Java Software Security

COMP620

Java Security

- The Java platform was designed with a strong emphasis on security
- There are language features to improve security, such as strong typing and range checking on arrays
- There are many system features to support security of Java programs running as applets, applications and server programs

Bytecode Verifier

Upon loading a class, the JVM checks

- The class file is in the correct format
- Final classes and methods are not overridden
- No prohibited casting
- No stack overflows or underflows

Basic Security Architecture

- cryptography
- public key infrastructure
- authentication
- secure communication
- access control.

Platform Security Implementation

- Implementation independence
  - Applications can request security services from the Java platform
  - Security services are implemented in providers
- Implementation interoperability
- Algorithm extensibility
  - Platform includes a number of built-in providers
  - Additional services can be added

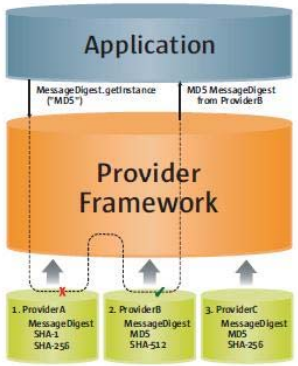
Providers

- The Java system provides a list of vendor providers for security services
- The standard JVM includes a number of default providers
- The **`java.security.Provider`** class encapsulates the list of providers
- The **`getInstance`** method can be used by a program to get a provider

Provider Search

Consider a request for any provider that has message digest MD5

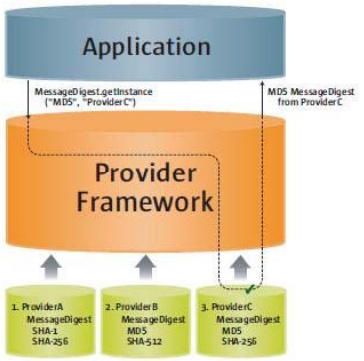
```
MessageDigest md = MessageDigest.getInstance("MD5");
```



Specified Provider

Consider a request for a specific provider that has message digest MD5

```
MessageDigest md =
      MessageDigest.getInstance("MD5", "ProviderC");
```



Secure Communications

- Java provides APIs for several different encrypted communication protocols including
  - Secure Socket Layer (SSL)
  - Transport Layer Security (TLS)
- The Java platform also defines basic Kerberos classes

Access Control

- The Java platform protects access to sensitive resources (for example, local files) or sensitive application code (for example, methods in a class)
- All access control decisions are mediated by a security manager
- A SecurityManager must be installed into the Java runtime in order to activate the access control checks

Security Manager

- By default, there is no security installed when you execute a Java application from the command line
- You can load a security manager programmatically or by using the **`-Djava.security.manager`** argument on the command line

Java Permissions

- Permissions are represented by **`java.security.Permission`** objects
- Permissions are based on
  - Where the code was loaded from
  - Who signed the code (if anyone)
  - Default permissions granted to the code

```
SecurityManager sm =
  System.getSecurityManager();
if (sm != null) {
  sm.checkPermission(perm); }
```

Java Policies

- The basic responsibility of a Policy object is to determine whether access to a protected resource is permitted
- Java encapsulates a security policy in the java.security.Policy class
- Policies are kept in a .policy file, either the default or a user created .policy file
- Policy files can be created using the GUI **policytool** utility

Access Control Example

- Consider a simple Java application that reads files from **`/mydir`** and from **`/otherdir`**
- When run with no security manager, the program can read both files
- When run with the security manager and default policies, neither file can be read
- Access to a specific directory can be allowed by creating a policy file and specifying it on the java command line

Java Command Line

- java -Djava.security.manager
  -Djava.security.policy=MyPolicy.policy
  -jar limited.jar

Java Security Files

- The default .policy and .security files are located in the Java Runtime Environment (JRE) directory in the subdirectory /lib/security

Types of Concerns

- General good programming practice
- Protection against attacks
- Protection when methods are used in a possibly threatening application

Some Useful Coding Rules that Have Little to Do with Security

- Inexperienced programmers may make these mistakes and generate programs that do not work correctly
- Most of these flaws create programs that do not work, but are not necessarily vulnerable

String Comparison

- Do not use the operators ==, !=, <, >, >=, <= when comparing strings

```
String one = new String("comp620");
String two = new String("comp620");
if (one == two)          // not true
if (one.equals(two))     // true
```

Final Does Not Lock Objects

- The Term FINAL when used in Java is generally thought to create a data type that is meant to not change, when this applies to an object, you may not change the reference to the object; however CAN change the contents in the object.

Modified final Object

//define an array and declare it as final and we do not want anyone to change it!

```
final int[] a = {1,2,3};
for(i=0;i<3;i++) System.out.println(a[i]);
```

//change the value but the reference is still unchanged!

```
a[0]= 10; a[1]= 20;
for(i=0;i<3;i++) System.out.println(a[i]);
```

Output:

1
2
3
10
20
3

Scope minimization

- Scope minimization helps in capturing common programming errors
- Improves code readability by tying together the declaration and actual use of a variable
- Eases maintainability because unused variables are easily caught and removed.

Solution

- To minimize the scope of variables declare loop indexes within the for statement.
- Methods should be designed to perform only one operation if possible. This reduces the need for variables existing in overlapping scopes and consequently, helps prevent errors.

Keeping the Code Safe in a Hostile Program

- Several of the Java coding issues involved writing methods that would maintain a secure system when incorporated into a malicious program
- It is possible to write a program that calls methods from a jar file downloaded as part of a web application

Minimizing Accessibility of Classes

- Creating public fields and methods gives any one access to the fields and methods
- Although a class maybe private, if the members are public they are still unsafe

Vulnerable Code

The class and its members are Public – very easy access to information of the class

Class Declared

Members of Class

```
public class PublicClass {
        public int x;
        public int y;

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

How To Fix It!

- Make fields private. Provide get and set methods to access the data.
- Declaring a class as 'Final' eliminates the class from being extended or sub-classed

More Secure Code

The class is finalized and its members are private - minimizing malicious attacks on information within the class

Declaring This Class Final

Members of Class

```
final class PrivateClass {
        private int x;
        private int y;

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

Time of Check Time of Use (TOCTOU)

- Be careful that code that validates an object always uses the same validated object. Don't allow an object to be changed after validation.

Extended Objects

- If your method has a parameter object of type Widget, it can be passed an object of any class that extends Widget
- A malicious programmer can extend a system class and pass it as a parameter
- The extended class might override methods to provide different results

Vulnerable Code

- The following code calls getPath twice, once to check authorization and again to create a file.

```
public RandomAccessFile java.io.File inFile){
if (!Permitted(inFile.getPath()) throw exception;
  // ...
  return new RandomAccessFile(inFile.getPath());
}
```

- This will work for a normal File object

Extended Object

- An attacker can extend java.io.File as :

```
public class BadFile extends java.io.File {
  private int count = 0;
  public String getPath() {
  if (count++ == 0)
     return super.getPath();   // for check
  else
     return "secrets.txt";     // for new file
  }
}
```

Secure Code

- The following code calls getPath once to create a new File object with the requested name

```
public RandomAccessFile java.io.File inFile){
final java.io.File copy =
          new java.io.File(inFile.getPath());
if (!Permitted(copy.getPath()) throw exception;
  // ...
  return new RandomAccessFile(copy.getPath());
}
```

Important Security Coding Issues

- Always Validate Input
- Sanitize Data Sent to Other Systems
- Heed Compiler Warnings
- Keep It Simple
- Default Deny
- Principle of Least Privilege
- Practice Defense in Depth
- Use Quality Assurance Techniques

Fragile Class Hierarchy

- Occurs when changes in the superclass indirectly cause changes to the subclass's behavior
- Example:
  - A subclass CheckDinnerTable inherits eat() and drink() methods from superclass DinnerTable to ensure that security checks are made.
  - The converse() method is then added to the superclass DinnerTable without the subclass knowing
  - Now there will be no security checks on the converse method

Original Code

```
class DinnerTable{
        void eat(typeFood){
            //eat something
        }
        void drink(typeDrink){
            //drink something
        }
}
```

```
public class Client {
    public static void main(String[] args) {
    DinnerTable sc = new CheckDinnerTable();
    sc.eat(chicken);

    }
}
```

```
class CheckDinnerTable extends DinnerTable{
        @override void eat(typeFood){
            //check typeFood, secCheck
            super.eat();
        }
        @override void drink(typeDrink){
            //check typeDrink, secCheck
            super.drink();
        }
}
```

BAD Code Example

```
class DinnerTable{
        void eat(typeFood){
            //eat something
        }
        void drink(typeDrink){
            //drink something
        }
        void converse(emptyMouth){
            //talk amongst peers
        }
}
class CheckDinnerTable extends DinnerTable{
        @override void eat(typeFood){
            //check typeFood, secCheck
            super.eat();
        }
        @override void drink(typeDrink){
            //check typeDrink, secCheck
            super.drink();
        }
}
```

```
public class Client {
    public static void main(String[] args) {
    DinnerTable sc = new CheckDinnerTable();
    sc.eat(chicken);
    sc.converse(False);
    }
}
```

Problem:
Converse method is not checked for input validation or security violations. Therefore the client can converse at the dinner table with a full mouth of food!!!

Improved Code Example

```
class DinnerTable{
        void eat(typeFood){
            //eat something
        }
        void drink(typeDrink){
            //drink something
        }
        void converse(emptyMouth){
            //talk amongst peers
        }
}
class CheckDinnerTable {
    DinnerTable table = new DinnerTable();
        void eat(typeFood){
            //check typeFood, secCheck
            table.eat();
        }
        void drink(typeDrink){
            //check typeDrink, secCheck
            table.drink();
        }
}
```

```
public class Client {
    public static void main(String[] args) {
    DinnerTable sc = new CheckDinnerTable();
    sc.eat(chicken);
    sc.converse(False);   // generates an error
    }
}
```

The CheckDinnerTable class does not extend DinnerTable, but creates one as a field. Methods of DinnerTable cannot be called.

Avoiding Attacks

- These Java coding issues deal directly with security.

Directory Traversal Attacks

- Programs can ask for user input to specify a desired file within a given directory
- This user input may allow an I/O operation to escape the specified operating directory.
- With this type of I/O operation, users could traverse through other directories other than the one specified, accessing undisclosed information
- Major threat to confidentiality

Example

```
FileOutputStream fis =
  new FileOutputStream("/img/"+input);
```

Input: `../etc/passwd`

- Code allows user to specify the absolute path of a file on which operations are to be performed.
- If user enters an argument that contains "`../`", it is possible to escape out of the `/img` directory and operate on a file present within the root directory.
- the path `/img/../etc/passwd` resolves to `/etc/passwd`. This is insecure because the program breaks out of the specified directory `/img`.

Proper Solution 1

```
File f = new File("/img/" + input);
String canonicalPath = f.getCanonicalPath();
if(!canonicalPath.startswith("/img/")) { // Validation
  throw new IllegalArgumentException();
}
```

- Obtain the canonicalized file name from the untrusted user input and validate it against the target path, before operating on the file.

Proper Solution 2

```
String Path = "/img/" + input;
if(Path.contains("../")) { // Search for operator
  throw new IllegalArgumentException();
 }
```

- Search user input for "../" before using the path to create file

Avoid Creating Temporary Files in Shared Directories

- Frequently created in directories that are writable by all users (i.e. /tmp and C:\TEMP)
- Misused by adversaries that have access to the local file system
  - Name of the file can be predetermined, using the time-of-check time-of-use (TOCTOU) condition
  - Cause a program to incorrectly interpret the temporary data if temporary files are not created safely
  - Create a denial of service if the file cannot be created

Correction Strategies

Temporary files in shared directories should be:

- Created with unique and unpredictable file names
- Opened with exclusive access
- Removed before the program exits
- Opened with appropriate permissions

Never hardcode sensitive information

- Consider the simple class with a private field

```
public class MySecrets {
    private String password
            = "TopSecret";
    // …
}
```

javap Will Display Class Fields

```
javap -private -c MySecrets
```

```
public class MySecrets extends java.lang.Object{
private java.lang.String password;
Code:
  0:  aload_0
  1:  invokespecial  #1; //Method java/lang/Object."<init>"
  4:  aload_0
  5:  ldc    #2;     //String TopSecret
  7:  putfield       #3; //Field password:Ljava/lang/String;
  10: return
```

Numeric Overflow

- An int can hold a value from –2147483648 to 2147483647
- If arithmetic results in number that would be outside this range, the number will wrap around.
- 2147483647 + 1 = –2147483648
- If you are checking an input number

```
if (input+x < 5)// true if input is very large
```

Validate Input Numbers

- You should validate integer input

```
if( input > Integer.MAX_VALUE - x ||
  input < Integer.MIN_VALUE - x )
{ throw new ArithmeticException();
```

- The simple check

```
if (input+x > Integer.MAX_VALUE)
```

- will not work.