Fault-Tolerant Computer System Design
ECE 60872

Secure Coding Practices

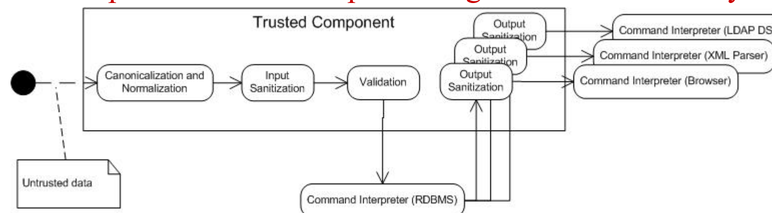
Saurabh Bagchi
ECECS
Purdue University

Goals

- The goal of software security is to maintain the *confidentiality, integrity*, and *availability* of information
- The goal is accomplished through the implementation of *security controls*
- Here we discuss technical controls specific to *mitigating* the occurrence of common software *vulnerabilities*.

Input Validation and Data Sanitization

- Software programs often contain multiple components wherein each component operates in one or more trusted domains.
  - Example: Component 1 has access to the file system but no access to the network, while component 2 has access to the network but not to file system.
- Distrustful decomposition and privilege separation are examples of secure design patterns that reduce the amount of code that runs with special privileges
- Data that crosses a trust boundary should be validated unless the code that produces this data provides guarantees of validity.



How to Validate Data that Crosses Trust Boundary

- Validation: Process of ensuring that input data falls within the expected domain of valid program input.
  - Requires that inputs conform to type and numeric range requirements plus to input invariants
- Sanitization: In many cases, the data is passed directly to a component in a different trusted domain. Data sanitization is the process of ensuring that data conforms to the requirements of the subsystem to which it is passed.
  - Sanitization also involves ensuring that data conforms to security-related requirements regarding leaking or exposure of sensitive data when output across a trust boundary.
  - Sanitization may include the elimination of unwanted characters from the input by means of removing, replacing, encoding, or escaping the characters.
  - Sanitization may occur following input (input sanitization) or before the data is passed across a trust boundary (output sanitization).
- Data sanitization and input validation may coexist and complement each other.
- String data that specify commands or instructions are a special concern because they may contain special characters that can trigger commands or actions, resulting in a software vulnerability

Example: Need for Input Validation

- A user who has the ability to provide input string data that is incorporated into an XML document can inject XML tags.
- These tags are interpreted by the XML parser and may cause data to be overridden.
- An online store application that allows the user to specify the quantity of an item has the following XML document:

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity>
</item>
```

- An attacker might input the following string instead of a count for the quantity:

```
1</quantity><price>1.0</price><quantity>1
```


Example: Need for Input Validation

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity>
</item>
```

```
1</quantity><price>1.0</price><quantity>1
```

- In this case, the XML resolves to the following:

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity><price>1.0</price><quantity>1</quantity>
</item>
```

- An XML parser may interpret the XML in this example such that the second price field overrides the first, changing the price of the item to $1.

Leaking Capabilities

- A capability is a communicable, unforgeable token of authority. It refers to a value that references an object along with an associated set of access rights.
- A user program on a capability-based operating system must use a capability to access an object.
- Every Java object has an unforgeable identity in addition to its contents, because the == operator tests reference equality.
  - This unforgeable identity allows use of a reference to an object as a token, serving as an unforgeable proof of authorization to perform some action
- One surprising source of leaked capabilities and leaked data is inner classes, which have access to all the fields of their enclosing class.

Example: Leaked Capabilities

- Root cause: Nested class has access to the private fields of the outer class. Also, the same fields can be accessed by any other class within the package when the nested class is declared public or if it contains public methods or constructors.
- Principle: The nested class must not expose the private members of the outer class to external classes or packages.

```
class Coordinates {
  private int x;
  private int y;

  public class Point {
    public void getPoint() {
      System.out.println("(" + x + "," + y + ")");
    }
  }
}

class AnotherClass {
  public static void main(String[] args) {
    Coordinates c = new Coordinates();
    Coordinates.Point p = c.new Point();
    p.getPoint();
  }
}
```

Example: Leaked Capabilities

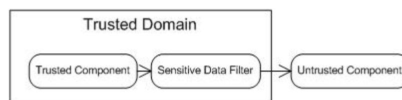
- Compliant Solution: Uses the private access specifier to hide the inner class and all contained methods and constructors.

```java
class Coordinates {
  private int x;
  private int y;

  private class Point {
    private void getPoint() {
      System.out.println("(" + x + "," + y + ")");
    }
  }
}

class AnotherClass {
  public static void main(String[] args) {
    Coordinates c = new Coordinates();
    Coordinates.Point p = c.new Point();    // Fails to compile
    p.getPoint();
  }
}
```

- Compilation of AnotherClass now results in a compilation error because the class attempts to access a private nested class.

Leaking Sensitive Data

- A system's security policy determines which information is *sensitive*.
  - Sensitive data may include user information such as social security or credit card numbers, passwords, or private keys.
- When components with differing degrees of trust share data, the data are said to flow across a trust boundary.
- Preventing unauthorized access may be as simple as not transmitting the data, or it may involve filtering sensitive data from data that can flow across a trust boundary.



Example: Leaking Information through Exception Messages

- Failure to filter sensitive information when propagating exceptions can result in leaks that help an attacker develop further exploits.
- An attacker may craft input arguments to expose internal structures and mechanisms of the application.
- Both the exception message text and the type of an exception can leak information.
  - For example, the FileNotFoundException message reveals information about the file system layout, and the exception type reveals the absence of the requested file.

Example: Leaking Information through Exception Messages

- The program must read a file supplied by the user, but the contents and layout of the file system are sensitive.
- The program accepts a file name as an input argument.

```
class ExceptionExample {
  public static void main(String[] args) throws FileNotFoundException {
    // Linux stores a user's home directory path in
    // the environment variable $HOME, Windows in %APPDATA%
    FileInputStream fis =
        new FileInputStream(System.getenv("APPDATA") + args[0]);
  }
}
```

- When a requested file is absent, the FileInputStream constructor throws a FileNotFoundException, allowing an attacker to reconstruct the underlying file system by repeatedly passing fictitious path names to the program.

Example: Leaking Information through Exception Messages

- The code example logs the exception and then wraps it in a more general exception before rethrowing it

```
try {
  FileInputStream fis =
      new FileInputStream(System.getenv("APPDATA") + args[0]);
} catch (FileNotFoundException e) {
  // Log the exception
  throw new IOException("Unable to retrieve file", e);
}
```

- Even when the logged exception is not accessible to the user, the original exception is still informative and can be used by an attacker to discover sensitive information about the file system.
- This example also violates principle of release resources when they are no longer needed, as it fails to close the input stream in a finally block.

Race Conditions

- A race condition occurs when multiple threads of control try to perform a non-atomic operation on a shared object
  - Multithreaded applications accessing shared data
  - Accessing external shared resources such as the file system
- General causes
  - Threads or signal handlers without proper synchronization
  - Non-reentrant functions (may have shared variables)
  - Performing non-atomic sequences of operations on shared resources (file system, shared memory) and assuming they are atomic
- A program contains a data race if two threads simultaneously access the same variable, where at least one of these accesses is a write

Slides courtesy of Barton P. Miller (U of Wisconsin)

Example: Race Condition

```
void TransFunds(Account srcAcct, Account dstAcct, int xfrAmt)
{
  if (xfrAmt < 0)
    FatalError();
  int srcAmt = srcAcct.getBal();
  if (srcAmt - xfrAmt < 0)
    FatalError();
  srcAcct.setBal(srcAmt - xfrAmt);
  dstAcct.setBal(dstAcct.getBal() + xfrAmt);
}
```

JAVA

Thread 1	time	Thread 2	Balances: Bob	Balances: Ian
XfrFunds(Bob, Ian, 100)		XfrFunds(Bob, Ian, 100)	100	0
srcAmt = 100				
		srcAmt = 100		
srcAmt - 100 < 0 ?				
		srcAmt - 100 < 0 ?		
srcAcct.setBal(100 - 100)			0	
		srcAcct.setBal(100 - 100)	0	
dst.setBal(0 + 100)				100
		dst.setBal(0 + 100)		200

Slides courtesy of Barton P. Miller (U of Wisconsin)

Example: Mitigation of Race Condition

```
public void TransFunds(Account srcAcct, Account dstAcct, int xfrAmt)
{
      if (xfrAmt < 0)  FatalError();
      synchronized(srcAcct) {
         int srcAmt = srcAcct.getBal();
         if (srcAmt - xfrAmt < 0)
            FatalError();
         srcAcct.setBal(srcAmt - xfrAmt);
      }
      synchronized(dstAcct) {
         dstAcct.setBal(dstAcct.getBal() + xfrAmt);
      }
}
```

JAVA

Thread 1	time	Thread 2	Bob	Ian
XfrFunds(Bob, Ian, 100)		XfrFunds(Bob, Ian, 100)	100	0
In use srcAcct? No, proceed.				
		In use srcAcct? Yes, wait.		
srcAmt = 100				
srcAmt - 100 < 0 ?				
srcAcct.setBal(100 - 100)			0	
In use dstAcct? No, proceed.		srcAmt = 0		
dst.setBal(0 + 100)		srcAmt - 100 < 0? Yes, fail		100

Slides courtesy of Barton P. Miller (U of Wisconsin)

Summing it up: The Top 10

1. Validate input.
2. Heed compiler warnings.
3. Architect and design for security policies.
4. Keep it simple.
5. Default deny.
6. Adhere to the principle of least privilege.
7. Sanitize data sent to other systems.
8. Practice defense in depth.
9. Use effective quality assurance techniques.
10. Adopt a secure coding standard.

References

- OWASP "Secure Coding Practices - Quick Reference Guide," November 2010.
- SEI "CERT Top 10 Secure Coding Practices," last modified May 2011.
- SEI "CERT Oracle Coding Standard for Java," last modified: May 2015.