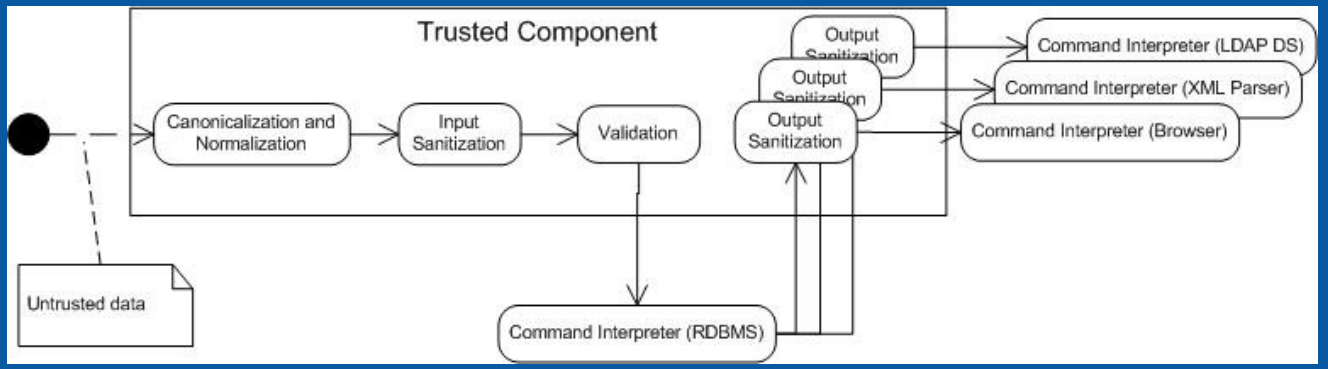
Secure Coding in Java

Highlights from SEI CERT Java Coding Standard

Sanitization & Validation of Data



The order of operations matter:

- Canonicalization and Normalization
- Sanitization
- Validation

Sanitization & Validation of Data

```
String s = "\uFE64" + "script" + "\uFE65";

// Normalize
s = Normalizer.normalize(s, Form.NFKC);

// Validate
Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(s);

if (matcher.find()) {
  throw new IllegalStateException();
} else {
  // ...
}
```

Sanitization & Validation of Data

```
String str="/home/fakih/../victor/pass.txt";
File file = new File(str);

String path1 = file.getAbsolutePath();
String path2 = file.getCanonicalPath();
```

Sanitization & Validation of Data

```
String str="/home/fakih/../victor/pass.txt";
File file = new File(str);

String path1 = file.getAbsolutePath();
String path2 = file.getCanonicalPath();
```

Calling `startsWith("/home/fakih")` will return false with the canonical path.

Beware of integral types

```
Set<Short> set = new HashSet<>();
short i = 11, j = 12;
set.add(i);
set.add(j);

set.remove(j - 1);

System.out.println(set.size());
```

Beware of integral types

```
Set<Short> set = new HashSet<>();
short i = 11, j = 12;
set.add(i);
set.add(j);

set.remove((short)(j - 1));

System.out.println(set.size());
```

Beware of integral types

```
FileInputStream in = …
byte data;
while((data = (byte) in.read()) != -1)
{
    //...
}
```

Beware of integral types

```
FileInputStream in = …
byte data;
while((data = (byte) in.read()) != -1)
{
    //...
}
```

If the next value is FF, data is set to -1.

Beware of integral types

```
FileInputStream in = …
byte data;
int buffer;
while((buffer = in.read()) != -1)
{
    data = (byte) buffer;
    // ...
}
```

The finalizer attack

```
class Vulnerable {
    Integer value = 0;

    Vulnerable(int value) {
     if(value <= 0) {
        throw new IllegalArgumentException
        ("Vulnerable value must be positive");
     }
     this.value = value;
    }
}


class AttackVulnerable extends Vulnerable {
 static Vulnerable vulnerable;

 public void finalize() {
    vulnerable = this;
 }
 // ...
}
```

Time of check, time of use (TOCTOU) race conditions

Preventing TOCTOU race windows is not possible because Java doesn't provide any mechanisms to prevent tampering with a file during its race window. But detecting some tampering is possible.

TOCTOU detect (check-use-check):

```
BasicFileAttributes attr = …
Object fileKey = attr.fileKey();
// use the file etc.
// get the file key again and compare with
// the first key.
```

Synchronization

Do not synchronize on objects that may be reused.

```
private final Boolean initialized = Boolean.FALSE;

synchronized(initialized){
    // …
}
```

Prefer using private final lock objects.

Synchronizing on non-thread-safe objects may cause problems:

```
private static volatile DateFormat format =
DateFormat.getDateInstance(DateFormat.MEDIUM);
```

AccessController

java.security.AccessController is the actual enforcer of Java's security model.

java.lang.SecurityManager is an ambassador. Most SecurityManager methods delegate their work to AccessController.

For a priviliged operation to proceed, every method on the call stack must be allowed to do it.

AccessController

AccessController.doPrivileged() executes a block of code with elevated privileges.
The 2-argument form of doPrivileged() accepts an AccessControlContext object from the caller and restricts the privileges of the contained code.

```
AccessControlContext context = …

final FileInputStream f[] = { null };

AccessController.doPrivileged(
new PrivilegedAction() {
    public Object run() {
    try {
         f[0] = new FileInputStream("file");
    } catch (FileNotFoundException ex) {
                // Forward to handler
    }
    return null;
   }
 }, context);
```

Serialization

Do not serialize direct handles to system resources, such as files. Do not serialize class fields which may convey sensitive data, mark them as transient.

Duplicate the SecurityManager checks enforced in a class during serialization and deserialization.

```
public final class SensitiveClass implements
java.io.Serializable
{
  public SensitiveClass() {
    securityManagerCheck();
  }

  private void readObject(ObjectInputStream in)
  {
    securityManagerCheck();
  }
}
```

Cloning objects

Cloning is another way of creating objects without executing a constructor. This bypasses any security checks in the constructor.

Make sensitive classes non-cloneable.

Provide a clone() method that throws `CloneNotSupportedException`.

Other Highlights

- The only unsigned integral type in Java is char.
It's intended for holding 16-bit characters not for arithmetic.
Performing arithmetic on char values is strongly discouraged.

- Keep the data ranges of integral types in mind.
Detect or prevent overflow appropriately.

- Two classes are the same class if they are loaded by the same class loader and they have the same fully qualified class name.

```
if(obj.getClass() ==
this.getClassLoader.loadClass("fully qualified class
name")
{
    //…
}
```

Other Highlights

- Do not expose sensitive information (such as file paths, personal data, configuration details) to application logs and exceptions.

- Also check the library logs that do not have the semantic knowledge of the data they are dealing with. For example: String parser libraries.

References:

https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

http://www.oracle.com/technetwork/java/seccodeguide-139067.html

https://www.ibm.com/developerworks/library/j-fv/j-fv-pdf.pdf