9 Handling failures securely

This chapter covers

- Separating business and technical exceptions
- Removing security issues by designing for failure
- Why availability is an important security goal
- Designing for resilience for a more secure system
- Unvalidated data and security vulnerabilities

What is it that makes failures so interesting from a security perspective? Could it be that many systems reveal their internal secrets when they fail? Or is it how handling failure defines a system's level of security? Regardless, recognizing that failures and security go hand-in-hand is incredibly important when designing secure software. This, in turn, requires understanding what the security implications are when making certain design choices. For example, if you choose to use exceptions to signal errors, you need to make sure you don't leak sensitive data. Or when integrating systems, if you don't recognize the danger of cascading failures, you could end up with a system as fragile as a house of cards.

Regardless of which design choices you make, or why, you need to consider *failure*. The focus of this chapter isn't to tell you which design is better, but rather to give you insight into the security implications when making certain design choices. Also, the scope of failures is huge. To give you an idea of how complex the topic is, we'll show you examples ranging from low-level code constructs to high-level system design. All in all, this is a good starting point for learning how to handle failures securely. With that said, let's start with one of the most common design choices—exceptions.

9.1 Using exceptions to deal with failure

Exceptions are often used to represent failures because they allow you to disrupt the normal flow of an application program.[1] Because of this, it's common that exceptions carry information about why and where the execution flow was disrupted—the *why* is described in the message and the *where* by the stack trace. In listing 9.1, you see a stack trace resulting from a closed database connection. At first glance, it seems harmless, but if you look carefully, you'll see that it reveals information that you might want to keep secret. For example, the first line shows that the exception is a `java.sql.SQLException`. This tells you that data is stored in a relational database, and the system can be susceptible to SQL injection attacks. The same line also shows that the code is written in Java, which hints that the overall system might be vulnerable to exploits present in the language and the Java Virtual Machine (JVM).

Listing 9.1 Stack trace of a SQL exception when the database connection was closed

java.sql.SQLException shows Java is used.

SQLException indicates that data is stored in a relational database.

```
java.sql.SQLException: Closed Connection
  at oracle.jdbc.driver.DatabaseError...
  at oracle.jdbc.driver.DatabaseError.throwSqlException(...
  at oracle.jdbc.driver.PhysicalConnection.rollback(...
  at org.apache.tomcat.dbcp.dbcp.DelegatingConnection...
  at org.apache.tomcat.dbcp.dbcp.PoolingDataSource$
                    PoolGuardConnectionWrapper.rollback(...
  at net.sf.hibernate.transaction.JDBCTransaction...
...
```

net.sf.hibernate shows that Hibernate is used as an object relational mapper.[3]

org.apache.tomcat.dbcp shows that Apache Tomcat's database connection pool component is used.[2]

Obviously, the level of detail in a stack trace is meant for troubleshooting rather than sharing. But why is it that stack traces get revealed to the end user every now and then? The answer lies in a combination of sloppy design and not understanding why

[1] See the Oracle documentation on exceptions at https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html.

[2] See the Apache Commons DBCP documentation at https://commons.apache.org/proper/commons-dbcp/index.html.

[3] See "Hibernate ORM: What Is Object/Relational Mapping?" at http://hibernate.org/orm/what-is-an-orm/.

exceptions are thrown. To illustrate this, we'll walk you through an example where sensitive business information is leaked from the domain because of intermixing business and technical exceptions of the same type. The example also helps to demonstrate why it's important to never include business data in technical exceptions, regardless of whether it's sensitive or not.

9.1.1 Throwing exceptions

As illustrated in figure 9.1, there are three main reasons why exceptions are thrown in an application: business rule violations, technical errors, and failures in the underlying framework. All exceptions share the same objective of preventing illegal actions, but the purpose of each one differs. For example, *business exceptions* prevent actions that are considered illegal from a domain perspective, such as withdrawing money from a bank account with insufficient funds or adding items to a paid order. *Technical exceptions* are exceptions that aren't concerned about domain rules. Instead, they prevent actions that are illegal from a technical point of view, such as adding items to an order without enough memory allocated.

We believe separating business exceptions and technical exceptions is a good design strategy because technical details don't belong in the domain.[4] But not everyone agrees. Some choose to favor designs that intermix business exceptions and technical exceptions because the main objective is to prevent illegal actions, regardless of whether the illegality is technical or not. This might seem to be a minor detail, but intermixing exceptions is a door opener to a lot of complexity and potential security problems.

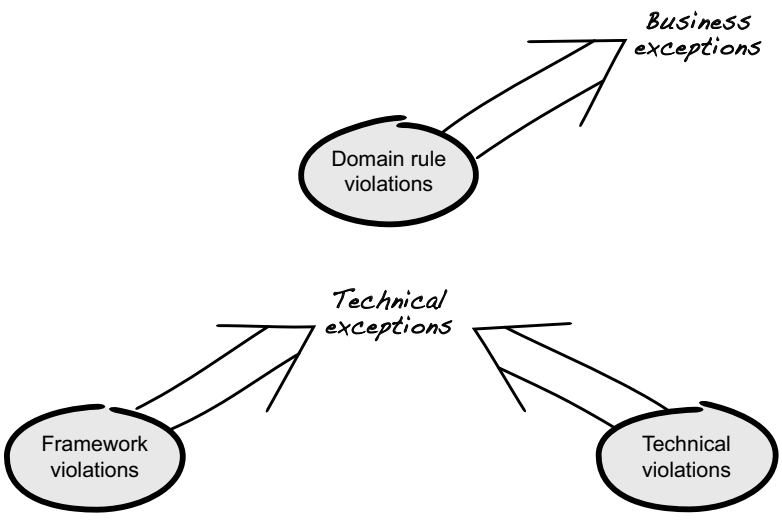

Figure 9.1 Three reasons for throwing exceptions in an application: domain rule violations, technical violations, and framework violations

[4] See Dan B. Johnsson's essay "Distinguish Business Exceptions from Technical" in *97 Things Every Programmer Should Know: Collective Wisdom from the Experts*, edited by Kevlin Henney (O'Reilly, 2010).

In listing 9.2, business and technical exceptions are intermixed using the same exception type. The main flow is fairly straightforward: a customer's accounts are fetched from a database, and the account matching the provided account number is returned. As part of this, an exception is thrown if no account is found or if an error occurs in the database.

Listing 9.2 Intermixing business and technical exceptions using the same type

```
import static java.lang.String.format;
import static org.apache.commons.lang3.Validate.notNull;

public Account fetchAccountFor(final Customer customer,
                               final AccountNumber accountNumber) {
   notNull(customer);
   notNull(accountNumber);

   try {
      return accountDatabase
            .selectAccountsFor(customer)
            .stream()
            .filter(account ->
                    account.number().equals(accountNumber))
            .findFirst()
            .orElseThrow(
              () -> new IllegalStateException(
                      format("No account matching %s for %s",
                             accountNumber.value(), customer)));
   } catch (SQLException e) {
      throw new IllegalStateException(
            format("Unable to retrieve account %s for %s",
                   accountNumber.value(), customer), e);
   }
}
```

Fetches the customer's accounts from the database

Selects only the accounts that match the provided account number

Throws an IllegalStateException if there's no matching account for the provided account number

Selects the first matching account because there can only be one matching account per account number

Translates a SQLException into an IllegalStateException with a specific message if there's an error in the database

The documentation of `IllegalStateException` specifies that it should be used to signal that a method has been invoked at an illegal or inappropriate time. It could be argued that not matching an account is neither illegal nor inappropriate and using an `IllegalStateException` is incorrect—a better choice might be `IllegalArgumentException`. But using `IllegalStateException` as a generic way of signaling failure is quite common, and we've decided to follow this pattern to better illustrate the problem of intermixing technical and business exceptions.

Throwing an exception when no account is found is logically sound, but is this a technical problem or a business rule violation? From a technical point of view, not matching an account is perfectly fine, but from a business perspective, you might want to communicate this to the user—for example, "Incorrect account number, please try again." This motivates having business rules around it, which makes the exception a business exception.

The second exception (thrown in the catch clause) is caused by a failing database connection or a malformed SQL query in the database. This also needs to be communicated, but not by the domain. Instead, you could rely on the surrounding framework to give an appropriate message—for example, "We're experiencing some technical problems at the moment, please try again later." This means the domain doesn't need rules for this exception, which makes it a technical exception. But how can you tell if you're dealing with a business or technical exception when both are of type `IllegalStateException`? Well, this is why you shouldn't intermix business and technical exceptions using the same type. But sometimes things are just the way they are, so let's find out how to handle this and learn what the security implications are.

Be careful using findFirst

The Java Stream API offers a rich set of functionality, where `findFirst` is a method that lets you short-circuit stream processing by selecting the first occurrence of an object. In listing 9.2, it's assumed that a one-to-one mapping exists between account and account number. Applying `findFirst` might then seem to be the natural choice, but this is where you need to be careful.

If you choose to use `findFirst`, it implies you don't care which element you choose as long as it exists. But that's not the case when fetching accounts: associating an account number with the correct account is imperative, and anything else is a disaster. The only reason that `findFirst` works in `fetchAccountFor` is because of the underlying relationship between account and account number. If this suddenly changes (either intentionally or because of a bug), the behavior of fetching accounts becomes random, and that's a hard bug to find!

A better solution is to use the Stream API's `reduce` method instead of `findFirst` to state the uniqueness assumption explicitly and to fail if multiple elements are found. The `reduce` operation is sometimes perceived as complex and hard to understand, but the essence is that it reduces the number of elements in a stream by applying an associative accumulation function to derive a new element. For example, summation can be expressed as `reduce((a, b) → a + b)`. This implies that `reduce` executes only if there are two or more elements present in a stream, and this is something you can use as a guarantee or contract. If `reduce` executes, you know the uniqueness requirement has been violated, but instead of reducing two elements into one, you throw an exception; for example, `reduce((accountA, accountB) → throw new IllegalStateException(...)`. This way, assumptions are stated explicitly, along with avoiding ambiguities and random behavior by design.

9.1.2 Handling exceptions

Handling exceptions seems easy at first; you surround a statement with a `try-catch` block and you're done. But when different failures use the same exception type, things get a bit more complicated. In listing 9.3, you see the calling code of the `fetchAccountFor` method in listing 9.2. Because you want to deal with only business exceptions in the

domain, you need to figure out how to distinguish between business exceptions and technical exceptions, even though both are of type `IllegalStateException`.

Unfortunately, you don't have much to go on, because both exceptions carry the same data. The only tangible difference is the internal message: the business exception message contains "No account matching," and the technical exception contains "Unable to retrieve account." This allows you to use the message as a discriminator and pass technical exceptions to a global exception handler that catches all exceptions, logs the payload, and rolls back the transaction due to technical problems.

Listing 9.3 Separating technical and business exceptions by message contents

```
import static org.apache.commons.lang3.Validate.notNull;

private final AccountRepository repository;

public Balance accountBalance(final Customer customer,
                              final AccountNumber accountNumber) {
   notNull(customer);
   notNull(accountNumber);
   try {
      return repository.fetchAccountFor(customer, accountNumber)
                       .balance();
   } catch (IllegalStateException e) {
      if (e.getMessage().contains("No account matching")) {
         return Balance.unknown(accountNumber);
      }
      throw e;
   }
}
```

Checks the internal message to determine if it's a business exception or not

Returns an unknown balance for the requested account number if the message matches

Propagates the exception further up the call stack if the message doesn't match

But what happens if you change the message or add another business exception with a different message? Won't that cause the exception to propagate out of the domain? It certainly will, and this is how sensitive data often ends up in logs or accidentally being displayed to the end user.

In listing 9.1, you saw how stack traces reveal information that doesn't make sense to show to a normal user. Instead, displaying a default error page with an informative message would be far better; for example, the message "Oops, something has gone terribly wrong. Sorry for the inconvenience. Please try again later." A global exception handler is often used for this purpose because it prevents exceptions from propagating to the end user by catching all exceptions. Different frameworks use different solutions for this, but the idea is the same. All transactions execute via a global exception handler, and if an exception is caught, the exception payload is logged and the transaction is rolled back. This way, it's possible to prevent exceptions from propagating further, which makes it a lot harder for an attacker to retrieve internal information when a transaction fails.

Let's turn back to the `accountBalance` method in listing 9.3. It's obvious you can't discriminate based on the exception message, because it makes the design too fragile.

Instead, you should separate business and technical exceptions by explicitly defining exceptions that are important for the business.

In listing 9.4, you can see an explicit domain exception (`AccountNotFound`) that signifies the event of not matching an account. The exception extends the generic type `AccountException`, which acts only as a marker type—a design decision that helps to prevent accidental business exceptions from leaking from the handling logic.

Listing 9.4 An explicit domain exception signifying that no account has been found

```
import static org.apache.commons.lang3.Validate.notNull;

public abstract class AccountException extends
                                RuntimeException {}

public class AccountNotFound extends
                                AccountException {
   private final AccountNumber accountNumber;
   private final Customer customer;

   public AccountNotFound(final AccountNumber accountNumber,
                          final Customer customer) {
      this.accountNumber = notNull(accountNumber);
      this.customer = notNull(customer);
   }
   ...
}
```

Generic domain type that all account exceptions extend

Explicit domain exception that signifies that no account has been found

In listing 9.5, the `fetchAccountFor` method is revised to use the `AccountNotFound` exception instead of a generic `IllegalStateException`. This way, the code is clarified in the sense that you don't need to provide a message or worry about intermixing its purpose with other exceptions.

Listing 9.5 Explicitly defining a domain exception to signal that no account is found

```
import static java.lang.String.format;
import static org.apache.commons.lang3.Validate.notNull;

private final AccountDatabase accountDatabase;

public Account fetchAccountFor(final Customer customer,
                               final AccountNumber accountNumber) {
   notNull(customer);
   notNull(accountNumber);

   try {
      return accountDatabase
            .selectAccountsFor(customer).stream()
            .filter(account -> account.number().equals(accountNumber))
            .findFirst()
            .orElseThrow(() ->
               new AccountNotFound(accountNumber,customer));
```

Replaces the generic IllegalStateException with an explicit domain exception

```
   } catch (SQLException e) {
      throw new IllegalStateException(
            format("Unable to retrieve account %s for %s",
                  accountNumber.value(), customer), e);
   }
}
```

In listing 9.6, the handling logic is revised to catch the exceptions `AccountNotFound` and `AccountException`. From a security perspective, this is much better because it allows less complex mappings between business rules and exceptions, compared with using only generic exceptions such as `IllegalStateException`. Catching `AccountException` seems redundant, but this safety net is quite important. Because all business exceptions extend `AccountException`, it's possible to guarantee that all business exceptions are handled and that only technical exceptions propagate to the global exception handler.

Listing 9.6 Revised handling logic with explicit domain exception

```
import static java.lang.String.format;
import static org.apache.commons.lang3.Validate.notNull;

private final AccountRepository repository;

public Balance accountBalance(final Customer customer,
                              final AccountNumber accountNumber) {
   notNull(customer);
   notNull(accountNumber);

   try {
      return repository.fetchAccountFor(customer, accountNumber)
                       .balance();
   }
   catch (AccountNotFound e) {
      return Balance.unknown(accountNumber);
   }
   catch (AccountException e) {
      throw new IllegalStateException(
            format("Unhandled domain exception: %s",
                  e.getClass().getSimpleName()));
   }
}
```

Handles AccountNotFound exception explicitly without parsing the internal message

Catches all unhandled business exceptions

Signals that an unhandled domain exception has been detected and that the transaction should be aborted

Separating business exceptions and technical exceptions clearly makes the code less complex and helps prevent accidental leakage of business information. But sensitive data isn't leaked only through unhandled business exceptions. It's often the case that business data is included in technical exceptions for debugging and failure analysis as well; for example, in listing 9.5, the `SQLException` is mapped to an `IllegalStateException` that includes the account number and customer data, which are needed only during failure analysis. To some extent, this counteracts the work of separating business and technical exceptions, because sensitive data leaks regardless. To address this issue, you need a design that enforces security in depth—so let's have a look at how to deal with exception payload.

9.1.3 Dealing with exception payload

There are two parts of an exception that are of special interest when analyzing failures: the type and the payload. By combining type information and payload data, you get an understanding of what failed and why it failed. As a consequence, many developers tend to include lots of business information in exceptions, regardless of how sensitive it is. For example, in the next listing, you see a snippet from the `fetchAccountFor` example where an `IllegalStateException` is populated with the account number and customer data, even though it's a technical exception.

Listing 9.7 Including sensitive data in a technical exception

```
import static java.lang.String.format;

catch (SQLException e) {
   throw new IllegalStateException(
         format("Unable to retrieve account %s for %s",
             accountNumber.value(), customer), e);
}
```

Leaks account number and customer data from the business domain when logged by the global exception handler

Having the account number and customer data during failure analysis certainly helps, but from a security perspective, you have a major problem. All technical exceptions propagate to the global exception handler that logs all exception data before rolling back the transaction. This means that the account number and customer data, like Social Security number, address, and customer ID, get logged when a database error occurs—a major security problem that requires logs to be placed under strict access and authorization control. And this isn't what you want when developers need access to production logs.

Obviously, you don't want sensitive data to escape the business domain, but sometimes it's hard to recognize what's sensitive. Exceptions can travel across context boundaries, and insensitive data in one context could become sensitive when entering another context. For example, a car's license plate number tends to be seen as public information, but if someone runs your car's plate against the Department of Motor Vehicles database to identify you, it suddenly becomes information you don't want to share. This puts you in a difficult position. On one hand, you need enough information to facilitate failure analysis; on the other hand, you want to prevent data leakage. How does this affect the design?

To start with, you need to recognize that almost any business data is potentially sensitive in another context. This means it's good design practice to never include business data in technical exceptions, regardless of whether it's sensitive or not. Also, you need to make sure to provide only information that makes sense from a technical perspective; for example, "Unable to connect to database with ID XYZ," instead of the account number and customer data that caused the failure. This way, you know that it's safe to propagate technical exceptions from the domain and that the payload never contains sensitive business data.

But following this practice gets you only halfway. You also need to identify sensitive data in your domain and model it as such. In chapter 5, you learned about the *read-once* pattern, which prevents data from being read multiple times and accidentally serialized; for example, when sent over the network or written to log files. If the account number and customer data had been modeled as sensitive and the read-once pattern used, illegal logging in the global exception handler would have been detected.

The choice of using exceptions to represent technical errors and valid results primarily opens the door to data leakage problems. By separating business and technical exceptions along with using the read-once pattern, it's possible to solve this—but is it the best solution, or is there another way? Perhaps, so let's evaluate how to handle failures without exceptions and see what security benefits there are.

9.2 Handling failures without exceptions

Using exceptions to represent failures in domain logic is a common practice, but another equally common approach is to not use exceptions at all. This approach starts with the design mindset that failures are a natural and expected outcome of anything we do. Because exceptions represent something exceptional (the name kind of gives it away) and failures are expected outcomes, it doesn't make sense to model them as exceptions. Instead, a failure should be modeled as a possible result of a performed operation in the same way a success is. By designing failures as unexceptional outcomes, you can avoid several of the problems that come from using exceptions—including ambiguity between domain and technical exceptions, and inadvertently leaking sensitive information.

If you look at the logic you implement in an application, it quickly becomes obvious that it's not only about happy cases. When you execute a method, you have an intention of performing a specific action. Performing that action can almost always have multiple outcomes. At the very least, it can succeed or it can fail. In this section, you'll learn how to gain more security by designing failures without using exceptions.

To explain this design approach, let's illustrate the difference between designing failures as exceptions versus designing them as expected outcomes. We'll do this by solving the same task using the two different approaches. The task is to implement a system to transfer money between bank accounts. In the current domain of banking, a money transfer can have two possible outcomes: either the transaction is performed or it fails due to insufficient funds (figure 9.2).

9.2.1 Failures aren't exceptional

If you choose to design using exceptions, your implementation will look something like listing 9.8. The method to transfer money from one bank account to another is called `transfer` and takes two arguments: the amount to transfer and the destination account. The first thing that needs to be done in the `transfer` method is to check that there are enough funds in the source account to cover the transfer. If the source account is lacking funds, an `InsufficientFundsException` is thrown; in which case, the exception needs

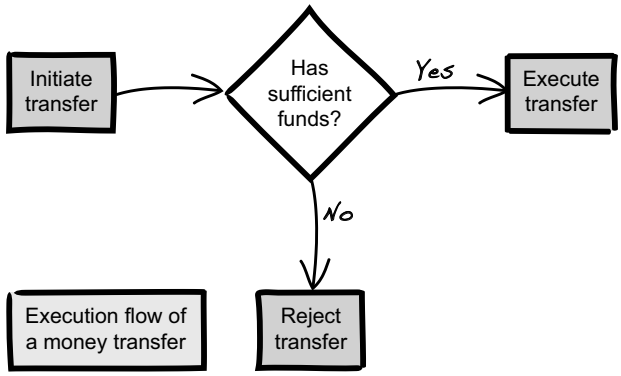

Figure 9.2 The possible outcomes of a money transfer between bank accounts

to be handled appropriately—perhaps by asking the user to adjust the amount or abort the transaction. If sufficient funds are available, you can execute the transfer by calling another backend system through the `executeTransfer(amount, toAccount)` method. This method can also throw exceptions in case of a failure. If the `executeTransfer` method is successful, nothing more happens and the code calling `transfer` continues to execute as normal.

Listing 9.8 Using exceptions for business logic

```
import static org.apache.commons.lang3.Validate.notNull;

public final class Account {

  public void transfer(final Amount amount,
                       final Account toAccount)
        throws InsufficientFundsException {
    notNull(amount);
    notNull(toAccount);

    if (balance().isLessThan(amount)) {
      throw new InsufficientFundsException();
    }

    executeTransfer(amount, toAccount);
  }

  public Amount balance() {
      return calculateBalance();
  }

  // ...
}

-------------------------------------------------
```

Checks whether there are sufficient funds

Throws an exception if there aren't enough funds

Calls underlying systems to execute the transfer; may also throw exceptions

```
import static org.apache.commons.lang3.Validate.isTrue;
import static org.apache.commons.lang3.Validate.notNull;

public final class Amount implements Comparable<Amount> {
   private final long value;

   public Amount(final long value) {
      isTrue(value >= 0, "A price cannot be negative");
      this.value = value;
   }

   @Override
   public int compareTo(final Amount that) {
      notNull(that);
      return Long.compare(value, that.value);
   }

   public boolean isLessThan(final Amount that) {
      return compareTo(that) < 0;
   }

   // ...

}
```

With this approach, you're handling the flow of your business logic using two mechanisms in the programming language. One is the result from calling a method (in this case, the result is `void`), and the other is the exception mechanism. You're using the exception mechanism of the programming language as part of your control flow.

Let's stop for a second and think about the semantics of using exceptions as a control flow in the `transfer` method. By doing so, you're saying that not having sufficient funds in the source account is an exceptional occurrence. You're treating the negative result as something exceptional. This is a common way of designing code, but there's an alternative way to view failures. This alternative way stems from the perspective that failures aren't exceptional but rather an expected outcome of any task you try to perform.

9.2.2 Designing for failures

In banking, it's not uncommon for users to try to initiate a transfer of an amount that's larger than the current account balance. This can, for example, happen if the amount is entered incorrectly, or if the user thinks they have more money in the account than they actually have. For whatever reason, it's relatively common for a money transfer to fail due to insufficient funds. Not having sufficient funds is therefore an expected outcome of the operation "trying to transfer money." Because it's an expected outcome, it shouldn't be modeled as an exceptional one. Rather, it should be modeled as a possible result of the action.

If you redesign the `transfer` method from listing 9.8 so that insufficient funds are an expected outcome, you'll have something like that in listing 9.9. This new method won't throw any exceptions as part of the logical flow. Instead, it returns the result of the transfer operation. The result can either be a success or a failure, and in the case

of a failure, it's possible for the calling code to find out what type of failure it was by inspecting the result. If there aren't enough funds, an `INSUFFICIENT_FUNDS` failure is returned. Otherwise, the method continues and will try to execute the transfer via the `executeTransfer(amount, toAccount)` method. The `executeTransfer` method also returns a result that can either be a success or a failure due to difficulties in executing the transfer. When the `executeTransfer` method finishes, the money transfer either succeeds or fails, with a failure message indicating the reason for failure.

Listing 9.9 Expected results not modeled as exceptional

```
import static Result.Failure.INSUFFICIENT_FUNDS;
import static Result.success;
import static org.apache.commons.lang3.Validate.notNull;

public final class Account {

  public Result transfer(final Amount amount,
                         final Account toAccount) {
     notNull(amount);
     notNull(toAccount);

     if (balance().isLessThan(amount)) {
        return INSUFFICIENT_FUNDS.failure();
     }

     return executeTransfer(amount, toAccount);
  }

  public Amount balance() {
     return calculateBalance();
  }

  // ...
}

-------------------------------------------------

import java.util.Optional;

public final class Result {

   public enum Failure {
      INSUFFICIENT_FUNDS,
      SERVICE_NOT_AVAILABLE;

      public Result failure() {
         return new Result(this);
      }
   }

   public static Result success() {
      return new Result(null);
   }
```

Checks whether there are sufficient funds

Returns the failure as a result instead of throwing an exception

Returns the result of calling the underlying systems to execute the transfer

Different types of possible failures

```
    private final Failure failure;

    private Result(final Failure failure) {
        this.failure = failure;
    }

    public boolean isFailure() {
        return failure != null;
    }

    public boolean isSuccess() {
        return !isFailure();
    }

    public Optional<Failure> failure() {
        return Optional.ofNullable(failure);
    }
}
```

It's worth pointing out that the `Result` shown in listing 9.9 is a basic implementation. Once you start using result objects, you'll probably find that you want to design them in certain ways to make them easy to work with and error-free. As an example, if you're using a functional style of programming, you might want to add the ability to perform operations such as map, flatmap, and reduce to simplify dealing with results. Exactly how you choose to design them is up to you or your team.

By designing failures as expected and unexceptional outcomes, you completely eliminate the use of exceptions as part of the domain logic. By doing so, you're able to either avoid or reduce the risk of many of the security issues you faced when designing your code with exceptions. Some of the security benefits of this approach are listed in table 9.1.

Table 9.1 Security benefits of designing failures as expected outcomes

Security issue	Solved through
Ambiguity between domain exceptions and technical exceptions	Domain exceptions are completely removed.
Exception payload leaking into logs	Failures aren't handled by generic error-handling code, and, therefore, the data the payload carries doesn't accidentally slip into error logs.
Inadvertently leaking sensitive information	Failures are handled in a context that has knowledge about what's sensitive and what's not and knows how to handle sensitive data properly.

In our experience, another benefit of treating failures as unexceptional is that once you start designing both successes and failures as results, the only exceptions that can still occur are those caused by either bugs or a violation of an invariant.

So far, you've seen how to handle failures in a secure way on a code level by either using exceptions or designing your failures as unexceptional. In the next section, we'll

discuss more high-level designs to show you how you can use design principles commonly used for resilience to gain security benefits.

9.3 Designing for availability

The availability of data and systems is an important security goal and is part of the CIA acronym (confidentiality, integrity, and availability).[5] The National Institute of Standards and Technology (NIST) publication "Engineering Principles for Information Technology Security" talks about five different goals for IT security: confidentiality, availability, integrity, accountability, and assurance.[6] It defines *availability* as the "goal that generates the requirement for protection against intentional or accidental attempts to (1) perform unauthorized deletion of data or (2) otherwise cause a denial of service or data." In this section, you'll learn about design concepts that improve the availability of a system—concepts you can use to create more secure systems.

We've gathered some well-known and commonly used concepts that promote availability, and it's our belief that they're also some of the most important and foundational principles on the subject. We could easily write an entire book on how to build systems that are robust and that will stay available even when experiencing failures. Going into great depth on each concept is beyond the scope of this book, but it's our intention to provide you with enough knowledge to understand each one and how they promote the security of a system. Once you see the connection to security for the concepts described, they will become even more valuable as guiding design principles.

9.3.1 Resilience

It's becoming increasingly common to design and build systems to be *resilient*. A system that's resilient is designed to stay functional even during high stress. Stress for a system can be caused by both internal failures (such as errors in code or failing network communication) and external factors (such as high traffic load). Stress can cause a resilient system to slow down or run with reduced functionality, and parts of the system can crash, but the system as a whole will stay available, and it'll recover once the stress it's been put under disappears.

Another way to describe a system that's resilient is to say it's stable. You can design stable systems in several ways (some of which you'll learn about in this chapter), but the main goal of a resilient system is to survive failures and continue to provide its service. Put differently, a resilient system is a system that stays available in the presence of failures.

Because availability is an explicit security goal, and a resilient system stays available during failures, a system that is resilient must also by definition be a more secure system. This, in turn, leads to the conclusion that all contemporary and well-established design practices that promote the resilience and stability of a system are also beneficial to use when designing secure systems.

---

[5] Go back to chapter 1 for more details on CIA.

[6] NIST Special Publication 800-27 Rev A, "Engineering Principles for Information Technology Security (A Baseline for Achieving Security)," by Gary Stoneburner, Clark Hayden, and Alexis Feringa. Available at https://csrc.nist.gov/publications/detail/sp/800-27/rev-a/archive/2004-06-21.

9.3.2 Responsiveness

Say you've built a system that's resilient and stays available during high stress. It doesn't crash, but when the load on the system becomes high enough, the response times increase dramatically. When the system is available but responds slowly, another system calling the slow system eventually gets a response, but it can take an unacceptably long time. From the caller's point of view, the system under load can be considered to be unusable, even though it's technically still available. This is where responsiveness comes in as an important trait when discussing availability.

For a system to be *responsive*, it not only needs to survive stress but also has to respond quickly to anyone trying to use it during times of stress. Even if you've optimized the processing logic as much as possible to make the system run faster, the system will still have a threshold for how much stress it can handle before the response times go through the roof. When this happens, you might wonder how you can possibly make the system respond any faster. In this situation, it's important to realize that to stay responsive, it's far better to answer quickly with an error saying that the system is unable to accept any more requests than to have the caller sit around waiting for an answer that might never come. Any answer is better than no answer, even if that answer is rejecting the request.

To make the system more responsive without rejecting requests, you could, for example, place all the processing work in a queue. Separating the requests for processing and the actual processing makes the system more asynchronous. This way, even if the work queue is growing because of a high load and the requested work takes a long time to finish, the system will be able to accept new requests. The caller gets a fast response saying that the request has been accepted, but it'll have to wait before the result of the work is available. The work queue might eventually get full; in which case, you'll have to decide how to handle that situation—possibly by denying more work to be queued and asking the caller to try again later.

Staying responsive is important for security, because for a system to be truly available, it's not enough that it be resilient and survive stress, it's also necessary for it to continue to respond quickly. How quickly it needs to respond depends on the system you're building and the maximum acceptable response time before the system is considered to be unavailable.

9.3.3 Circuit breakers and timeouts

A useful design pattern when building resilient systems is the *circuit breaker pattern*.[7] This pattern is handy for dealing with failures in a way that promotes system resilience, responsiveness, and overall availability—and therefore, also security.

The general idea of the circuit breaker pattern is to write code that protects a system from failures in the same way that an electrical fuse protects a house in the case of failure in the electrical system. A fuse is designed to break the electrical circuit if an

[7] See Michael T. Nygard's *Release It! Design and Deploy Production-Ready Software* (The Pragmatic Bookshelf, 2007).

excessive load is placed on the system (for example, by a faulty appliance or a short-circuit somewhere). If the circuit doesn't break, a high electrical current can generate so much heat that a fire can start. By breaking the circuit, and thereby isolating or stopping the failure, it's possible to prevent the entire house from burning down.

In the same way that a fuse protects a house, a circuit breaker in software can isolate failures and prevent an entire system from crashing. Just as an electrical current passes through a fuse during normal operation, you use a circuit breaker to protect your system by having your method calls or requests to other systems pass through it. Figure 9.3 shows an example of a rudimentary circuit breaker.

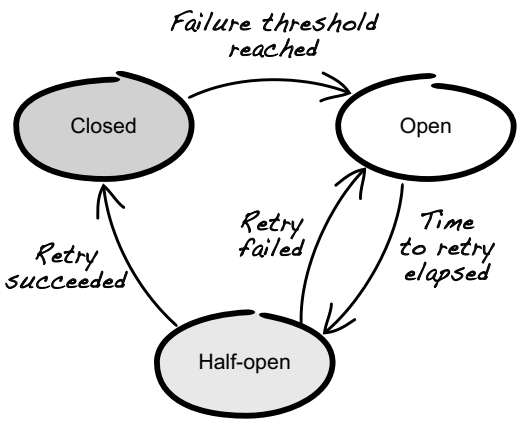

Figure 9.3 The three states of a circuit breaker

Depending on which state the circuit breaker is in, the request is handled differently. If it's in the closed state, any request performed passes through it. If the request completes successfully, nothing more happens. If the request fails, it'll increment a failure counter. If multiple subsequent calls fail, then the failure count eventually reaches a threshold that triggers the circuit breaker to open. Once it's in its open state, any new requests won't pass through but will fail immediately by the circuit breaker.

When a circuit breaker is open, it's effectively applying the fail-fast pattern instead of letting the requests pass through. If a circuit breaker is open, after a while it'll transition into a half-open state. In the half-open state, it can let one or more requests pass through to see if they'll succeed. In the event of success, the circuit breaker can return to its closed state and let all requests pass through. In the event of failure, the circuit breaker goes back to its open state until it's ready to let another trial request through.

When you make a call to another service, it's important to specify a timeout for that request. If an integration point is unresponsive, you don't want your application to hang forever, waiting for a response, because that eventually makes your system unstable. Because both timeouts and circuit breakers deal with protecting a system when making requests to other systems, they typically go hand-in-hand. It's common for implementations of circuit breakers to track timeouts separately and sometimes to even provide built-in functionality for managing timeouts for requests.

Always specify a timeout

In Java, for example, the default timeout for a network call is infinite—meaning that if a timeout isn't explicitly set, a network call waits forever for a response. As a result, the number of systems that have turned into unresponsive memory hogs due to unresponsive integration points is almost infinite too. Whatever environment you're in, whatever programming language or framework you use, always make sure you explicitly set a timeout for all your network requests.

Circuit breakers are typically used when making inter-process requests from one service to another, but they can also be used within the same service if it makes sense to do so. What makes the circuit breaker pattern so effective is that it works well for preventing failures from spreading from one part of a system to another. Isolating the failures and offloading the part of the system currently experiencing stress increases the stability of the system. Through the use of the fail-fast approach, it also improves the responsiveness of the system. Because circuit breakers promote both the resilience and the responsiveness of a system, this is an effective design pattern for improving the security of a system by increasing its availability.

A NOTE ON CIRCUIT BREAKERS AND DOMAIN MODELING

When using circuit breakers, it's common to use a default answer when a call fails. This is sometimes called a *fallback answer*. This pattern is quite effective and allows systems to continue to function despite failures, albeit with reduced or limited functionality.

One thing that's important to remember is that the way the system should behave if a request fails is usually a decision that affects your domain logic, and, therefore, it's a decision that needs to be made together with the domain experts. For example, if you're unable to check the inventory for an item when a customer places an order, do you refuse to take the order and lose a sale, or do you continue processing the order and deal with the unlikelihood of the item being out of stock? The answer is that it depends on how the business wants to handle this scenario. Another example is if you need to get a list of all books written by a particular author. Is it OK to return an empty list if the remote service you need to call is down? Sometimes that might be acceptable, but at other times, it's necessary to convey the failure so the client can distinguish a failure from the fact that no books exist for a given author.

TIP When you use default or fallback answers with circuit breakers, make sure to involve the domain experts. Then model and design your failure-handling code as you would any other business logic.

The approach of designing for failures that you read about earlier in this chapter is a great way of handling these scenarios. It forces you to handle failures within your domain logic rather than as part of your infrastructure logic.

Finally, circuit breakers are typically a design tool brought into a codebase by developers, and it's easy to think they are only relevant from a technical perspective. More

often than not, the opposite is true. We encourage you to involve your domain experts and stakeholders in the use of circuit breakers so you can design systems that not only technically stay available, but also continue to work as intended from a business perspective.

9.3.4 Bulkheads

The bulkhead design pattern is another tool you can use to efficiently prevent failures in one part of a system from spreading and taking down the entire system. Bulkheads can be applied as both a high-level design pattern when architecting infrastructure (such as servers, networks, routing of traffic, and so on) and a low-level design pattern for designing resilient code. Because bulkheads are so commonly used and do such a good job of isolating failures, it's a pattern that you should become familiar with to create systems with a high degree of availability.

In ship building, the term *bulkhead* refers to a wall or panel used to compartmentalize the hull into sections that are sealed against both water and fire (figure 9.4). Constructing a ship with these types of compartments means that if a water leak or fire were to occur in one part of the ship, the bulkheads would prevent the ship from taking on too much water and sinking or the fire from spreading.

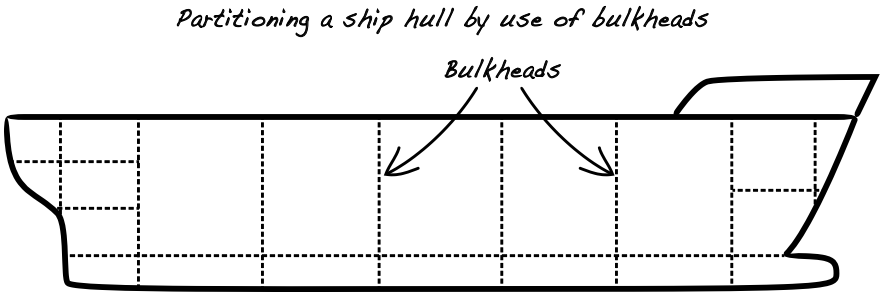

Figure 9.4 A ship hull constructed with bulkheads

In software, the same design techniques can be used to build resilient systems. One thing that's interesting with the bulkhead pattern is that it can be applied on different levels in your architecture. Let's take a look at each of these levels so you can get an idea of various ways to apply this pattern.

LOCATION LEVEL

At a high level, a system's availability can be improved by running the system on servers distributed over multiple geographical locations. If one location becomes inoperable—perhaps because of a power outage, a network fiber dug up during construction work, or an earthquake—then the other location or locations will still be available to provide the service. When deploying systems this way, you typically design each location to be completely self-sustained so that no interdependencies exist. How you choose the geographic locations depends on the business requirements, but they can be anything from different parts of a town to different parts of the world.

INFRASTRUCTURE LEVEL

Zooming in a bit, you can also apply the bulkhead pattern when designing your system infrastructure. For example, if you have a backend for a webshop, you can have one set of servers handling the load of customers browsing products and adding them to the shopping cart and another set of servers handling the checkout and payment flow, as seen in figure 9.5.

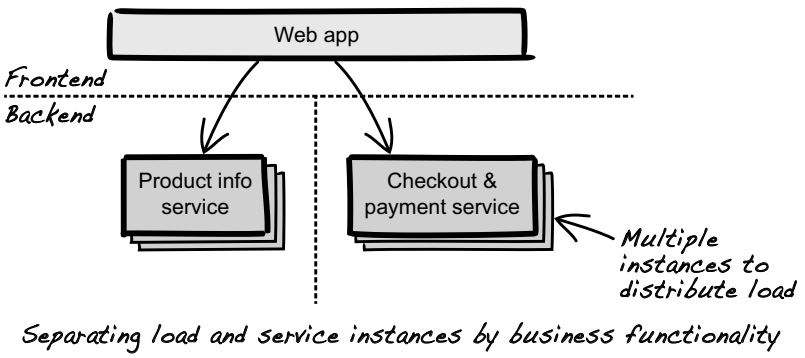

Figure 9.5 Protecting business functionality by partitioning workload

By partitioning the workload on your backend servers, you separate different areas of business functionality so they don't affect each other. (You could also choose to partition your frontend servers, but we'll disregard that in this example.) For example, a release of a popular product that you sell could potentially generate so much traffic that the servers handling product information start to slow down due to the high load. But because the checkout process is handled by a different set of servers, the ongoing sales aren't affected by the high demand on product information. The use of bulkheads ensures that the reduction or loss of availability in one part of the system doesn't affect the availability of another part. Service-oriented architectures are typically a good fit for applying bulkheads in this manner.

One thing to watch out for when partitioning services and servers is hidden dependencies. If you have multiple services using the same database, for example, then one service can cause slow responses or deadlocks in the database, which in turn can reduce the availability of another service. In this scenario, you've failed to properly apply the bulkhead by not separating the persistence solution. Other common hidden dependencies are message queues; network storage, like storage area networks (SANs) and network-attached storage (NAS); and shared network infrastructure, like routers and firewalls. It's also common to see bulkheading applied via use of virtualization technologies, such as containers and virtual machines. These technologies are great, but if you run everything on the same physical hardware, you've managed to create not only a lot of complexity, but also a hidden dependency. If the hardware crashes, it doesn't matter how many containers you partitioned your system on—they'll all go down together.

CODE LEVEL

You can also use bulkheads when designing your code. A common example of the bulkhead pattern applied on a code level is thread pools. The reason this is common is that if you let your code create an unlimited number of threads, your application inevitably grinds to a halt and possibly crashes. An easy and effective way to limit the number of threads is to use a thread pool. A thread pool lets you set a limit on how many threads the code can create. You can then use the threads in the pool to process work. Regardless of how much work needs to be processed, there'll never be more threads than are allowed in the pool. You also have the benefit of reusing threads in the pool instead of constantly creating new ones. Request pools in web servers and connection pools for databases are typical real-world uses of thread pools that you might have encountered before.

Queues are another code construct you can use in order to isolate failures in your code base. Queues are often used together with thread pools. If all the threads in the pool are busy, additional work can be put in a queue. As soon as a thread becomes available in the pool, the queue can be queried for work to be processed. If work is added to the queue at a higher rate than the thread pool can process, the queue grows in size. If this continues, the queue eventually becomes full and, at that point, the application can refuse to accept any new work.

Going back to the example of the webshop backend that provides product information and processes orders, you can write your code to use different thread pools and queues for different types of work. If one thread pool becomes so busy it needs to put work in a queue for later processing, the other thread pool can continue unaffected. Moreover, if the queue for fetching product information becomes full and the system starts to refuse more requests for product information, the queue for order processing can still accept new work and continue processing.

By using thread pools and queues as bulkheads, you're preventing the consumption of resources in one part of your code from affecting another part. If one part of your code becomes unavailable, other parts will remain available even if they're within the same service instance. This effectively increases the availability of your system.

The Reactive Manifesto

The Reactive Manifesto defines four important traits that need to be present for what it calls a reactive system: the system must be responsive, resilient, elastic, and message driven (http://reactivemanifesto.org). A *reactive system*, according to the manifesto, is a system that's "more robust, more resilient, more flexible, and better positioned" to meet the demands put on modern systems. But, as it turns out, the Reactive Manifesto is also interesting from a security perspective. Let's take a look at why.

The goal of the manifesto is to promote good design practices by creating a common, ubiquitous vocabulary to use when discussing modern system design. The manifesto talks about the four traits and what defines each of them, and it also discusses how to achieve them.

(continued)

The main focus of the manifesto is how to build systems that can live up to the demands put on them. Modern systems need to live up to far higher demands than their predecessors. They need to serve more data to more users and with shorter response times. Downtime should be minimal, and it's necessary for modern systems to be able to adapt to fluctuations in load. Reactive systems meet these demands and are also typically more modular in their design, which tends to make them easier to develop and to evolve.

A reactive system stays resilient and responsive during periods of high stress. It's designed to have a high degree of availability. This makes the Reactive Manifesto interesting from a security perspective, because availability is an important trait for secure systems. If you're designing your systems to be reactive, you not only are getting the benefits of scalability and high capacity, but you're also improving the security of the systems.

You've now learned that availability is an important security goal, and you've learned how you can improve the availability of a system by making it more resilient. You've also seen some common techniques for designing resilient and responsive systems. If you weren't familiar with them before, you now have a good starting point for learning more about building resilient systems. In any case, you've hopefully grokked the connection between resilient systems and security, and learned yet another reason for building systems that survive failures. Now it's time to take a look at how to avoid security flaws when dealing with bad data.

9.4 Handling bad data

When dealing with data, whether it's from a database, user input, or an external source, there's always a chance it'll be partially broken by having trailing spaces, missing characters, or other flaws that make it invalid. Regardless of the cause, how your code handles bad data is essential for security. In chapter 4, you learned about using contracts to protect against bad state and input that doesn't meet the defined preconditions. This certainly tightens the design and makes assumptions explicit, but applying contracts often leads to discussions about repairing data before validation to avoid unnecessary rejection. Unfortunately, choosing this approach is extremely dangerous because it can expose vulnerabilities and result in a false sense of security.

But modifying data to avoid false positives isn't the only security problem to consider. Another interesting issue is the urge to echo input verbatim in exceptions and write it to log files when a contract fails. This can be justifiable for debugging purposes, but from a security perspective, it's a ticking bomb waiting to explode. To see why, we'll guide you through a simple example of a webshop where it has been decided to expand the membership database with data from another system. Unfortunately, the data quality is poor, which makes the business decide to apply a repair filter before validation. This turns out to be a great mistake because, combined with echoing validation failures, it opens up security vulnerabilities such as cross-site scripting and second-order injection attacks. Let's see how this happens by starting with why you shouldn't blindly repair data before validation.

Cross-site scripting and second-order attacks

In a cross-site scripting (XSS) attack, the attacker sends malicious strings to a site, hoping that the site will repeat the same strings in the output on the page. For example, if a news site lets you search for words in articles, it might return "Cannot find an article containing Jane Doe" if you search for Jane Doe. But if a visitor searches for `<script>alert(0)</script>`, they'll get a result page saying "Cannot find an article containing" and at the same time cause the server to run some JavaScript that pops up an alert box. This doesn't sound so alarming, but an XSS attack might do something much nastier, like installing a keylogger, sending cookies to a remote server, or worse.

Unfortunately, even logs can be used as the starting point of an attack, and a browser-based admin tool used for viewing logs might have a vulnerability for specific formats. In that case, an attacker can cause an attack string to be logged and then wait for the admin to look at that log entry using the vulnerable tool. This is called a *second-order attack* because the attacker isn't attacking the system they face, but rather a second system behind it.

9.4.1 Don't repair data before validation

Picture a webshop where users sign up for a membership to get better deals. The registration form asks them to enter their name, address, and other information that's needed to create a membership. The domain model is well defined, and each term in the membership context has a precise meaning and definition. For example, in listing 9.10, you see that a name has a tight definition and is restricted to alphabetic characters (a-z and A-Z) and spaces, and a length between 2 and 100 characters. This seems a bit strict, but names containing special characters, such as Jane T. O'Doe or William Smith 3rd, are considered rare enough that the business has decided to require users to drop special characters instead of loosening the contracts; for example, Jane T. O'Doe needs to be registered as Jane T ODoe.

Listing 9.10 The name domain primitive

```
import static org.apache.commons.lang3.Validate.inclusiveBetween;
import static org.apache.commons.lang3.Validate.matchesPattern;
import static org.apache.commons.lang3.Validate.notBlank;

public final class Name {
   private final String value;

   public Name(final String value) {
      notBlank(value);
      inclusiveBetween(2, 100, value.length(),
          "Invalid length. Got: " + value.length());
      matchesPattern(value,"^[a-zA-Z ]+[a-zA-Z]+$",
                     "Invalid name. Got: " + value);
      this.value = value;
   }
   ...
}
```

A name can't be empty or null.

A valid name contains between 2 and 100 characters.

A name can only contain alphabetic characters (a-z and A-Z) and spaces.

But restricting names this way only worked well until it was decided to expand the membership database with data from another system, then the `Name` contracts blew up like fireworks on New Year's Eve. A failure investigation revealed that the quality of the new data was poor: some names were empty, others had special characters, and some contained < and > characters originating from an XML import that went bad a few years ago.

The preferred solution is to address this at the source, but modifying data to fit the membership context isn't as simple as it seems. This is because data is consumed by several systems, and making adjustments for one system (for example, removing special characters in a name) might not be acceptable for another. Consequently, the business decides to leave the data as it is in the database and apply a repair filter before it's validated in the membership context. This strategy turns out to be a great success, as it significantly reduces the frequency of unnecessary rejections in the membership context. In fact, the result is so good that it's decided to apply the filter for all types of input sources, as illustrated in figure 9.6.

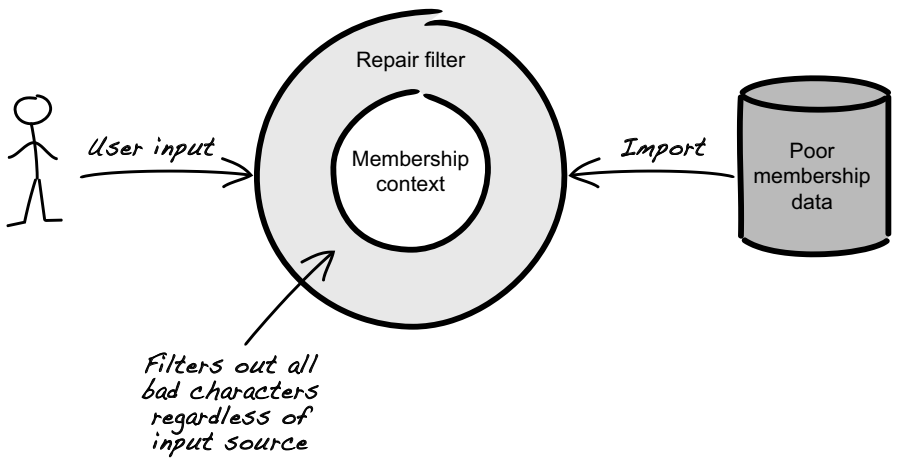

Figure 9.6 Bad characters filtered out for all data sources

Unfortunately, this is also when things start to get bad from a security perspective. To see how, you need to understand the relationship between the repair logic, validation, and failures, as shown in figure 9.7.

As illustrated, input is mutated every time it passes through the filter, and validation failures are echoed in the browser and log files. Although the data mutation is intentional, it also means the repair filter creates a derivative from the original input that could become dangerous. For example, consider the problem of cleaning up names with sporadic < and > characters. Applying a filter to remove them seems like the right thing to do; it creates a win-win situation by minimizing unnecessary rejection and avoiding XSS attacks by dismantling the `<script>` tag. Or at least, that's what many tend to believe. The

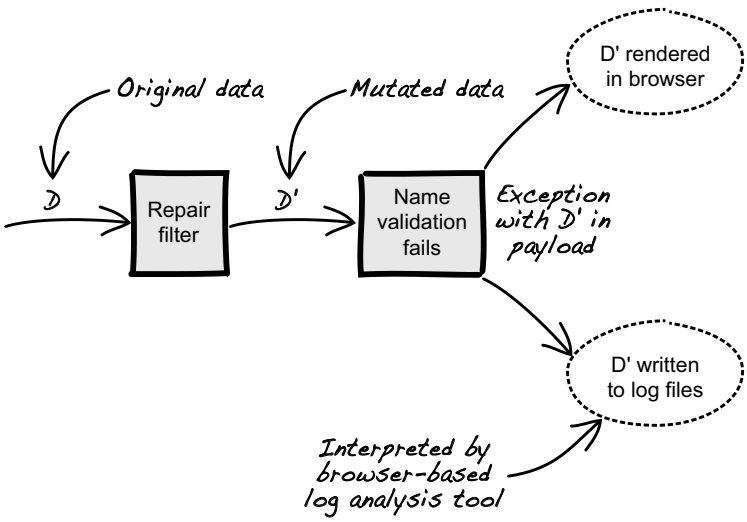

Figure 9.7 Relationship between repairing data and validation

truth is, dropping < and > only adds a false sense of security—it's still possible to launch an XSS attack. Consider injecting

```
%3<Cscript%3>Ealert("XSS")%3<C/script%3>E
```

to the repair filter.[8] Dropping the < and > characters yields

```
%3Cscript%3Ealert("XSS")%3C/script%3E
```

which is the same JavaScript code as `<script>alert("XSS")</script>`. The only difference is that < and > are expressed in hexadecimal. But passing JavaScript code to the membership context isn't dangerous per se unless it gets executed!

9.4.2 Never echo input verbatim

In listing 9.11, you see the validation logic applied in the `Name` constructor. When `%3Cscript%3Ealert("XSS")%3C/script%3E` is validated, the regular expression of `matchesPattern` fails and an error message is created. As developers, we often want to know why a contract failed—was it because of a programming error or invalid input? Consequently, many choose to echo input verbatim in error messages because it facilitates the failure analysis, but it could also expose vulnerabilities such as XSS and second-order attacks.

Listing 9.11 Validation applied in the `Name` constructor

```
Validate.notBlank(value);
Validate.inclusiveBetween(2, 100, value.length(),
             "Invalid length. Got: " + value.length());
```

[8] `<script>alert("XSS")</script>` is the classic way of testing if a system interprets input as data or code (JavaScript).

```
Validate.matchesPattern(value, "^[a-zA-Z ]+[a-zA-Z]+$",
                        "Invalid name. Got: " + value);
```

Input is echoed verbatim in the validation failure message.

By echoing the input verbatim in the validation failure message, the webshop practically allows attackers to control the output of the application, especially if exception payload is logged or displayed to the end user. It can seem harmless to log

```
%3Cscript%3Ealert("XSS")%3C/script%3E
```

but if log data is analyzed in a browser-based tool without proper escaping, `%3Cscript%3Ealert("XSS")%3C/script%3E` could be interpreted as code and executed. This simple example only results in an alert box popping up, but the mere fact that JavaScript is allowed to execute is extremely dangerous—an attacker could take advantage of this to install a keylogger, steal credentials, or hijack a session.

Although it sounds far-fetched, this kind of attack isn't unlikely. In chapter 3, you learned about the importance of context mapping and semantic boundaries. Injecting data with the intention of targeting vulnerabilities in a second system (a second-order attack) builds on the behavior of a broken context map, where data is misinterpreted only because it enters a different context. For instance, in our example, the JavaScript string only becomes harmful when interpreted as code in the log analyzer tool. Because of this, it can be difficult to determine whether it's OK to echo input or not—if you're unsure, play it safe and avoid doing so completely.

XSS Polyglots

Cross-site scripting (XSS) is an interesting type of attack because an attack vector can be crafted in an almost infinite number of ways. This makes it difficult to identify and to remove XSS flaws from a web application. The general recommendation is to do a security audit to find places where user input could end up in the HTML output, but the complexity of XSS makes it hard to guarantee that all places are found.* A complement could therefore be to test an application using an XSS polyglot, which is an attack vector that's executable in multiple contexts (places in the HTML where input is rendered as output). To illustrate, let's consider the following injection contexts:

- `<div class="{{input}}"></div>`
- `<noscript>{{input}}</noscript>`
- `<!--{{input}}-->`

An XSS polyglot is an attack vector that successfully executes a JavaScript (for example, `alert('XSS')`) in all three contexts, so let's see how to do this.

The first context is a `class` attribute in a `div` element. To allow script execution, you need to break out of the `class` attribute and close the `div` using a double quote and a greater than character (`">`). This is possible with a string that starts with `">` followed by a script. For example, injecting

```
"><svg onload=alert('XSS')>
```

results in an HTML string that looks like

```
<div class=""><svg onload=alert('XSS')>"></div>
```

This in turn creates an alert box with the message XSS when rendered in a browser. But to be an XSS polyglot, the attack vector must also apply to all other contexts as well.

The second context is a `<noscript>` block into which the attack vector is inserted. To allow for script execution, you need to break out of the context using a `</noscript>` tag. This results in

```
"></noscript><svg onload=alert('XSS')>
```

which is a slightly more complex attack vector that successfully executes the JavaScript.

The third context is within a comment block. To allow for script execution, the attack vector must contain `-->` before the script. Adding this to the existing vector results in

```
"></noscript--><svg onload=alert('XSS')>
```

which is an XSS polyglot that allows for script execution in all three contexts.

You've now learned how to create an XSS polyglot for three contexts presented by the XSS Polyglot Challenge (a contest that challenges you to create an XSS polyglot for up to 20 contexts using as few characters as possible).† But XSS polyglots are actually part of a bigger class of attacks called polyglot injections.‡

Polyglot attacks exploit the fact that many applications are implemented using several languages (for example Java, SQL, and JavaScript), which potentially makes them susceptible to attack vectors that combine these languages. For example, the attack vector

```
/*!SLEEP(1)/*/alert(1)/*/*/
```

combines SQL and JavaScript, which could exploit weaknesses in systems using these languages.§

* See "OWASP Cross-Site Scripting (XSS)" at https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

† See "XSS Polyglot Challenge" at https://polyglot.innerht.ml.

‡ See "Polyglots: Crossing Origins by Crossing Formats" at https://research.chalmers.se/publication/189673.

§ See "Polyglot Payloads in Practice," by Mathias Karlsson, at https://www.slideshare.net/MathiasKarlsson2/polyglot-payloads-in-practice-by-avlidienbrunn-at-hackpra.

By now, you've learned why failures need to be considered and how failure handling affects security. In the next chapter, we'll shift focus and explore several design concepts used in the cloud that use security; for example, immutable deployments, externalized configuration, and the three R's of enterprise security.

Summary

- Separating business exceptions and technical exceptions is a good design strategy because technical details don't belong in the domain.
- You shouldn't intermix technical and business exceptions using the same type.
- It's a good design practice to never include business data in technical exceptions, regardless of whether it's sensitive or not.

- You can create more secure code by designing for failures and treating failures as normal, unexceptional results.
- Availability is an important security goal for software systems.
- Resilience and responsiveness are traits that add security by improving the availability of a system.
- You can use design patterns like circuit breakers, bulkheads, and timeouts to design for availability.
- Repairing data before validation is dangerous and should be avoided at all costs.
- You should never echo input verbatim.