Chapter 22 - Security

Designing a Secure Object

Limiting Accessibility

Let's start with a terrible implementation.

```
package animals.security;
public class ComboLocks {
   public Map<String, String> combos;
}
```

This is terrible because the `combos` object has `public` access. This is also poor encapsulation. A key security principle is to limit access as much as possible. Think of it as “need to know” for objects. This is called the *principle of least privilege*.

It would be better to make the `combos` object `private` and write a method to provide the necessary functionality.

Remember, one good practice to thwart Hacker Harry and his cronies is to limit accessibility by making instance variables `private` or package-private, whenever possible. If your application is using modules, you can do even better by only exporting the security packages to the specific modules that should have access. Here's an example:

`exports animals.security to zoo.staff;`

Restricting Extensibility

Suppose you are working on a class that uses `ComboLocks`.

```
public class GrasshopperCage {
   public static void openLock(ComboLocks comboLocks, String combo) {
      if (comboLocks.isComboValid("grasshopper", combo))
         System.out.println("Open!");
   }
}
```

Ideally, the first variable passed to this method is an instance of the `ComboLocks` class. However, Hacker Harry is hard at work and has created this subclass of `ComboLocks`.

```
public class EvilComboLocks extends ComboLocks {
   public boolean isComboValid(String animal, String combo) {
      var valid = super.isComboValid(animal, combo);
      if (valid) {
         // email the password to Hacker Harry
      }
      return valid;
   }
}
```

This is great. Hacker Harry can check whether the password is valid and email himself all the valid passwords. Mayhem ensues! Luckily, there is an easy way to prevent this problem. Marking a sensitive class as `final` prevents any subclasses.

Creating Immutable Objects

an immutable object is one that cannot change state after it is created. Immutable objects are helpful when writing secure code because you don't have to worry about the values changing. They also simplify code when dealing with concurrency.

Although there are a variety of techniques for writing an immutable class, you should be familiar with a common strategy for making a class immutable.

1. Mark the class as `final`.
2. Mark all the instance variables `private`.
3. Don't define any setter methods and make fields final.
4. Don't allow referenced mutable objects to be modified.
5. Use a constructor to set all properties of the object, making a copy if needed

```
 import java.util.*;
2:
3:  public final class Animal {
4:     private final ArrayList<String> favoriteFoods;
5:
6:     public Animal() {
7:        this.favoriteFoods = new ArrayList<String>();
8:        this.favoriteFoods.add("Apples");
9:     }
10:    public List<String> getFavoriteFoods() {
11:       return favoriteFoods;
12:    }
13: }
```

We carefully followed the first three rules, but unfortunately, Hacker Harry can modify our data by calling `getFavoriteFoods().clear()` or add a food to the list that our animal doesn't like. It's not an immutable object if we can change it contents! If we don't have a getter for the `favoriteFoods` object, how do callers access it? Simple, by using delegate methods to read the data, as shown in the following:

```
...
10:    public int getFavoriteFoodsCount() {
11:       return favoriteFoods.size();
12:    }
13:    public String getFavoriteFoodsElement(int index) {
14:       return favoriteFoods.get(index);
15:    }
...
```

In this improved version, the data is still available. However, it is a true immutable object because the mutable variable cannot be modified by the caller. Another option is to create a copy of the `favoriteFoods` object and return the copy anytime it is requested, so the original remains safe.

```
10:    public ArrayList<String> getFavoriteFoods() {
11:       return new ArrayList<String>(this.favoriteFoods);
12:    }
```

So, what's this about the last rule for creating immutable objects? Let's say we want to allow the user to provide the `favoriteFoods` data, so we implement the following:

```
...
6:     public Animal(ArrayList<String> favoriteFoods) {
7:        if(favoriteFoods == null)
8:           throw new RuntimeException("favoriteFoods is required");
9:        this.favoriteFoods = favoriteFoods;
10:    }
...
```

He decides to send us a `favoriteFood` object but keep his own secret reference to it, which he can modify directly.

```java
void modifyNotSoImmutableObject() {
      var favorites = new ArrayList<String>();
      favorites.add("Apples");
      var animal = new Animal(favorites);
      System.out.print(animal.getFavoriteFoodsCount());
      favorites.clear();
      System.out.print(animal.getFavoriteFoodsCount());
}
```

This method prints `1`, followed by `0`. Whoops! It seems like `Animal` is not immutable anymore, since its contents can change after it is created. The solution is to use a *copy constructor* to make a copy of the list object containing the same elements.

```java
6:     public Animal(List<String> favoriteFoods) {
7:        if(favoriteFoods == null)
8:           throw new RuntimeException("favoriteFoods is required");
9:        this.favoriteFoods = new ArrayList<String>(favoriteFoods);
10:    }
```

The copy operation is called a *defensive copy* because the copy is being made in case other code does something unexpected.

Cloning Objects

Java has a `Cloneable` interface that you can implement if you want classes to be able to call the `clone()` method on your object. This helps with making defensive copies.

The `ArrayList` class does just that, which means there's another way to write the statement on line 9.

```java
9: this.favoriteFoods = (ArrayList) favoriteFoods.clone();
```

The `clone()` method makes a copy of an object. Let's give it a try by changing line 3 of the previous example to the following:

```java
public final class Animal implements Cloneable {
```

Now we can write a method within the Animal class:"

```java
public static void main(String… args) throws Exception {
   ArrayList<String> food = new ArrayList<>();
   food.add("grass");
   Animal sheep = new Animal(food);
   Animal clone = (Animal) sheep.clone();
   System.out.println(sheep == clone);
   System.out.println(sheep.favoriteFoods == clone.favoriteFoods);
}
```

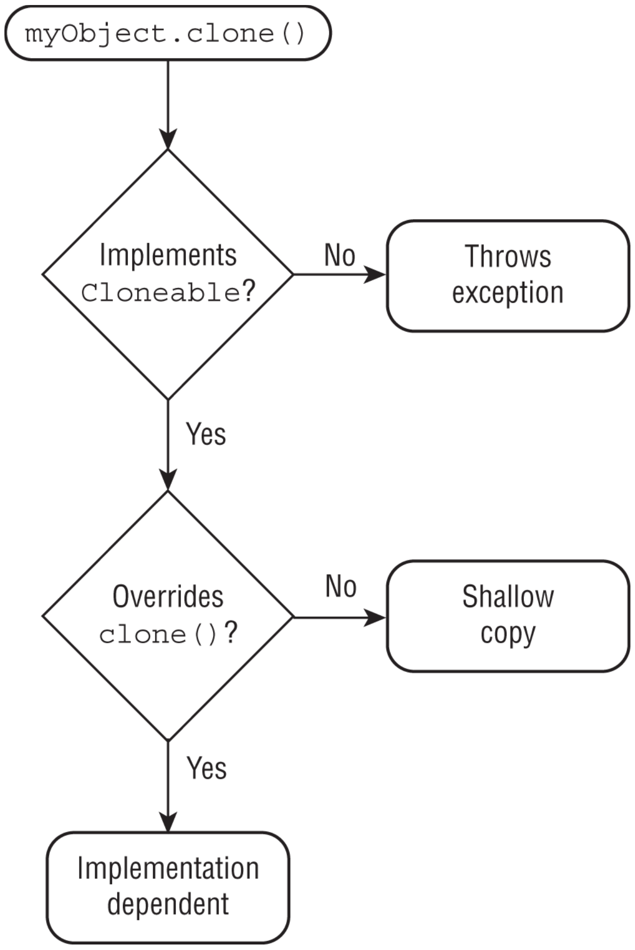
This code outputs the following:

```
false
true
```

By default, the `clone()` method makes a *shallow copy* of the data, which means only the top-level object references and primitives are copied. No new objects from within the cloned object are created.

By contrast, you can write an implementation that does a *deep copy* and clones the objects inside. A deep copy does make a new `ArrayList` object. Changes to the cloned object do not affect the original.

```java
public Animal clone() {
   ArrayList<String> listClone = (ArrayList) favoriteFoods.clone();
   return new Animal(listClone);
}
```



Introducing Injection and Input Validation

An *exploit* is an attack that takes advantage of weak security. Hacker Harry is ready to try to exploit any code he can find. He especially likes untrusted data.

Using *Statement*

```
public int getOpening(Connection conn, String day)
      throws SQLException {
   String sql = "SELECT opens FROM hours WHERE day = '" + day +"'";

   try (var stmt = conn.createStatement();
      var rs = stmt.executeQuery(sql)) {
      if (rs.next())
         return rs.getInt("opens");
   }
   return -1;
}
```

`int opening = attack.getOpening(conn, "monday");  // 10`

Then Hacker Harry comes along to call the method. He writes this:

```
int evil = attack.getOpening(conn,
   "monday' OR day IS NOT NULL OR day = 'sunday");  // 9
```

Hacker Harry's parameter results in the following SQL, which we've formatted for readability:

```
SELECT opens FROM hours
```

```
   WHERE day = 'monday'
      OR day IS NOT NULL
      OR day = 'sunday'
```

It says to return any rows where `day` is `sunday`, `monday`, or any value that isn't `null`. Since none of the values in Figure 22.2 is `null`, this means all the rows are returned. Luckily, the database is kind enough to return the rows in the order they were inserted; our code reads the first row.

Security Sienna shows us that we need to rewrite the SQL statement using bind variables like we did in Chapter 21.

This time, Hacker Harry's code does behave differently.

```
int evil = attack.getOpening(conn,
   "monday' or day is not null or day = 'sunday");  // -1
```

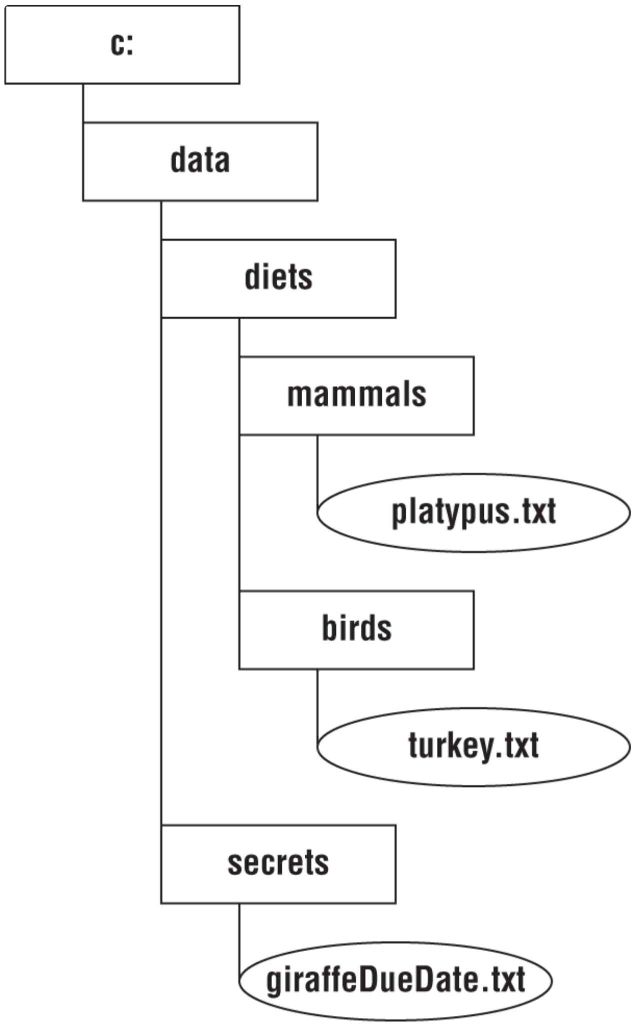
The entire string is matched against the `day` column. Since there is no match, no rows are returned. This is far better!

Invalidating Invalid Input with Validation

SQL injection isn't the only type of injection. *Command injection* is another type that uses operating system commands to do something unexpected.

The following code attempts to read the name of a subdirectory of `diets` and print out the names of all the `.txt` files in that directory:

```
Console console = System.console();
String dirName = console.readLine();
Path path = Paths.get("c:/data/diets/" + dirName);
try (Stream<Path> stream = Files.walk(path)) {
   stream.filter(p -> p.toString().endsWith(".txt"))
      .forEach(System.out::println);
}
```



Then Hacker Harry came along and typed `..` as the directory name.

```
c:/data/diets/../secrets/giraffeDueDate.txt
c:/data/diets/../diets/mammals/Platypus.txt
c:/data/diets/../diets/birds/turkey.txt
```

We decide to chat with Security Sienna about this problem. She suggests we validate the input. We will use a *whitelist* that allows us to specify which values are allowed.

```
Console console = System.console();
String dirName = console.readLine();
if (dirName.equals("mammal") || dirName.equals("birds")) {
   Path path = Paths.get("c:/data/diets/" + dirName);
   try (Stream<Path> stream = Files.walk(path)) {
      stream.filter(p -> p.toString().endsWith(".txt"))
         .forEach(System.out::println);
   }
}
```

A *blacklist* is a list of things that aren't allowed. In the previous example, we could have put the dot (`.`) character on a blacklist. The problem with a blacklist is that you have to be cleverer than the bad guys. There are a lot of ways to cause harm. For example, you can encode characters.

By contrast, the whitelist is specifying what is allowed. You can supply a list of valid characters. Whitelisting is preferable to blacklisting for security because a whitelist doesn't need to foresee every possible problem.

That said, the whitelist solution could require more frequent updates. In the previous example, we would have to update the code any time we added a new animal type. Security decisions are often about trading convenience for lower risk.

Working with Confidential Information

Category	Examples
Login information	• Usernames • Passwords • Hashes of passwords
Banking	• Credit card numbers • Account balances • Credit score
PII (Personal identifiable information)	• Social Security number (or other government ID) • Mother's maiden name • Security questions/answers

Guarding Sensitive Data from Output

Security Sienna makes sure confidential information doesn't leak. The first step she takes is to avoid putting confidential information in a `toString()` method. That's just inviting the information to wind up logged somewhere you didn't intend.

She is careful what methods she calls in these sensitive contexts to ensure confidential information doesn't escape. Such sensitive contexts include the following:

- Writing to a log file
- Printing an exception or stack trace
- `System.out` and `System.err` messages
- Writing to data files

Protecting Data in Memory

When calling the `readPassword()` on `Console`, it returns a `char[]` instead of a `String`. This is safer for two reasons.

- It is not stored as a `String`, so Java won't place it in the `String` pool, where it could exist in memory long after the code that used it is run.
- You can `null` out the value of the array element rather than waiting for the garbage collector to do it.

When the sensitive data cannot be overwritten, it is good practice to set confidential data to `null` when you're done using it. If the data can be garbage collected, you don't have to worry about it being exposed later.

```
LocalDate dateOfBirth = getDateOfBirth();
// use date of birth
dateOfBirth = null;
```

The idea is to have confidential data in memory for as short a time as possible. This gives Hacker Harry less time to make his move.

Limiting File Access

It is good to apply multiple techniques to protect your application. This approach is called *defense in depth*. If Hacker Harry gets through one of your defenses, he still doesn't get the valuable information inside. Instead, he is met with another defense

You do need to be able to read one to understand security implications. Luckily, they are fairly self-explanatory. Here's an example of a policy:

```
grant {
   permission java.io.FilePermission
      "C:\\water\\fish.txt",
      "read";
};
```

This policy gives the programmer permission to read, but not update, the `fish.txt` file. If the program is allowed to read and write the file, we specify the following:

```
grant {
   permission java.io.FilePermission
      "C:\\water\\fish.txt",
      "read, write";
};
```

Serializing and Deserializing Objects

Specifying Which Fields to Serialize

Sienna reminds us that marking a field as `transient` prevents it from being serialized.

```
   private transient int age;
```

Alternatively, you can specify fields to be serialized in an array.

```
private static final ObjectStreamField[] serialPersistentFields =
   { new ObjectStreamField("name", String.class) };
```

You can think of `serialPersistentFields` as the opposite of `transient`. The former is a whitelist of fields that should be serialized, while the latter is a blacklist of fields that should not.

Take a look at the following implementation that uses `writeObject()` and `readObject()` for serialization, which you learned about in Chapter 19. For brevity, we'll use `ssn` to stand for Social Security number.

```
import java.io.*;

public class Employee implements Serializable {
   private String name;
   private String ssn;
   private int age;

   // Constructors/getters/setters

   private static final ObjectStreamField[] serialPersistentFields =
      { new ObjectStreamField("name", String.class),
      new ObjectStreamField("ssn", String.class) };

   private static String encrypt(String input) {
      // Implementation omitted
   }
   private static String decrypt(String input) {
      // Implementation omitted
   }

   private void writeObject(ObjectOutputStream s) throws Exception {
      ObjectOutputStream.PutField fields = s.putFields();
      fields.put("name", name);
      fields.put("ssn", encrypt(ssn));
```

```java
      s.writeFields();
   }
   private void readObject(ObjectInputStream s) throws Exception {
      ObjectInputStream.GetField fields = s.readFields();
      this.name = (String)fields.get("name", null);
      this.ssn = decrypt((String)fields.get("ssn", null));
   }
}
```

This version skips the `age` variable as before, although this time without using the `transient` modifier. It also uses custom read and write methods to securely encrypt/decrypt the Social Security number. Notice the `PutField` and `GetField` classes are used in order to write and read the fields easily.

Suppose we were to update our `writeObject()` method with the `age` variable.

```java
      fields.put("age", age);
```

When using serialization, the code would result in an exception.

```
java.lang.IllegalArgumentException: no such field age with type int
```

Pre/Post-Serialization Processing

Applying *readResolve()*

Now we want to start reading/writing the employee data to disk, but we have a problem. When someone reads the data from the disk, it deserializes it into a new object, not the one in memory pool. This could result in two users holding different versions of the `Employee` in memory!

Enter the `readResolve()` method. When this method is present, it is run *after* the `readObject()` method and is capable of replacing the reference of the object returned by deserialization.

```java
import java.io.*;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Employee implements Serializable {
   …
   public synchronized Object readResolve()
         throws ObjectStreamException {
      var existingEmployee = pool.get(name);
      if(pool.get(name) == null) {
         // New employee not in memory
         pool.put(name, this);
         return this;
      } else {
         // Existing user already in memory
         existingEmployee.name = this.name;
         existingEmployee.ssn = this.ssn;
         return existingEmployee;
      }
   }
}
```

If the object is not in memory, it is added to the pool and returned. Otherwise, the version in memory is updated, and its reference is returned.

Notice that we added the `synchronized` modifier to this method. Java allows any method modifiers (except `static`) for the `readResolve()` method including any access modifier. This rule applies to `writeReplace()`, which is up next.

Applying *writeReplace()*

The `writeReplace()` method is run *before* `writeObject()` and allows us to replace the object that gets serialized.

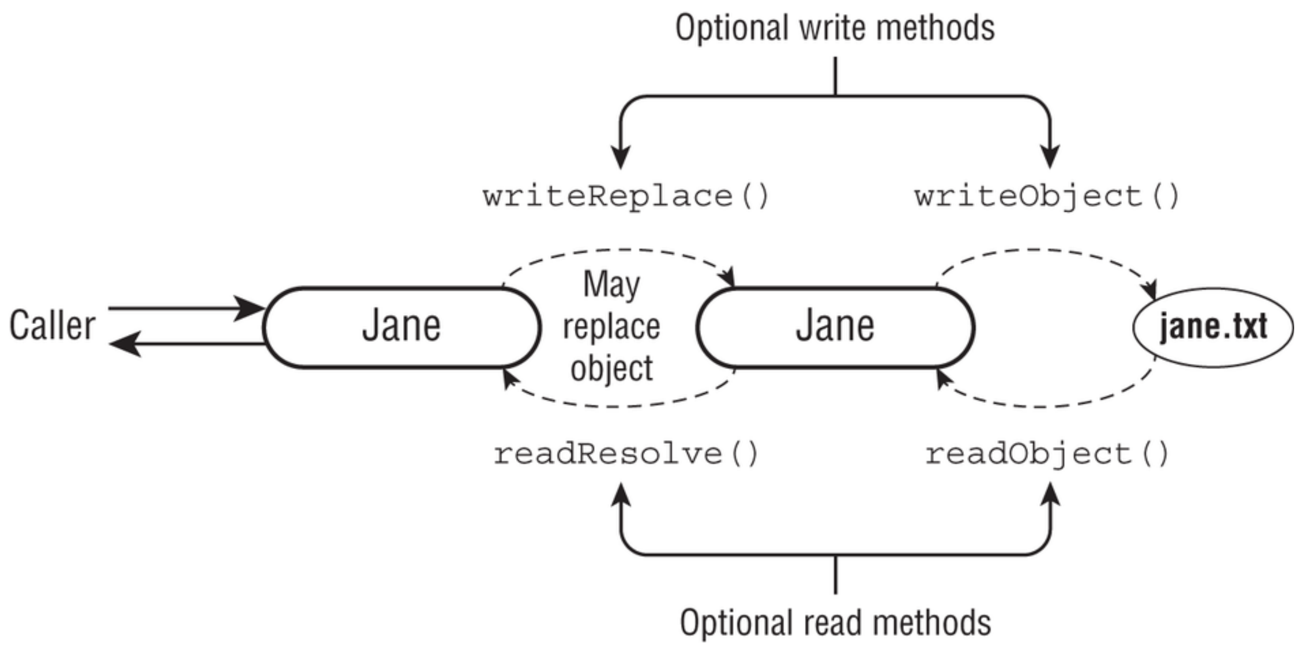
```
import java.io.*;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Employee implements Serializable {
   …
   public Object writeReplace() throws ObjectStreamException {
      var e = pool.get(name);
      return e != null ? e : this;
   }
}
```

This implementation checks whether the object is found in the pool. If it is found in the pool, that version is sent for serialization; otherwise, the current instance is used. We could also update this example to add it to the pool if it is somehow missing.

Reviewing Serialization Methods

TABLE 22.2 Methods for serialization and deserialization

Return type	Method	Parameters	Description
`Object`	`writeReplace()`	None	Allows replacement of object *before* serialization
`void`	`writeObject()`	`ObjectInputStream`	Serializes optionally using `PutField`
`void`	`readObject()`	`ObjectOutputStream`	Deserializes optionally using `GetField`
`Object`	`readResolve()`	None	Allows replacement of object *after* deserialization



Constructing Sensitive Objects

When constructing sensitive objects, you need to ensure that subclasses can't change the behavior. Suppose we have a `FoodOrder` class.

```
public class FoodOrder {
   private String item;
   private int count;
```

```java

   public FoodOrder(String item, int count) {
      setItem(item);
      setCount(count);
   }
   public String getItem() { return item; }
   public void setItem(String item) { this.item = item; }
   public int getCount() { return count; }
   public void setCount(int count) { this.count = count; }
}
```

This seems simple enough. It is a Java object with two instance variables and corresponding getters/setters. We can even write a method that counts how many items are in our order.

```java
public static int total(List<FoodOrder> orders) {
   return orders.stream()
      .mapToInt(FoodOrder::getCount)
      .sum();
}
```

This method signature pleases Hacker Harry because he can pass in his malicious subclass of `FoodOrder`. He overrides the `getCount()` and `setCount()` methods so that `count` is always zero.

```java
public class HarryFoodOrder extends FoodOrder {
   public HarryFoodOrder(String item, int count) {
      super(item, count);
   }
   public int getCount() { return 0; }
   public void setCount(int count) { super.setCount(0); }
}
```

Making Methods *final*

Security Sienna points out that we are letting Hacker Harry override sensitive methods. If we make the methods `final`, the subclass can't change the behavior on us.

Making Classes *final*

Remembering to make methods `final` is extra work. Security Sienna points out that we don't need to allow subclasses at all since everything we need is in `FoodOrder`.

Making the Constructor *private*

Security Sienna notes that another way of preventing or controlling subclasses is to make the constructor `private`. This technique requires `static` factory methods to obtain the object.

Preventing Denial of Service Attacks

A *denial of service* (DoS) attack is when a hacker makes one or more requests with the intent of disrupting legitimate requests. Most denial of service attacks require multiple requests to bring down their targets. Some attacks send a very large request that can even bring down the application in one shot. In this book, we will focus on denial of service attacks.

Unless otherwise specified, a denial of service attack comes from one machine. It may make many requests, but they have the same origin. By contrast, a *distributed denial of service* (DDoS) attack is a denial of service attack that comes from many sources at once.

Leaking Resources

One way that Hacker Harry can mount a denial of service attack is to take advantage of poorly written code. This simple method counts the number of lines in a file using NIO.2 methods we saw in Chapter 20:

```java
public long countLines(Path path) throws IOException  {
   return Files.lines(path).count();
}
```

Hacker Harry likes this method. He can call it in a loop. Since the method opens a file system resource and never closes it, there is a *resource leak*.

Luckily, the fix for a resource leak is simple, and it's one you've already seen in Chapter 20. Security Sienna fixes the code by using the try-with-resources statement we saw in Chapter 16,

Reading Very Large Resources

Another source of a denial of service attacks is very large resources. Suppose we have a simple method that reads a file into memory, does some transformations on it, and writes it to a new file.

```java
public void transform(Path in, Path out) throws IOException  {
   var list = Files.readAllLines(in);
   list.removeIf(s -> s.trim().isBlank());
   Files.write(out, list);
}
```

On a small file, this works just fine. However, on an extremely large file, your program could run out of memory and crash. Hacker Harry strikes again! To prevent this problem, you can check the size of the file before reading it.

Including Potentially Large Resources

An *inclusion* attack is when multiple files or components are embedded within a single file. Any file that you didn't create is suspect. Some types can appear smaller than they really are. For example, some types of images can have a “zip bomb” where the file is heavily compressed on disk. When you try to read it in, the file uses much more space than you thought.

Extensible Markup Language (XML) files can have the same problem. One attack is called the “billion laughs attack” where the file gets expanded exponentially.

The reason these files can become unexpectedly large is that they can include other entities. This means something that is 1 KB can become exponentially larger if it is included enough times.

Overflowing Numbers

When checking file size, be careful with an `int` type and loops. Since an `int` has a maximum size, exceeding that size results in integer overflow. Incrementing an `int` at the maximum value results in a negative number, so validation might not work as expected. In this example, we have a requirement to make sure that we can add a line to a file and have the size stay under a million.

```java
public static void main(String[] args) {
   System.out.println(enoughRoomToAddLine(100));
   System.out.println(enoughRoomToAddLine(2_000_000));
   System.out.println(enoughRoomToAddLine(Integer.MAX_VALUE));
}

public static boolean enoughRoomToAddLine(int requestedSize) {
   int maxLength = 1_000_000;
   String newLine = "END OF FILE";

   int newLineSize = newLine.length();
```

```
    return requestedSize + newLineSize < maxLength;
}
```

The output of this program is as follows:

```
true
false
true
```

Then we get to the final output of `true`. We start with a giant number that is over a million. Adding a small number to it exceeds the capacity of an `int`. Java overflows the number into a very negative number. Since all negative numbers are under a million, the validation doesn't do what we want it to.

Wasting Data Structures

One advantage of using a `HashMap` is that you can look up an element quickly by key. Even if the map is extremely large, a lookup is fast as long as there is a good distribution of hashed keys.

Hacker Harry likes assumptions. He creates a class where `hashCode()` always returns 42 and puts a million of them in your map. Not so fast anymore.

This one is harder to prevent. However, beware of untrusted classes. Code review can help detect the Hacker Harry in your office.

Open Web Application Security Project (OWASP) publishes a top 10 list of security issues. Some will sound familiar from this chapter, like injection. Others, like cross-site scripting (XSS), are specific to web applications. XSS involves malicious JavaScript.

If you are deploying to a cloud provider, like Oracle Cloud or AWS, there is even more to be aware of. The Cloud Security Alliance (CSA) also publishes a security list. Theirs is called the Egregious Eleven. This list covers additional worries such as account hijacking.