuration settings are recorded in the properties file `${java.home}/conf/security/java.policy`, where the environment variable `java.home` is the root directory of a Java runtime image, or a directory where the JDK is installed on a given machine. This configuration file contains properties defined as name-value pairs that describe general security settings, references to certificate keystore files, and references to other security policy files that can be found in other locations.

Here is an example of a configuration in the j`ava.security` properties file to reference policy files:

Click here to view code image

```
policy.url.1=file:${java.home}/conf/security/java.policy
policy.url.2=file:/anypath/java.policy
```

These Java security policy descriptors configure security restrictions and permissions. These restrictions and permissions define access to resources and permissions to execute code for a specific codebase. The term *codebase* describes a location where Java code is placed, such as a directory or a URL, but more typically JAR archives. Each security policy is defined as a *grant* that is associated with a specific digital signature and allows the origin of the code to be authenticated.

Here is an example of a `java.policy` file structure:

Click here to view code image

```
keystore "xyz.keystore";
grant codeBase "file:/application.jar" signedBy "abc" {
  permission java.net.SocketPermission "localhost:8080", "listen";
  permission java.io.FilePermission "/FileSystemPath", "read, write";
};
```

The example above defines a keystore description and specifies a number of permissions. The keystore file is a secure store that contains keys and certificates. It is used to look up the public keys and associate digital signatures with a given codebase. Keystores are created and maintained using a keytool utility.

A grant is configured for a specific codebase given by the location of a JAR file containing classes to which this grant should be applied. In other words, classes within this archive will be allowed to perform restricted actions described within this grant. To make sure this is a genuine JAR file, it can be signed using a `signedBy` property that references a relevant digital signature alias from the keystore.

In the grant specification, this policy descriptor defines a number of permissions. One permission allows classes contained within the codebase to listen on a certain address given by the host and the specified port. Another permission allows read and write access to a specific file system path. The exact nature of the permissions depends on the permission type, such as a socket or file permis-

sion used in this example. Custom permissions can also be created by extending the `java.security .BasicPermission` class.

Once security policies are defined, they can be verified.

Click here to view code image

```
SocketPermission socketPermission
        = new SocketPermission("localhost:8080", "listen");     // (1)
FilePermission filePermission
        = new FilePermission("/FileSystemPath", "read, write"); // (2)
try {
  AccessController.checkPermission(socketPermission);           // (3)
  AccessController.checkPermission(filePermission);             // (4)
  // (5) ...
} catch(AccessControlException e) {
  // (6) ...
}
```

The numbered comments below correspond to the numbered lines in the code:

(1)–(2) A number of permission objects can be initialized to match permissions configured through the security policy descriptor.
(3)–(4) The `AccessController` class is used to validate whether such permissions were indeed granted to the given class. This is determined based on the class location within a codebase referenced by the relevant grant. The method `checkPermission()` throws an `AccessControlException` if corresponding permission was not granted.
(5) Once permissions are verified, the program can proceed to perform restricted actions.
(6) Handle exceptions that are thrown when requested access does not match permissions configured by the policies.

Executing Privileged Code

There are cases when it is not known which specific permissions must be verified—for example, when the program logic is supplied before the actual permission configuration has been created, or when such a configuration is expected to change in the future. In these cases, the program can use the `AccessController.doPrivileged()` method to execute the computation with privi-

leges enabled. This method accepts an object that implements the `PrivilegedAction` or the `PrivilegedExceptionAction` interface, depending on whether this computation can potentially throw a checked exception.

Click here to view code image

```
String content = AccessController.doPrivileged(
    new PrivilegedAction<String>() {
      @Override
    public String run() {                                          // (1)
      String result = null;
      try {
        result = Files.readString(Path.of("/FileSystemPath")); // (2)
      } catch (IOException ex) {
        // ...                                                        (3)
      }
      return result;
    }
  });
```

The numbered comments below correspond to the numbered lines in the code:

1. The implementation of the `PrivilegedAction` interface must override the `run()` method.
2. The implementation of the `run()` method contains logic that has access to restricted resources.
3. Any checked exceptions thrown must be handled within the `run()` method. If checked exceptions are to propagate outside of the `run()` method, then the `PrivilegedExceptionAction` interface should be implemented instead.

The method `doPrivileged()` is overloaded and can accept additional parameters, including specific permissions that must be satisfied for the privileged action to succeed.

26.4 Additional Security Guidelines

Attention is drawn to some selected security guidelines in this section.

Accessing the File System

When accessing the file system or performing I/O operations, consider the following recommendations:

- Remove redundant elements from the path and convert it to a canonical form using the methods `normalize()` and t`oRealPath()` to guard against directory traversal attacks, such as attempts to guess the directory structure by using relative paths (§21.3, p. 1297).
- Verify the file sizes and lengths of I/O streams before attempting to process information.
- Monitor I/O operations to detect excessive use of resources and terminate operations that process excessive amounts of data.
- Release resources as soon as possible, and terminate and time out long operations.

Deserialization

Be mindful of deserialization (§20.5, p. 1261), as it can create objects that are beyond the control of the application and that can bypass normal constructor invocations. Essentially, the deserialization process circumvents normal secure object creation, and thus must be considered a potential risk factor, especially if a serialized object arrived from an untrusted source. Thus it is a good idea to validate such objects as soon as they are created.

Class Design

Another important set of guidelines is related to class design.

- Always strive to enforce tight encapsulation using the most restrictive access permissions possible, also known as *principle of least privilege* (PoLP). Consider utilizing the Java Platform Module System for further reinforcement of encapsulation (§19.3, p. 1173), including restrictions imposed on the use of reflection (§19.8, p. 1191).
- Make objects immutable (§6.7, p. 356). Don't forget that even though a variable referencing an object can be marked as `final`, it does not mean that the object itself is immutable. It may be a good idea to create cloned object replicas of otherwise mutable objects to avoid unsafe memory mutable operations.

- When extending classes and overriding methods, carefully consider the semantics of the superclass methods to override. Also, remember that any modification made to a superclass, such as changing method implementation or introducing additional methods, may have a cascading effect on all its subclasses in the inheritance hierarchy (§5.1, p. 191).
- Always mark non-`private` classes and methods as `final` if they are not intended to be extended or overridden (§5.5, p. 225).
- Use factory methods to validate values before invoking a constructor to actually create an object.
- Avoid invoking overridden methods from constructors (§10.9, p. 555).
- Declare constructors as `private` if the class should not be instantiated.

Finally, consider never disabling bytecode verification with command-line options such as `-Xverify:none` or `-noverify` that protect bytecode against tampering and dangerous behavior.

Review Questions

26.1 Which of the following can prevent DoS attacks?

Select the one correct answer.

a. Obfuscating sensitive data

b. Sanitizing input values

c. Terminating recursive data references

d. Encapsulating code

26.2 An input parameter containing executable code is an example of which of the following?

Select the one correct answer.

a. Lack of encapsulation

b. Mutable code

c. Code corruption

d. Code injection

26.3 Given the following code:

Click here to view code image

```
public class RQ3 {
  public static void main(String[] args) {
    int z = Math.addExact(Integer.MAX_VALUE, 1);
    System.out.println(z);
  }
}
```

What is the result?

Select the one correct answer.

a. `2147483647`

b. `-2147483648`

c. `2147483648`

d. An `ArithmeticException`

26.4 Which of the following describes the meaning of the number 256 in the name of the SHA-256 algorithm?

Select the one correct answer.

a. A length of a public key used for encryption

b. A length of a private key used for encryption

c. A length of the resulting hash value

d. A length of the input value

26.5 What is the purpose of Java security policies?

Select the one correct answer.

a. To set restrictions on code execution and on access to resources

b. To set POSIX permissions for file system resources

c. To restrict access to data via security methods

d. To prevent code injections

26.6 Which of the following statements is true about privileged code execution?

Select the one correct answer.

a. The method `PrivilegedAction.run()` is expected to throw a checked exception if security constraints are violated.

b. The method `AccessController.doPrivileged()` can handle checked exceptions thrown by the `PrivilegedAction.run()` method.

c. Checked exceptions must be handled within the `PrivilegedAction.run()` method.

d. Checked exceptions thrown by the `PrivilegedAction.run()` method can be propagated out of the `AccessController.doPrivileged()` method.