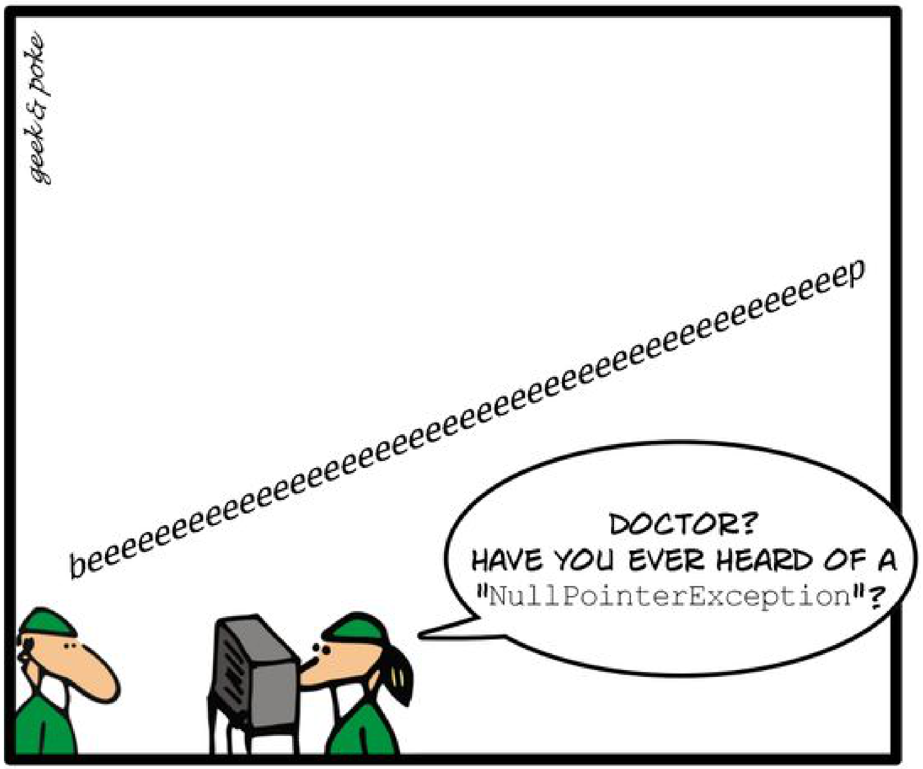
Edinburgh Napier UNIVERSITY

Secure Software Development



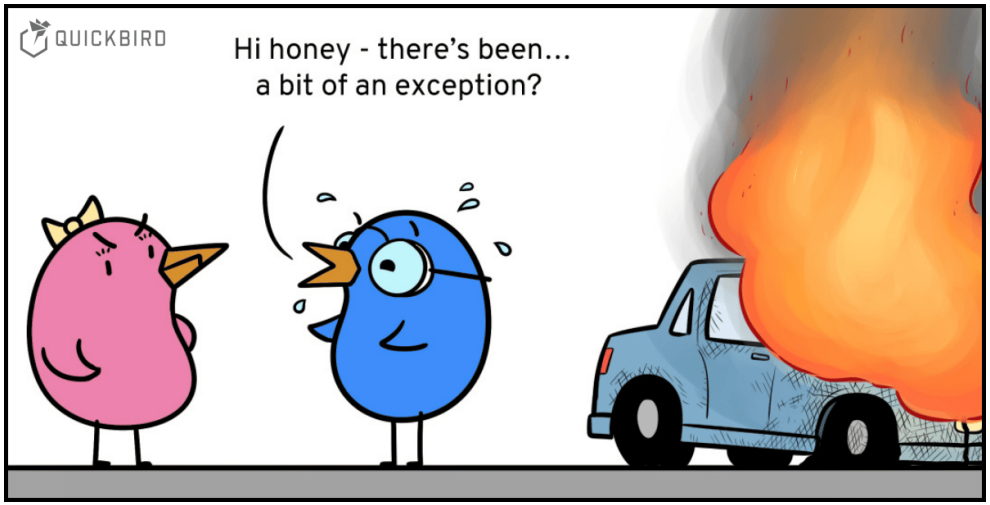
RECENTLY IN THE OPERATING ROOM

Secure Design Principles & Exceptions

Prof Ashkan Sami

Contents

- Secure System Design Principles
  - OWASP Principles of Security
- Exception overview
  - Can be skipped if familiar
- Their vulnerabilities
- Mitigation strategies



Edinburgh Napier UNIVERSITY

Overview of Secure Design Principles

- Introduction: Secure designs are intentional, founded on robust principles.
- Historical Context: Rooted in Saltzer and Schroeder's 1975 work on computer systems.
- Emphasis: Proven effectiveness in various security contexts.
  - Security should not be an afterthought or add-on.
  - Identify relevant security requirements and treat them in the overall process and system design.
  - Begin with establishing and adopting relevant principles and policies as a foundation for your design, then build security into your development life cycle.

Edinburgh Napier UNIVERSITY

Good Enough Security

- Concept: No absolute security; balance is key.
- Practical Approach: Align security level with asset value.
- Example: Avoid over-securing low-value assets - tailor security to necessity.

Principle of Least Privilege

- Definition: Minimal rights for tasks.
- Benefit: Reduces system vulnerability and damage potential.
- Case Study: Sendmail exploit - Demonstrates risks of excessive privileges.

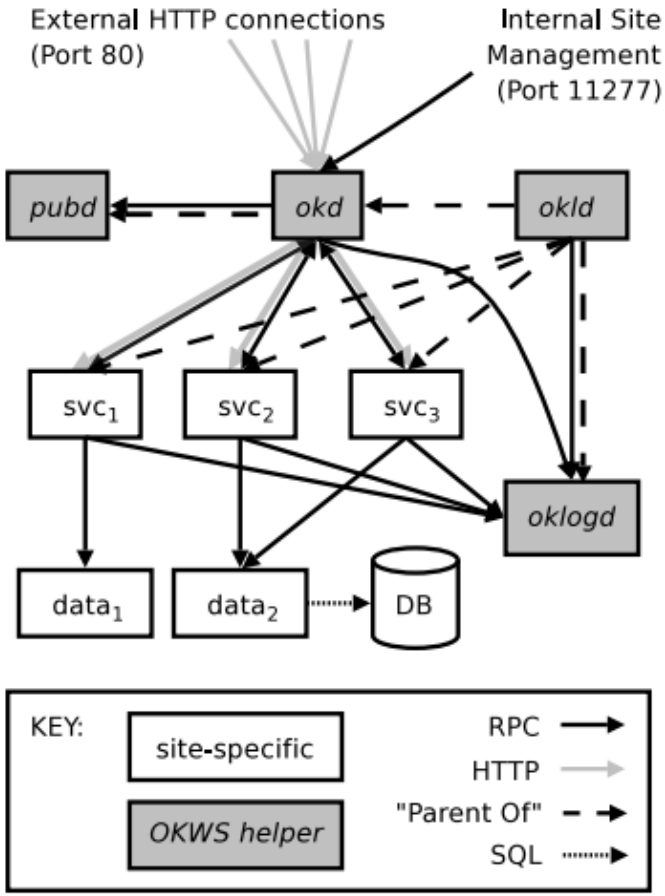


Figure 1: Block diagram of an OKWS site setup with three Web services (svc_1, svc_2, svc_3) and two data sources ($data_1$, $data_2$), one of which ($data_2$) is an OKWS database proxy.

Separation of Duties

- Concept: Divide critical tasks among multiple individuals.
- Implementation: Multi-condition checks in software for task completion.
- Objective: Prevent system abuse and ensure comprehensive oversight.

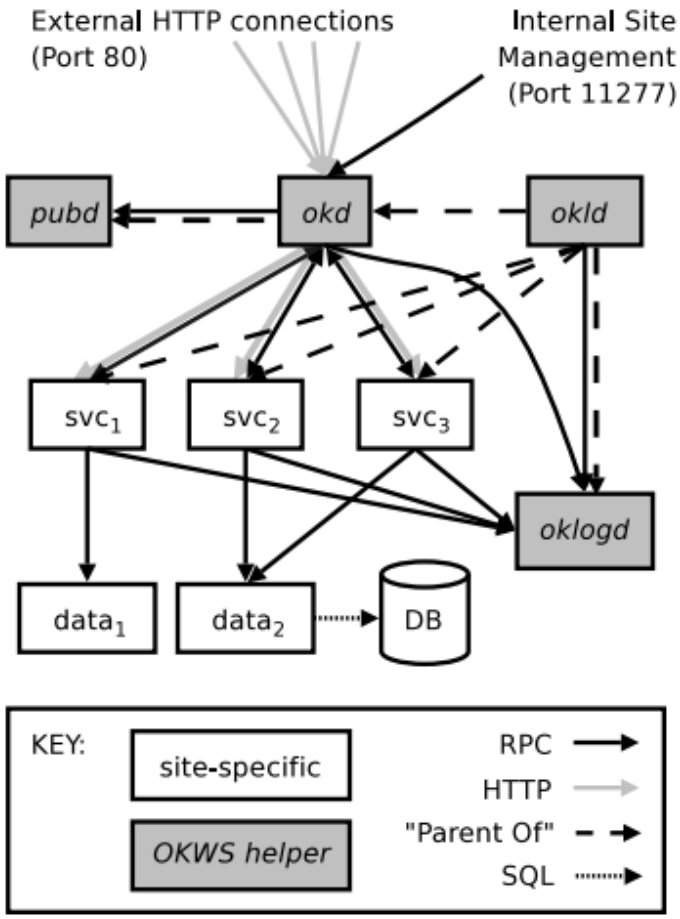
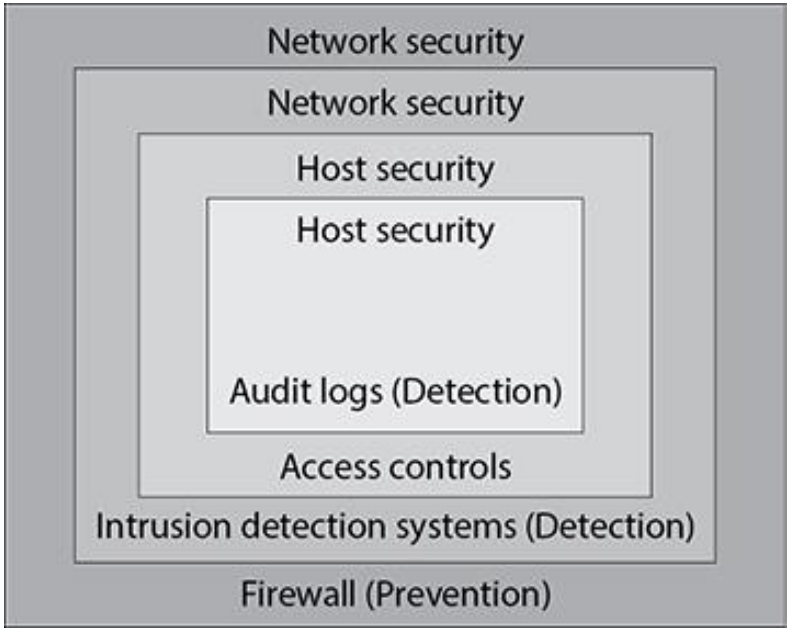


Figure 1: Block diagram of an OKWS site setup with three Web services (svc_1, svc_2, svc_3) and two data sources ($data_1$, $data_2$), one of which ($data_2$) is an OKWS database proxy.

Defense in Depth

- Strategy: Layered, overlapping defenses.
- Analogy: Castle defenses - multiple layers for robust protection.
- Software Application: Combine diverse security measures for comprehensive defense.



Bodiam Castle, East Sussex

Edinburgh Napier UNIVERSITY

Fail-Safe Design

- Principle: Systems failing to a secure state.
- Method: Default to deny unauthorized actions.
- Consideration: Error conditions may be a result of an attack, or may be due to design or implementation failures, in any case the system / applications should default to a secure state rather than an unsafe state.
  - This principle, suggested by E. Glaser in 1965, means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted.

Edinburgh Napier UNIVERSITY

Economy of Mechanism

- Concept: Simplifying security for better management.
- Strategy: Minimize services, streamline security processes.
- Objective: Enhance security management, reduce vulnerabilities.
  - The likelihood of vulnerabilities increases with the complexity of the software architectural design and code. And increases further if it is hard to follow or review the code.
  - The attack surface of the software is reduced by keeping the software design and implementation details simple and understandable.

Edinburgh Napier UNIVERSITY

Complete Mediation

- Principle: Consistent authorization verification.
- Importance: Ensures unbreachable security management.
- Application: Vital in operating systems with stringent authentication.
- A security principle that ensures that authority is not circumvented in subsequent requests of an object by a subject, by checking for authorization (rights and privileges) upon every request for the object.

- In other words, the access requests by a subject for an object are completely mediated every time, so that all accesses to objects must be checked to ensure that they are allowed.

Edinburgh Napier UNIVERSITY

Open Design

- Concept: Transparency in security mechanisms.
- Application: Cryptography focuses on key secrecy, not algorithm secrecy.
- Counter to Security through Obscurity: Open design is more effective than hiding mechanisms.

Edinburgh Napier UNIVERSITY

Least Common Mechanism

- Principle: Avoid shared mechanisms to minimize information leaks.
- Balance: Specialized processes vs. shared ones.
- Consideration: Weighing component reuse against the need for segregation.
- The security principle of least common mechanisms disallows the sharing of mechanisms that are common to more than one user or process if the users or processes are at different levels of privilege. This is important when defending against privilege escalation.

Edinburgh Napier UNIVERSITY

Psychological Acceptability

- Focus: User-friendly and unobtrusive security.
- Challenge: Security that doesn't hinder user efficiency.
- Importance: Encouraging user compliance without compromising security.

The Weakest Link

- Concept: Strength determined by the weakest component.
- Strategy: Focus on strengthening the weakest link.
- Application: Include diverse defenses to fortify the system.

Achilles grew up to become the greatest warrior in the world. According to this tale, his mother, Thetis, attempted to ensure his immortality by dipping the infant Achilles in the River Styx – but his heel, where she held him, was left untouched by the magic water.

Edinburgh Napier UNIVERSITY

Leveraging Existing Components

- Benefit: Efficiency and security through component reuse.
- Objective: Manage attack surface by minimizing new vulnerabilities.
  - Ensuring that the attack surface is not increased, and no new vulnerabilities are introduced.
  - Existing components are more likely to be tried and tested, and hence more secure, and also should have security patches available.
  - Components developed within the open-source community have the further benefit of ‘many eyes’ and are therefore likely to be even more secure.

Edinburgh Napier UNIVERSITY

Avoiding Single Points of Failure

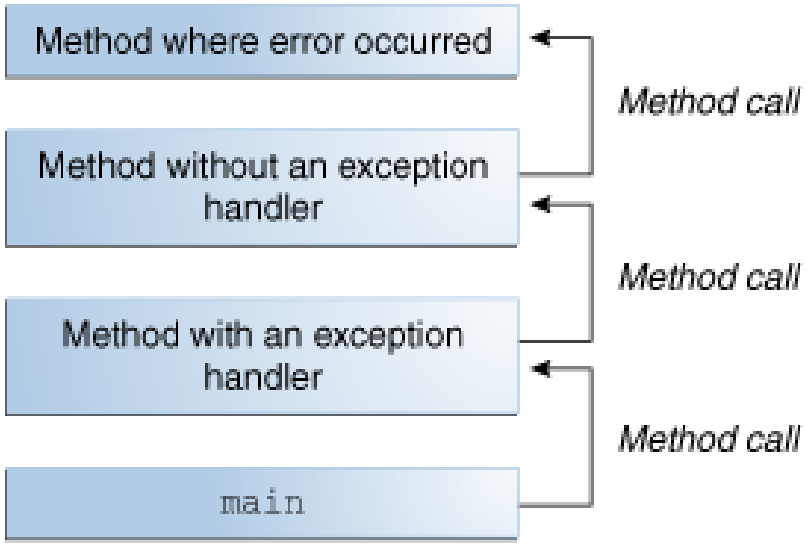
- Goal: Design that withstands individual component failures.
- Strategy: Analyze and mitigate singular failure points.
- Objective: Ensure redundancy and resilience in system design.

Edinburgh Napier UNIVERSITY

Summary on Design Principles

- Summary: Secure design integrates multiple, established principles.
- Final Thought: Comprehensive, deliberate design for robust system security.
- Keep also in mind that the system you are building also will be needing maintenance and that system operators will need to securely manage and even shutdown and dispose of the system.

Exception



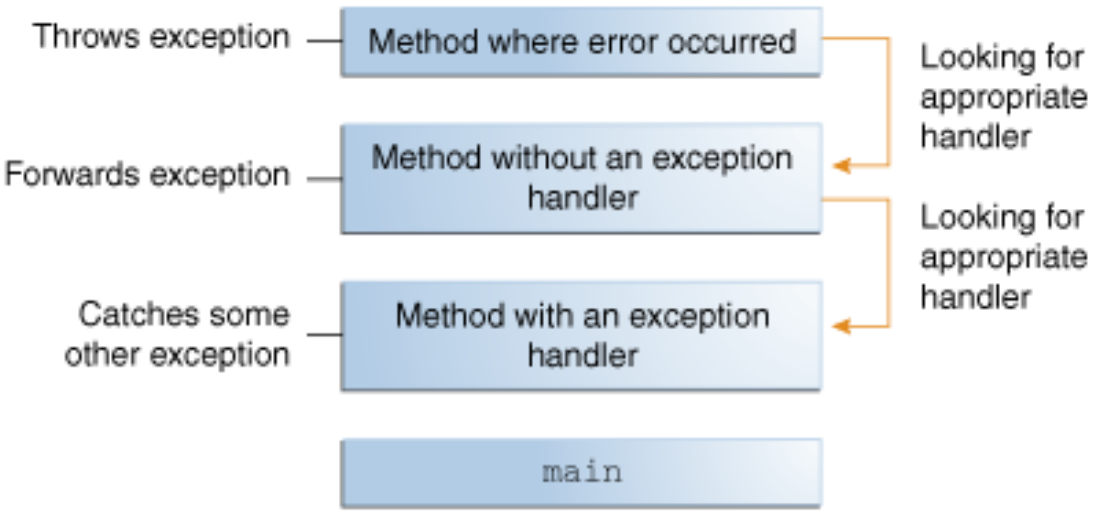
- After a method throws an exception, the runtime system attempts to find ***something*** to handle it.
- The set of possible "***somethings***" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.
- The list of methods is known as the ***call stack***.
- The call stack showing three method calls, where the first method called has the exception handler.

Exception

The runtime system searches the call stack for a method that contains a block of code that can handle the exception.

This block of code is called an exception handler.

- The search for a handler is done in reverse order that the function was called as shown in the figure.



try/catch/finally Block (continued)

Edinburgh Napier UNIVERSITY

```
try
{
    //statements
}
catch (ExceptionClassName1 objRef1)
{
    //exception handler code
}
catch (ExceptionClassName2 objRef2)
{
    //exception handler code
}
...
catch (ExceptionClassNameN objRefN)
{
    //exception handler code
}
finally
{
    //statements
}
```

Optional.

You can include a catch block for every possible exception that might have been generated inside the try block.

Edinburgh Napier UNIVERSITY

finally Block

- The most common use for a finally block is to close resources that were created inside the try block.
  - **Recall that resources take up memory!**
- For example, when you open various streams (FileReader, BufferedReader, FileWriter, etc) all of them are taking up resources.
- The ideal place to close those streams is NOT at the end of your try block, since if an exception is generated, control will never get as far as that point.
- Instead, close the resources in the finally block.

Order of catch Blocks

- The heading of a catch block specifies the type of exception it handles.
- A catch block can catch either exceptions of a specific type
  - e.g. **catch (ArithmeticException e)**
- or all types of exceptions
  - e.g. **catch (Exception e)**
- However, the order can matter! Can you see why?
  - If you catch the Exception class first (as opposed to one of its subclasses), that block will *always* get executed.

Order of catch Blocks (continued)

A reference variable of a superclass type can point to an object of its subclass:

- If in the heading of a catch block you declare an exception using the class Exception, then that catch block will catch **all** types of exceptions because the class Exception is the superclass of all exception classes

- In a sequence of catch blocks, exceptions of a **subclass** type should be placed **before** exceptions of a superclass type
  - In other words, if you declare the superclass type first, the catch block containing the subclass type will ***never*** be executed

Java Exception Hierarchy (continued)

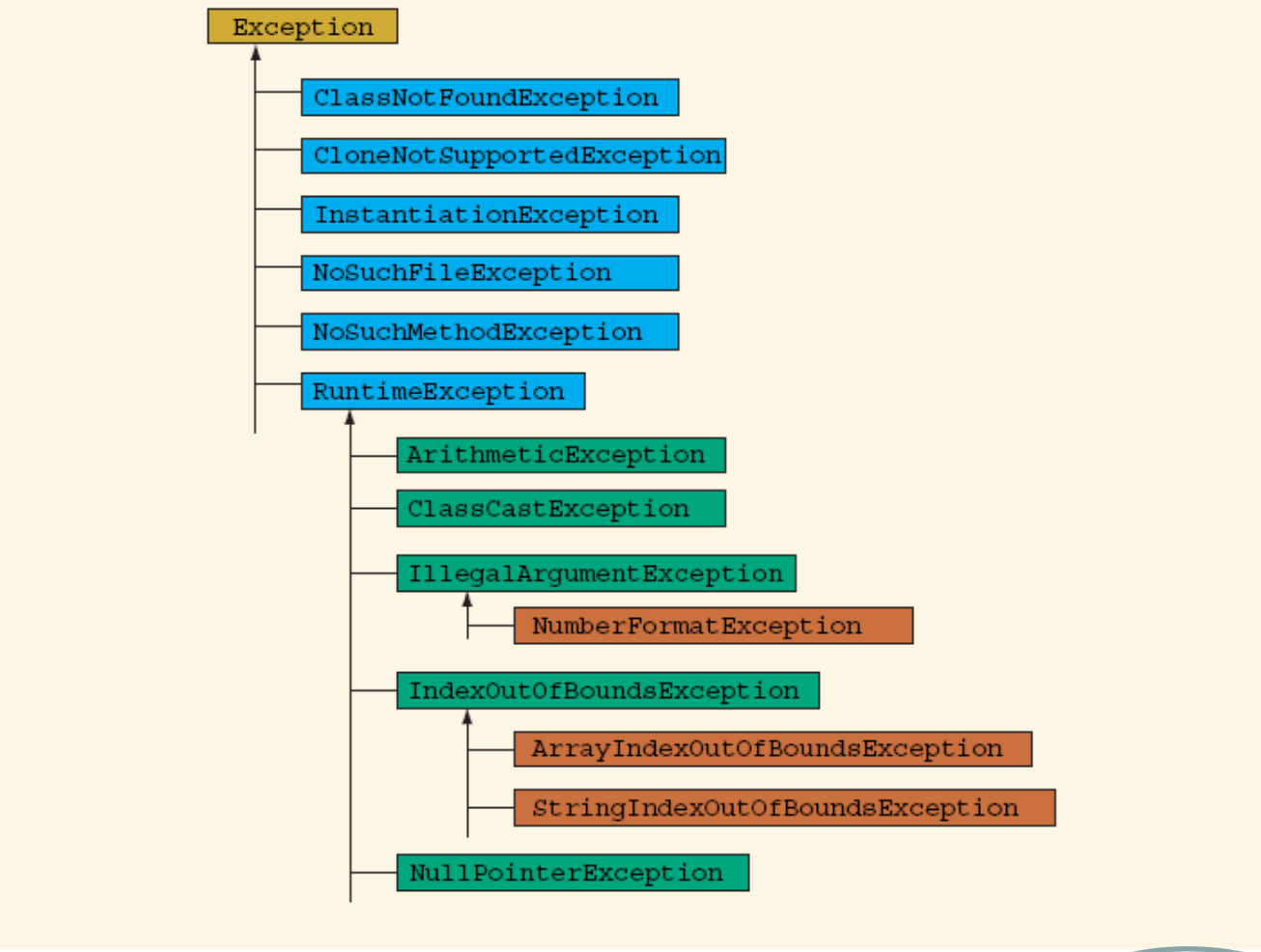


FIGURE 12-2 The class Exception and some of its subclasses from the package java.lang

Java Exception Hierarchy (continued)

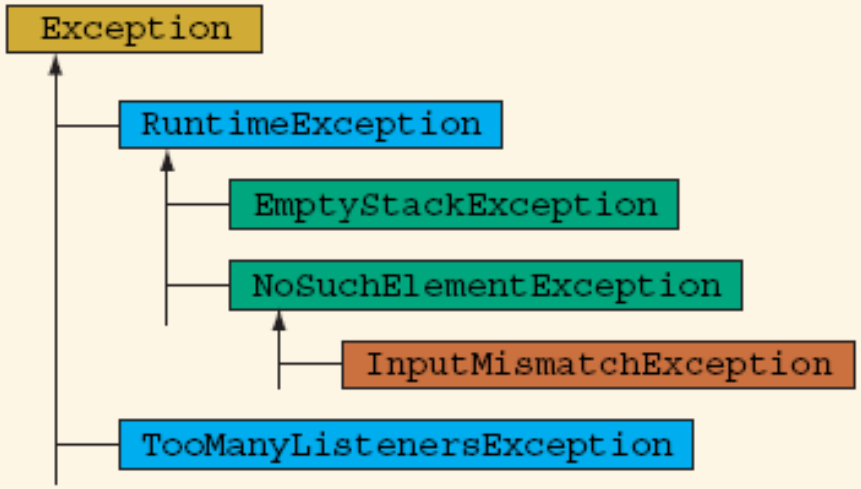


FIGURE 12-3 The `class` `Exception` and some of its subclasses from the `package` `java.util`. (Note that the `class` `RuntimeException` is in the `package` `java.lang`)

Checked vs Unchecked Exceptions

Edinburgh Napier UNIVERSITY

Exceptions are classified into two types: Checked and Unchecked.
If an exception is UN-checked, you are NOT required to place it inside a try/catch block and handle it. (Though you may choose to do so).

An **unchecked** exception is one that you are not *required* to provide for. For example, the nextInt() method of the Scanner class throws un-checked exceptions which is why we haven't had to write code to handle them.

A **checked** exception is one that the compiler *requires* you to provide for; that is, you must notify Java that you are aware that this exception might occur and that somewhere in your code, you are going to handle it. Your compiler will refuse to compile if you do NOT. For example, a FileNotFoundException is a checked exception, and is generated when you write code that is supposed to open a file for reading or writing.

- The FileNotFound exception can be generated in many places, such as when you attempt to open a File for reading input. This is why you had to enclose that code inside a try block.

Checked Exception: “Catch or Declare”

As was just mentioned, the compiler requires you to anticipate checked exceptions. It does so by enforcing a “**catch or declare requirement**” for checked exceptions.

Option 1: *Catch* the exception:
So far, we have handled checked exceptions by enclosing the code inside a try block and handling the code via a catch and/or finally block(s).

Option 2: *Declare* via a ‘throws’ clause.
The throws clause is part of your method signature. This clause indicates that the code inside your method may generate an exception, but that your method does not *handle* that exception (or handles it incompletely).

class RuntimeException

- Exceptions that inherit from this class are the only kinds of exceptions that do NOT have be enclosed in a try block.
  - In other words, these are unchecked exceptions
  - e.g. ArrayIndexOutOfBounds

- Therefore, all other types of exceptions (that are not subclasses of RuntimeException) *do* need to be enclosed in try blocks.
  - These are called **checked** exceptions.
  - Any code that you write that invokes a method that can generate an exception must be caught. If you neglect to do this, your program won't compile.

Checked vs Unchecked

- All exceptions that are *not* subclasses of RuntimeException must be checked (placed inside a try block) or thrown.
  - Class that are NOT of type RuntimeException are “checked” exceptions
    - Must be placed inside a try block
  - Subclasses of RuntimeException are unchecked exceptions.
    - Do not require a try block

Mnemonic: r**UN**time = **UN**checked!

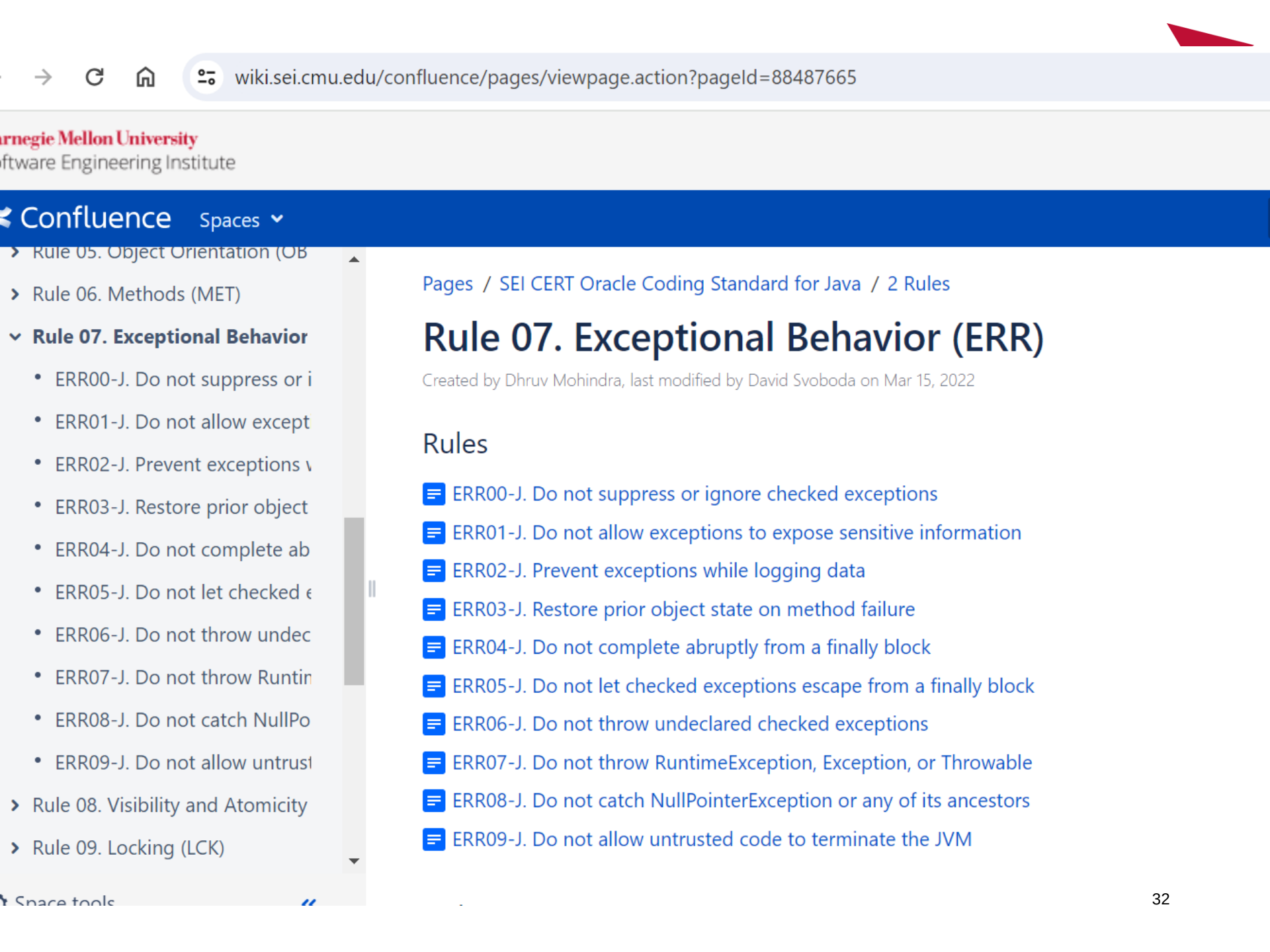
Review: Checked Exceptions

- Definition: any exception that can be recognized by the **compiler** (i.e. that the compiler requires you to anticipate).
- All subclasses of Exception except for subclasses of RuntimeException are Checked exceptions.
- Examples
  - IOException
  - FileNotFoundException
    - For example, see the API for the constructor of the FileReader or PrintWriter classes)

- **So, if you are writing a method that has code that can generate a checked exception, you must either:**
  1) Place the code inside a try block OR
  2) Throw the exception

Exceptions are for unavoidable situations

- Exceptions are for handling situations that arise that you (the programmer) can not control for. Exceptions are NOT for handling flaws in your code.
- In other words, don't use exceptions to deal with array boundaries – those are things the programmer should avoid with proper testing.
- Do use exceptions to deal with situations that can cause your program to malfunction, but for which you cannot control.
  - Bad user input
  - File not present / can't be opened


wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487665
rnegie Mellon University
ftware Engineering Institute
Confluence
Spaces
Rule 05. Object Orientation (OB
Rule 06. Methods (MET)
Rule 07. Exceptional Behavior
ERR00-J. Do not suppress or i
ERR01-J. Do not allow except
ERR02-J. Prevent exceptions v
ERR03-J. Restore prior object
ERR04-J. Do not complete ab
ERR05-J. Do not let checked e
ERR06-J. Do not throw undec
ERR07-J. Do not throw Runtir
ERR08-J. Do not catch NullPo
ERR09-J. Do not allow untrust
Rule 08. Visibility and Atomicity
Rule 09. Locking (LCK)
Space tools
Pages / SEI CERT Oracle Coding Standard for Java / 2 Rules
Rule 07. Exceptional Behavior (ERR)
Created by Dhruv Mohindra, last modified by David Svoboda on Mar 15, 2022
Rules
ERR00-J. Do not suppress or ignore checked exceptions
ERR01-J. Do not allow exceptions to expose sensitive information
ERR02-J. Prevent exceptions while logging data
ERR03-J. Restore prior object state on method failure
ERR04-J. Do not complete abruptly from a finally block
ERR05-J. Do not let checked exceptions escape from a finally block
ERR06-J. Do not throw undeclared checked exceptions
ERR07-J. Do not throw RuntimeException, Exception, or Throwable
ERR08-J. Do not catch NullPointerException or any of its ancestors
ERR09-J. Do not allow untrusted code to terminate the JVM

Exceptions pitfalls, mitigations and best practices based on Secure Coding

Ignoring exceptions

- If your code does not include exception handling, a single error could cause a crash, potentially bringing down the entire system.
- This vulnerability can be exploited by attackers who, by intentionally triggering an exception, could achieve a Denial of Service (DoS).
- This highlights the critical importance of robust error and exception handling in software development to prevent such vulnerabilities.

Not doing enough about exceptions

Merely catching an exception without addressing the underlying issue can lead to unstable states or hidden bugs in your application.

1. Ensure that any changes or allocations made by the code prior to the error are reversed and resources are properly deallocated if they are going to be abandoned due to the error.
2. try blocks should ideally encompass operations that are intended to be completed in full. If an exception interrupts this process, it is advisable to reset the state to what it was before, as though the operation never took place.

Not doing enough about exceptions

Merely catching an exception without addressing the underlying issue can lead to unstable states or hidden bugs in your application.

3. When considering a retry after an exception, be cautious to avoid leaving unnecessary allocations in place or getting trapped in an infinite loop.
4. Exercise caution with any code within a catch block that might itself cause an exception. Although such exceptions can be managed, the resulting logic can become complex and is generally better avoided to maintain clarity.

Information leakage

- Systems under development often contain extensive debugging output to aid in diagnosis.
- If this is not completely removed or deactivated in production systems, it can result in significant unintended information disclosure.
- Logging or error messages that are publicly visible might disclose internal states, potentially including sensitive data.
- While many programming languages facilitate the logging of stack traces in exception handlers, which is beneficial for debugging, this practice can also unintentionally expose internal details.

Best Practices for Secure Coding with Exceptions

1. **Implement Appropriate Exception Handling:** When calling methods that might throw exceptions, it's crucial to handle all possible exceptions, unless it's certain that the condition will not arise.

2. **Use Exceptions for Problem Notification:** Alert the calling code of problems through exceptions rather than ad hoc methods. For example, if a function returns null to indicate failure, any calling code that does not properly check for null can result in a crash due to null pointer access.

Best Practices for Secure Coding with Exceptions

3. **Catch and Respond to Specific Exceptions:** Prioritize addressing specific exceptions over general, abstract handling. Anticipating specific problems, like null pointer or I/O errors, enables the code to take more accurate and effective remedial actions.

4. Handling Exceptions in Large Try Blocks: In try blocks with extensive code, the same type of exception might occur at multiple points. To effectively address the issue, it may be necessary to implement flags or other mechanisms to pinpoint the origin of the exception.

5. Ensuring Test Coverage of Exception Paths: Comprehensive testing of all paths that may lead to exceptions is crucial for minimizing surprises in a production environment.

Best Practices for Secure Coding with Exceptions

6. Responding to Exceptions Appropriately: Exceptions typically signal significant issues; thus, merely catching them without any action is rarely advisable. Ignoring an exception conceals the problem from all higher-level callers in the stack.

7. Recovery or Rethrow Strategy: Either handle the error gracefully and recover from it, or rethrow the exception for it to be managed further up the stack. Unless you are confident in the recovery process, opting to rethrow the exception is a safer approach.

Best Practices for Secure Coding with Exceptions

8. Catching General Exceptions for Long-Lived Services: In Java, services that are intended to run continuously (long-lived services) should have a strategy for handling exceptions that might not be foreseen during development. Normally, it's considered a bad practice to catch general exceptions (like Exception or Throwable) because it can obscure specific errors that should be handled differently. However, in the case of long-lived services, this rule is relaxed.

High-Level Exception Handling: The advice is to place these general exception catch blocks at a high level in the code, typically at the boundaries of the system where requests are processed. For example, in a web service, this would be in the code handling incoming HTTP requests.

Best Practices for Secure Coding with Exceptions

Purpose of This Strategy:

a. Avoid Crashes: By catching general exceptions, the service can avoid crashing due to unexpected errors. This is crucial for services that need to maintain high availability.

b. Error Reporting and Recovery: Once a general exception is caught, the service can log the error, report it to the client in a controlled manner (e.g., sending an error response), and perform any necessary cleanup or rollback operations. This might include reversing transactional changes to maintain data integrity.

c. Application in Java: In Java, this might look like a try-catch block around the main request processing logic in your service. Inside the catch block, you'd handle the exception by logging it, preparing an appropriate error response for the client, and performing any necessary rollback of transactions or other cleanup activities.

Best Practices for Secure Coding with Exceptions

9. Ensuring Logs Exclude Sensitive Information: It is crucial to avoid including data values in logs unless you are certain that they are not sensitive. Details on how to log sensitive information in a separate, secure manner, without exposing it in standard logs or error responses, will be discussed later. The difficulty in determining the sensitivity of data necessitates a cautious approach. This is particularly important in general-purpose code, which may process various types of data depending on the application.

10. Caution with Stack Traces and Configuration Data: Avoid logging stack traces or configuration details, as these often contain sensitive information. They also include technical specifics, such as names of internal classes and methods, which could provide valuable information to potential attackers.

Best Practices for Secure Coding with Exceptions

9. Ensuring Logs Exclude Sensitive Information:

Noncompliant Code Example: writes a critical security exception to the standard error stream:

```
try {
  // ...
} catch (SecurityException se) {
  System.err.println(se);
  // Recover from exception
}
```

Compliant Code Example: uses java.util.logging.Logger, the default logging API provided by JDK 1.4 and later. Use of other compliant logging mechanisms, such as log4j, is also permitted.

```
try {
  // ...
} catch(SecurityException se) {
  logger.log(Level.SEVERE, se);
  // Recover from exception
}
```

https://wiki.sei.cmu.edu/confluence/display/java/ERR02-J.+Prevent+exceptions+while+logging+data

Edinburgh Napier UNIVERSITY

ERR02-J. Prevent exceptions while logging data

Risk Assessment

Exceptions thrown during data logging can cause loss of data and can conceal security problems.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR02-J	Medium	Likely	High	P6	L2

Automated Detection

Tool	Version	Checker	Description
CodeSonar	8.0p0	JAVA.DEBUG.LOG	Debug Warning (Java)
SonarQube	9.9	S106	Standard outputs should not be used directly to log anything

ERR07-J. Do not throw RuntimeException, Exception, or Throwable

Created by David Svoboda, last modified by Jon O'Donnell on Aug 06, 2021

Methods must not throw `RuntimeException`, `Exception`, or `Throwable`. Handling these exceptions requires catching `RuntimeException`, which is disallowed by ERR08-J. Do not catch NullPointerException or any of its ancestors. Moreover, throwing a `RuntimeException` can lead to subtle errors; for example, a caller cannot examine the exception to determine why it was thrown and consequently cannot attempt recovery.

Methods can throw a specific exception subclassed from `Exception` or `RuntimeException`. Note that it is permissible to construct an exception class specifically for a single `throw` statement.

Noncompliant Code Example

The `isCapitalized()` method in this noncompliant code example accepts a string and returns true when the string consists of a capital letter followed by lowercase letters. The method also throws a `RuntimeException` when passed a null string argument.

```
boolean isCapitalized(String s) {
  if (s == null) {
    throw new RuntimeException("Null String");
  }
  if (s.equals("")) {
    return true;
  }
  String first = s.substring(0, 1);
  String rest = s.substring(1);
  return (first.equals(first.toUpperCase()) &&
          rest.equals(rest.toLowerCase()));
}
```

Compliant Solution

This compliant solution throws `NullPointerException` to denote the specific exceptional condition:

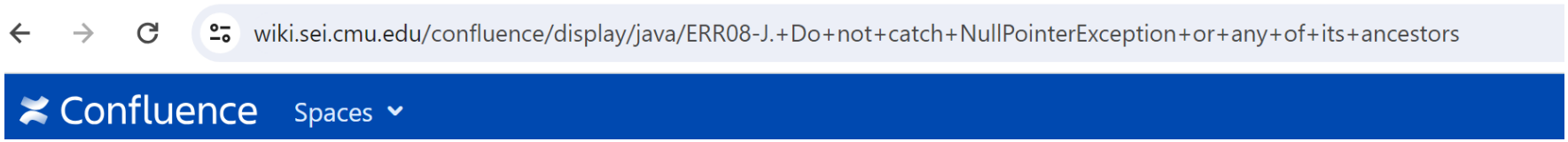
```
boolean isCapitalized(String s) {
  if (s == null) {
    throw new NullPointerException();
  }
  if (s.equals("")) {
    return true;
  }
  String first = s.substring(0, 1);
  String rest = s.substring(1);
  return (first.equals(first.toUpperCase()) &&
          rest.equals(rest.toLowerCase()));
}
```

Note that the null check is redundant; if it were removed, the subsequent call to `s.equals("")` would throw a `NullPointerException` when `s` is null. However, the null check explicitly indicates the programmer's intent. More complex code may require explicit testing of invariants and appropriate `throw` statements.

Risk Assessment

Throwing `RuntimeException`, `Exception`, or `Throwable` prevents classes from catching the intended exceptions without catching other unintended exceptions as well.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR07-J	Low	Likely	Medium	P6	L2



ERR08-J. Do not catch NullPointerException or any of its ancestors

Created by David Svoboda, last modified by Winfried Gerlach on Jan 25, 2022

Programs must not catch `java.lang.NullPointerException`. A `NullPointerException` exception thrown at runtime indicates the existence of an underlying `null` pointer dereference that must be fixed in the application code (see EXP01-J. Do not use a null in a case where an object is required for more information). Handling the underlying null pointer dereference by catching the `NullPointerException` rather than fixing the underlying problem is inappropriate for several reasons. First, catching `NullPointerException` adds significantly more performance overhead than simply adding the necessary null checks [Bloch 2008]. Second, when multiple expressions in a `try` block are capable of throwing a `NullPointerException`, it is difficult or impossible to determine which expression is responsible for the exception because the `NullPointerException catch` block handles any `NullPointerException` thrown from any location in the `try` block. Third, programs rarely remain in an expected and usable state after a `NullPointerException` has been thrown. Attempts to continue execution after first catching and logging (or worse, suppressing) the exception rarely succeed.

Likewise, programs must not catch `RuntimeException`, `Exception`, or `Throwable`. Few, if any, methods are capable of handling all possible runtime exceptions. When a method catches `RuntimeException`, it may receive exceptions unanticipated by the designer, including `NullPointerException` and `ArrayIndexOutOfBoundsException`. Many `catch` clauses simply log or ignore the enclosed exceptional condition and attempt to resume normal execution; this practice often violates ERR00-J. Do not suppress or ignore checked exceptions. Runtime exceptions often indicate bugs in the program that should be fixed by the developer and often cause control flow vulnerabilities.

Noncompliant Code Example (NullPointerException)

```
boolean isName(String s) {
  try {
    String names[] = s.split(" ");

    if (names.length != 2) {
      return false;
    }
    return (isCapitalized(names[0]) && isCapitalized(names[1]));
  } catch (NullPointerException e) {
    return false;
  }
}
```

- isName() method that takes a String argument and returns true if the given string is a valid name.
- A valid name is defined as two capitalized words separated by one or more spaces.
- Rather than checking the given string is null, the method catches NullPointerException and returns false.

Edinburgh Napier UNIVERSITY

Compliant Code Example (NullPointerException)

```
boolean isName(String s) {
  if (s == null) {
    return false;
  }
  String names[] = s.split(" ");
  if (names.length != 2) {
    return false;
  }
  return (isCapitalized(names[0]) && isCapitalized(names[1]));
}
```

isName() checks the String argument for null rather than catching NullPointerException:

Risk Assessment

Catching `NullPointerException` may mask an underlying null dereference, degrade application performance, and result in code that is hard to understand and maintain. Likewise, catching `RuntimeException`, `Exception`, or `Throwable` may unintentionally trap other exception types and prevent them from being handled properly.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR08-J	Medium	Likely	Medium	P12	L1

Compliant Code Example (NullPointerException)

- This compliant solution does not check for a null reference and permits a NullPointerException to be thrown.

```
boolean isName(String s) /* Throws NullPointerException */ {
  String names[] = s.split(" ");
  if (names.length != 2) {
    return false;
  }
  return (isCapitalized(names[0]) && isCapitalized(names[1]));
}
```

- Omitting the null check means that the program fails more quickly than if the program had returned false, and
- lets an invoking method discover the null value.
- A method that throws a NullPointerException without a null check must provide a precondition that the argument being passed to it is not null.

Exception handling: Example 1

```
01: boolean Login(String user, String pwd) {
02:    boolean loggedIn = true;
03:    String realPwd = GetPwdFromDb(user);
04:    try {
05:       if (!MessageDigest.getInstance("SHA-256").digest(pwd).equals(realPwd))
06:       {
07:          loggedIn = false;
08:       }
09:    } catch (Exception e) {
10:       // This cannot happen, ignore
11:    }
12:    return loggedIn;
13: }
```

- Consider a scenario where an attacker calls the Login function with parameters user="admin", and pwd=null.
- A null pointer exception is thrown at line 5, the program jumps to the catch block on line 9.
- The catch block on line 10, is mistakenly assumed to be redundant and thus, does nothing.
- Because the loggedIn flag was initially set to true, the program proceeds to line 12.
- The true value of the loggedIn flag is returned, access is granted login access. The attacker can log in successfully, bypassing the need for a valid password by triggering the exception.

Exception handling: Example 1

Edinburgh Napier UNIVERSITY

The solution to this security vulnerability is straightforward:

simply reset the loggedIn flag to false in the event of any exception.

This adjustment is demonstrated in the revised code on line 10.

```
01: boolean Login(String user, String pwd) {
02:   boolean loggedIn = true;
03:   String realPwd = GetPwdFromDb(user);
04:   try {
05:      if  (!MessageDigest.getInstance("SHA-256").digest(pwd).equals(realPwd))
06:      {
07:         loggedIn = false;
08:      }
09:   } catch (Exception e) {
10:            loggedIn = false;
11:   }
12:   return loggedIn;
13: }
```

Exception handling: Example 2

Edinburgh Napier UNIVERSITY

Example 2: Too Much Information in Error Messages & Debugging Tools

- Error messages and debugging tools that reveal internal information publicly can be valuable to attackers in identifying and honing their exploits, and may even lead to serious data breaches.
- It is common to encounter detailed internal error messages like stack traces, error codes, and data dumps on websites when an error occurs.
- In a production environment, applications should never display stack traces or internal states through error messages or logs.
- Additionally, internal logs must have controlled access.
- Consider the following Java code example, which is used for checking username/password authentication. The original author faced difficulties in making it work. To diagnose the problem, the exception messages were altered to include actual data values. However, it's crucial to remember that in a production setting, username/password information is highly sensitive and should be handled with utmost care

Exception handling: Example 2

```
boolean Login(… user, … pwd) {
    try {
                ValidatePwd(user, pwd);
                return true;
    } catch (Exception e) {
                print("Login failed.\n");
                print(e + "\n");
                e.printStackTrace();
                return false;
    }
}
 void ValidatePwd(… user, … pwd)
                throws BadUser, BadPwd {
   realPwd = GetPwdFromDb(user);
   if (realPwd == null)
                throw BadUser("user=" + user);
    if (!pwd.equals(realPwd))
                throw BadPwd("user=" + user
                + " pwd=" + pwd
                + " expected=" + realPwd);
}
```

Exception handling: Example 2

Edinburgh Napier UNIVERSITY

- Ideal: never to log sensitive data, but for debugging purposes, this is not always feasible.
- A practical solution: to establish a secondary, secure logging repository.
- This repository should be accessible only to those with administrator privileges or under similar strict access controls.

```
boolean Login { try {
   ValidatePwd(user, pwd);
  } catch (Exception e) {
    logId = LogError(e); // write exception and return log ID.
    print("Login failed, username or password is invalid.\n");
    print("Contact support referencing problem id " + logId
       + " if the problem persists."); return false;
  }
  return true;
}

// (ValidatePwd is unchanged)
```

Exception handling: Example 2

Edinburgh Napier UNIVERSITY

In this method, sensitive information is directed to a protected storage, and the LogError() function provides a reference ID for it.

This ensures that only a developer authorized by the administrator can access the detailed information linked to this ID, significantly reducing the risk of it being acquired by an attacker.

It is important to use unpredictable and arbitrary identifiers as reference IDs.

Using sequential numbers could inadvertently reveal to attackers the frequency of these errors – not a major security breach, but even this small insight could assist an attacker in understanding the system's operations.

Exception handling: Example 2

In this method, sensitive information is directed to a protected storage, and the LogError() function provides a reference ID for it.

This ensures that only a developer authorized by the administrator can access the detailed information linked to this ID, significantly reducing the risk of it being acquired by an attacker.

It is important to use unpredictable and arbitrary identifiers as reference IDs.

Using sequential numbers could inadvertently reveal to attackers the frequency of these errors – not a major security breach, but even this small insight could assist an attacker in understanding the system's operations.

References

- OWASP Principles of Security: https://owasp.org/www-project-developer-guide/draft/foundations/security_principles/
- Slides from Yosef (Joseph) Mendelsohn on CSC 212 - Programming in Java II.
- SEI CERT Oracle Coding Standard for Java: https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java
- https://introcs.cs.princeton.edu/java/61data/ [last visited 29/1/2024]
- Elisa Heymann, Barton P. Miller and Loren Kohnfelder, Introduction to Software Security. https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/
- Seacord, Robert C. Secure Coding in C and C++. Addison-Wesley, 2013.

Excellent references for learning exceptions in Java:

- Sun's Java Tutorial, Exceptions Chapter
- Oracle reference: Lesson: Exceptions (The Java™ Tutorials > Essential Java Classes) (oracle.com)

Edinburgh Napier UNIVERSITY

QUESTIONS AND COMMENTS

- To contact me: a.sami@napier.ac.uk