Lecture 4: Java Security model, part I

Lecture topics

- Introduction to the Java security model

Java architecture, high level
Lecture 4: Java Security model, part I

Security in Java is achieved through

- Host- and application-specific security configurations
- Lack of pointer arithmetic
- Garbage collection
- Strict compile-time checking
- Load-time checking (bytecode verification)
  - Integrity of class file format
  - Integrity of the constant pool
  - Access restrictions obeyed
- Run-time checking (array bounds, type casts, …)
- Explicit security model

Evolution: Java 1.0 security model

- Applets are not allowed to read or write local files
- Applets are allowed to open TCP sockets only to the server they are loaded from
Lecture 4: Java Security model, part I

Evolution: Java 1.1 security model

- Applets that carry a trusted digital signature are allowed full access to system resources

Evolution: Java 2 security model

- Classes are assigned protection domains that govern their access to resources
Lecture 4: Java Security model, part I

**The protection domain concept**

- A domain includes a set of classes whose objects are granted the same set of permissions with respect to system resources.
- A security policy defines protection domains for a JVM instance.
  - The JVM maintains mapping from classes to their protection domains and permissions:

![Diagram showing classes and domains with permissions]

**Java security model: permissions**

- Remotely loaded classes may be given permission to perform specific operations:
  ```java
grant signedBy "Poly", codebase
  http://cis.poly.edu/gleb/classes/- {
    permission java.io.FilePermission "/tmp/**", "read, write";
  }
  ```
- Code dealing with sensitive operations (e.g., reading/writing, opening sockets) has to check that callers have sufficient permissions:
  ```java
  AccessController.checkPermission(new java.io.FilePermission("/tmp/**", "read, write"));
  ```
- Permissions are always positive:
  - I.e., specify what can be done, never what cannot be done.
Lecture 4: Java Security model, part I

Security policy format

- `grant [SignedBy "signer_names"] [ CodeBase "URL"]
  [, Principal [principal_class_name] "principal_name"]
  [, Principal [principal_class_name] "principal_name"] ...
  [
    permission permission_class_name [ "target_name" ]
    [, "action" [, SignedBy "signer_names";]
    permission ...
  ];
- A separate keystore entry provides a mapping between public keys and associated digital certificates
- Principals are owners of Java threads performing computations
- Example:
  ```
  keystore "http://foo.bar.com/blah/.keystore";
  grant principal "alice" {
    permission java.io.FilePermission "/tmp/games",
    "read, write";
  };
  ```
  ° "alice" will be replaced by
  `javax.security.auth.x500.X500Principal "cn=Alice",
  taken from keystore`

Sample permissions

- **File access**
  - `java.io.FilePermission "/tmp/*", "read,write"
  - `java.io.FilePermission "/{user.home}${/}*",
    "read"

- **System permissions**
  - `java.io.RuntimePermission "getClassLoader", ""

- **GUI permissions**
  - `java.io.AWTPermission "accessEventQueue", ""

- **Network access**
  - `java.io.SocketPermission "*:1024-", "connect"
  - `java.io.SocketPermission "*:8080", "accept,
    listen"
Lecture 4: Java Security model, part I

Permission class hierarchy

- AllPermission allows access to all resources
- BasicPermission is a base class for named permissions
  - Have a target name, but no actions
- Important: permissions are immutable — their state cannot be modified once initialized

Important methods of Permission

- Permission(String name)
- String getName()
- abstract boolean implies(Permission other)
  - Determines if this permission implies the given permission
  - If permission $P_1$ implies permission $P_2$, then any class that is granted $P_1$ is automatically granted $P_2$
Lecture 4: Java Security model, part I

**Aggregating permissions**

- **PermissionCollection** can be used to specify aggregate permissions
  - A **homogeneous** collection of permissions
    - All permissions it holds must be of the same type
  - A class that is granted a collection of permissions has all permissions in this collection
  - Important methods:
    - `void add(Permission)`
    - `boolean implies(Permission)`
    - `void setReadOnly()`
    - `boolean isReadOnly()`

- **Permissions** holds a **heterogenous** collection of permissions
  - Technically, a collection of PermissionCollection objects

- Alternatively, aggregate permissions can be defined via the `implies` method

---

**Permission checking example**

```
public class Account {
    private Money balance;
    private String persistentLocation;

    public Account(Money initialAmount, String persistentLocation) {
        AccessController.checkPermission( 
            new NewAccountPermission("NewAccountPermission"));
        this.balance = (Money) initialAmount.clone();
        this.persistentLocation = persistentLocation;
    }

    public Money getBalance() {
        AccessController.checkPermission( 
            new BalancePermission("BalancePermission"));
        return (Money) this.balance.clone();
    }

    public void credit(Money amount) {
        AccessController.checkPermission( 
            new CreditPermission("CreditPermission"));
        this.balance.add(amount);
        this.write();
    }

    public void debit(Money amount) {
        AccessController.checkPermission( 
            new DebitPermission("DebitPermission"));
        this.balance.subtract(amount);
        this.write();
    }

    private void write() {
        AccessController.doPrivileged( 
            new PrivilegedAction() {
                public Object run() {
                    FileWriter writer = 
                        new FileWriter(this.persistentLocation);
                    writer.write(balance);
                    writer.close();
                    return null;
                }
            });
    }

    public void transfer(Money amount, Account toAccount) {
        this.debit(amount);
        toAccount.credit(amount);
    }
}
```
Permission checking: running example

```java
public class AccountWithProtection extends Account {
    private Account protection;

    public AccountWithProtection(Money initialAmount,
                                   String persistentLocation,
                                   Account protection) {
        super(initialAmount, persistentLocation);
        this.protection = protection;
    }

    public void debit(Money amount) {
        AccessController.checkPermission(
            new CustomerPermission("CustomerPermission"));
        AccessController.doPrivileged() {
            public Object run() {
                Money currentBalance = this.getBalance();
                if (currentBalance.compareTo(amount) == Money.LESS_THAN) {
                    Money toTransfer = amount.clone();
                    toTransfer.subtract(currentBalance);
                    this.protection.transfer(toTransfer, this);
                }
                super.debit(amount);
            }
        };
    }
}
```

java.security.AccessController class

- Decides whether access to a critical system resource should be granted or denied
- Marks code as privileged if necessary
  - Exempt from access checking
- Obtains snapshots of the current calling context, so that access-control decisions with respect to this context can be made from a different context
  - Multi-threading issue
Lecture 4: Java Security model, part I

**How checkPermission works**

- Examines all classes on the call stack and makes sure that all of them have permission to do an operation

```java
class MainClass {
    public static void main(String[] args) {
        MyClass o = new MyClass();
        o.m();
    }
}
class MyClass {
    ...
    public void m() {
        ...
        RemoteClass obj = new RemoteClass();
        obj.passAccount(new Account(1000));
    }
}
```

JVM makes sure all classes have the permission checked in Account

---

**Java security model: privileged regions**

- "Trusted" code can temporarily grant additional privileges to other classes in the call stack
- Useful in situations where untrusted classes can be allowed to perform specific actions that would otherwise require privileges
- E.g., an application programmer may decide to
  - Not give untrusted classes permissions to check balance, credit, or debit an account
  - Let untrusted classes find out whether the account balance is above or below $1000
Lecture 4: Java Security model, part I

**How privileged regions work**

```java
class MainClass {
    public static void main(String[] args) {
        MyClass o = new MyClass();
        o.m();
    }
}

class MyClass {
    private Account account;
    ...
    public void debit(int amount) {
        AccessController.doPrivileged(PrivilegedAction() {
            this.account.debit(amount);
        });
    }
    public void m() {
        ...
        RemoteClass obj = new RemoteClass();
        obj.giveAccess(MyClass.this);
    }
}
```

JVM checks that these classes have DebitPermission

JVM does not check that these classes have DebitPermission

**Inheritance of access control context**

- When a thread creates a new thread, a new stack is created
- When a new thread is created, a hand-over of the control context has to happen
- This hand-over is transitive
  - If thread A creates thread B that creates thread C, then C will have contexts of both A and B
How user-defined permissions are created

```java
public class CreditPermission extends BasicPermission {
  ...
  public boolean implies(Permission p) {
    return (p instanceof CreditPermission) ||
      (p instanceof FilePermission);
  }
}
```

- The `implies` method can contain arbitrary code
  - But in general, it is often a good idea to compare only types, targets, and operations of permissions

Avoid cycles in implication relationships across permission classes

- Permissions A, B, and C are equivalent
- The compiler/run-time system are not going to warn you about this
**Strength of permissions is measured not only by what other permissions they imply**

- Some types of permissions grant the ability to effectively acquire additional permissions
  - E.g. `java.io.FilePermission "<<ALL FILES>>, "write"
    - Allows loaded classes to replace the JVM; effectively, equivalent to `AllPermission`
  - E.g. `java.io.RuntimePermission "createClassLoader", ""
    - Loaded classes can create arbitrary class loaders, which may not force bytecode verification...
  - Similarly, the ability to load native code libraries, set system properties, and define packages is dangerous

**Unresolved permissions**

- Permission classes can be loaded into an application dynamically
  - Some permission classes may not be loaded by the time the policy file is loaded in the JVM
- **UnresolvedPermission** class is used to represent permissions that have not been loaded yet
- When `checkPermission` is executed with an unresolved permission, an attempt to resolve it is made
  - If the class for the permission is available, the unresolved permission is replaced with a permission of the loaded class
  - If the class for the permission is still not available, the permission is considered invalid
Is knowing all this enough to use the Java security model effectively and correctly?

- Need to know what methods in the standard libraries require what exceptions
  - So that security policy can be configured correctly
  - A technique for figuring this out statically at IBM T.J. Watson
    Immature at the moment
- Need to validate whether sensitive regions of code are protected by permissions
  - A technique for validation using static analysis here
    Immature and not fully implemented at the moment
- Lots of practice!