Security Policy: A Design Pattern for Mobile Java Code

Qusay H. Mahmoud
School of Computer Science, Carleton University
1125 Colonel By Drive, Ottawa, Ontario, Canada K1S 5B6
qmahmoud@scs.carleton.ca

Abstract

When users on the net visit a homepage that has an embedded applet, the mobile code is downloaded to the user's machine and executed there. In other words, the applet's code migrates from the host's machine to the user's machine, and it will run on the user's machine. In such an environment, we want to make sure that the code being downloaded does not do any harm to the system on which it will be executed. Also, when network computers (devices with not much local storage) get deployed on the net, they would have to use the network as a source for all sorts of full-fledged applications. In such an environment, it is difficult to predict what a downloaded application will need to do. In such distributed environments, security is a major concern. This paper presents the Security Policy pattern, a design pattern that has been used in many contexts, and proved to be useful, to develop applications capable of securely loading classes off the network and executing them locally. The Security Policy pattern can be used either on the client- or server-side. For example, in the case of a Web browser, the pattern is used on the client-side, and in the case of a global compute engine the pattern is used on the server-side. While the pattern may sound Java-centric, it can however be implemented in other languages.

Problem

How do you protect the user's machine file system and network resources from, possibly malicious, code loaded off the network.

Context

Millions of users are visiting applet-enabled homepages or downloading unseen programs, from unknown web sites, allowing them unquestioned access to their systems. The power of mobile code is undeniable, but so are the security issues associated with this technology. For example: a user allowed untrusted agents to run on his machine, not knowing that they can steal proprietary information (e.g. credit card numbers).

Use the security policy pattern under any of the following circumstances:

- You are an Internet programmer and want to write the next generation Web browser that is capable of downloading mobile code and executing it on the local machine. You would want to offer the users a comfort level by assuring them that no harm will be caused by the downloaded code through your browser.

- You are a High-Performance computing programmer who is interested in building the largest supercomputer (essentially out of idle machines on the Internet). In this context, users will have to be willing to share their CPU cycles to be part of such a supercomputer, and therefore you must provide them with security features to protect their machines.

- You need to establish a policy so that when you run downloaded applications, they would run within the context of that policy. For example, you can establish a policy such that all downloaded code is not allowed to establish network connections to any host.

Forces

- Users on the net are reluctant to visit web sites with applets embedded into their HTML pages, which forces some organizations not to use applets in their homepages.

- People are reluctant to allow outsiders use their machines as part of a global compute server because they are afraid that malicious outsiders may delete or alter the state of their files and applications.

- There are many restrictions on applets (e.g. they cannot read or write files or open network connections to random hosts) and this limits the usefulness of applets. This forces the development of a secure platform on which applets can do useful things.

- A user wishes to establish a policy that applets from site X may read files, whereas applets from site Y may read and write files. The level of granularity can be applied to any system resource: files, communication channels, port numbers, etc.

Solution

Define an extensible policy that can be customized and implemented easily, then establish a security policy that states what foreign code can and cannot do.

A security policy is a mapping from a set of properties that characterize running code to a set of access permissions granted to the code. The JDK1.0 introduced the `SecurityManager` class, which defines and implements a security policy by centralizing all access control decisions. Web browsers, such as Netscape's Navigator, and Microsoft's Internet Explorer use the `SecurityManager` class to implement a customized security policy that will be installed when executing untrusted code or applets, to reflect their own security policies. The `SecurityManager` is one of the layers of Java's sandbox. The essence of the sandbox is that local code is trusted and can have full access to the underlying file system. Likewise, downloaded remote code is untrusted and can access only limited resources provided inside the sandbox. JDK1.1 has introduced the concept of signed applets. A correctly signed applet is treated as trusted local code, and it can access the file system. Signed applets, together with their signatures, are delivered in the JAR (Java Archive) format.

While this evolving sandbox opens up interesting possibilities, it is till crude in the sense that all local Java applications enjoy full access to the underlying system resources while remote code is running in the sandbox, unless the code is signed by a trusted entity. This, however, has changed in Java 2 where signed code, in addition to remote, has been extended to local code. With the new security model in Java 2, all code (local and remote), signed or unsigned, will get access to system resources based on what is mentioned in a security policy file. A security policy file allows you to specify what permissions you wish to grant to code residing in a specified code source, and what permissions you wish to grant to code signed by specific persons. Note that the `SecurityManager` class (which was used to enforce the security policy in JDK1.0 and JDK1.1) has been kept in Java 2 for backward-compatibility.

Example

Consider the development of a global Web-based compute engine[Mah99] (or a supercomputer). In this distributed system, a client uploads programs to a compute engine that will in turn execute the code and send the results back to the client.

Implementing this system in Java would require the development of a custom class loader capable of loading arbitrary classes off the network. And, in order for your code to be executed by the compute engine, you must implement the `Compute` interface that has the following definition:

```
public interface Compute {
   public void run();
}
```

Now, if a malicious client is aware of a sensitive file (e.g. private.data) that exists on the compute engine's host file system, he may write a piece of malicious code to delete that file, for example. The malicious piece of code may look something like this:

```
public class Destroy implements Compute {
 public void run() {
   File f = new File("path to private.data");
   if (f.delete() == true) {
    System.out.println("File: "+f+"has been deleted");
   } else {
    System.out.println("operation is not allowed");
   }
 }
}
```

To protect the compute engine's host file system against this type of attack, we need to devise and implement a security policy that allows us to state what sort of instructions a program can and cannot do. This is accomplished by defining a security policy and implementing it by subclassing the `SecurityManager` abstract class. The following segment of code demonstrates how the Java interpreter's security manager works:

```
public boolean Operation(Type arg) {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
   sm.checkOperation(arg);
  }
}
```

This shows that when a public method call invokes the system security manager, the system determines whether the `Operation` is allowed. This means if a security manager is installed by an application, operations will be checked before they are performed. Once a security manager is installed it cannot be set to another security manager by some foreign code. In such a case an exception will be thrown signaling that no new security manager can be installed. On the other hand, if a security manager is not installed then foreign code will behave as local code and therefore can do anything (e.g. read and write files) just like local code.

The following snippet of code shows how to protect against deleting files, and disallowing client's code from quitting the compute engine's JVM:

```
public class EngineSecurityManager extends SecurityManager {
   private boolean silent = true;
   private boolean checkExit = true;
   private boolean checkDelete = true;

   EngineSecurityManager() {
     System.out.println("EngineSecurityManager started");
    }

  /**
   * The following operations are allowed. This is just
   * hypothetical though. More restricted access should be
   * imposed when working with class loaders.
    */
    public void checkConnect(String host, int port) { };
    public void checkCreateClassLoader() { };
    public void checkAccess(Thread g) { };
    public void checkExec(String cmd) { };

  /**
   * Check to see if a file with the specified name can be
   * deleted.
    */
    public void checkDelete(String file) {
      if(checkDelete) {
        throw new SecurityException("Cannot delete "+file);
      } else if (!silent) {
        System.out.println("File: "+file+" has been deleted");
       }
    }

   /**
    * Check to see if the JVM can be exited.
    */
    public void checkExit(int status) {
      if(checkExit) {
        throw new SecurityException("Cannot exit the JVM");
       } else if (!slient) {
        System.out.println("JVM is quitting");
       }
   }
}
```

To check whether or not it is ok to read a certain file, the Java API invokes the `checkRead()` method on the security manager and passes the path name of the file to be read as a parameter. Also, to check if a client can exit the JVM, the Java API invokes the `checkExit()` method. The `SecurityManager` class declares 28 of these checks, and new check methods have been added in Java 2. Once we have the `EngineSecurityManager` implemented, it must be installed by the compute engine. This can be easily done as shown in the `main()` method below:

```
public class ComputeEngine implements Runnable {
  // some methods go here
  public static void main(String argv[]) {
    EngineSecurityManager esm;
    try {
       esm = new EngineSecurityManager();
       System.setSecurityManager(esm);
    } catch(SecurityException e) {
       System.out.println("security manager already running");
    }
    new ComputeEngine();
}
```

Structure The diagram in Figure 1 shows the structure of the Security Policy pattern.

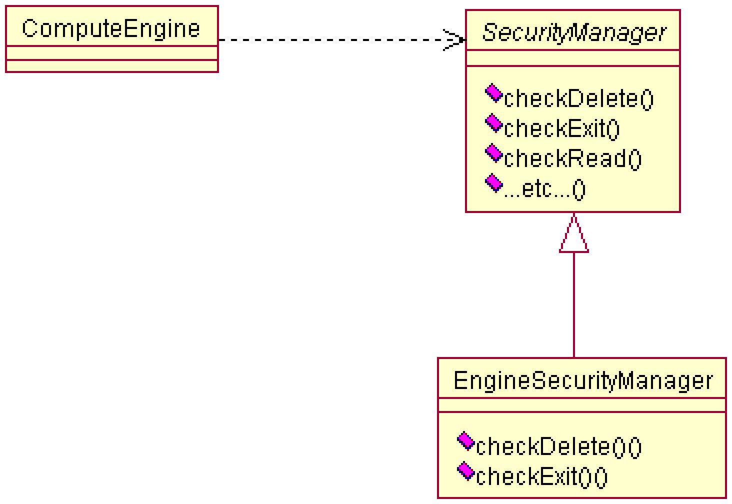

Figure 1: Structure of the EngineSecurityManager

The participants are:

- The `SecurityManager` abstract class that defines all the operations that clients may wish to protect their systems' against.
- The `EngineSecurityManager` concrete class, which is a direct subclass of the `SecurityManager`. It implements the operations that it needs to protect the system's against.
- The `ComputeEngine` which is really the client that collaborates with the `EngineSecurityManager` instances through the `SecurityManager`.

This technique works in Java 2 as well. However, Java 2 has a new feature known as *protection domains* that implements the site's security policy, which is defined by stating explicitly what permissions you want code running in a particular domain to have. The policy is represented by a policy object as instantiated from the class `java.security.Policy`. The policy object is a runtime representation of policy usually set up by the Virtual Machine at startup time (much like the `SecurityManager`). A policy object (in plaintext format) consists of a series of grant clauses, an example policy object is shown here:

```
grant codeBase "http://www.javacourses.com/*", signedBy "Qusay" {
  permission java.io.FilePermission "/tmp/*", "read";
  permission java.net.SocketPermission "*", "connect";
};
```

This security policy file states that any class downloaded from the site www.javacourses.com and signed by Qusay is allowed to *only* read (but not write or delete) any file in the directory tmp. It can also open a socket connection to any host. Now, to specify that this policy (assume it is saved in the file policy.all) should be used when invoking an application, the property-defining –D flag can be used as follows:

```
% java –Djava.security.policy=<path to>security.all
```

Resulting Context

The Security Policy pattern introduces several benefits for building secure distributed and mobile code based systems. Users will gain a level of comfort from knowing that they are protected and how their machines are being used. However, there are some limitations to this pattern:

- *Requires the existence of a framework*. The security policy pattern uses the `SecurityManager` as its framework. Also, the security policy pattern uses the advanced security features in Java 2 (e.g. protection domains). However, it should not be hard to implement similar things in other languages, as it is done in Safe-Tcl[OLW98].

- *A security policy can be set by a user* (as in Java 2). This idea has its own disadvantages as users may not know what exactly they are granting code to do. It is an error-prone task as any mistakes made could potentially translate into security holes at runtime.

- *No perfect world*. The security policy pattern does not address all potential threats posed by mobile code. For example, an activity of malicious code that is not addressed here is allocating memory (or creating new threads) until it runs out. This type of attack is called *denial of service,* as it denies end-users from using their own machines.

- *If a signer is honest, the code is secure*. A security policy file in Java 2 allows you to specify what signed code can and cannot do. One myth about code signing is if signer is honest, the code is secure. However, all the signature tells us is who signed the code, and it says absolutely nothing about the code's security. Certification authorities and schemes may begin to change the way this works[MF99].

- *Multiple security policies*. Sometimes it is better to have multiple security policies rather than just one policy that includes all the features that are safe for applets. Multiple security policies are needed because safe features do not compose: if feature X is safe, and feature Y is safe, then the combination of X and Y is not necessarily safe[OLW98]. For example, it is safe for an applet to open a socket connection outside the firewall as long as the applet cannot communicate with hosts inside the firewall. It is also safe for an applet to read files, as long as this is the only communication the applet makes outside its interpreter. However, if an applet has access to both of these features then it can transmit local files outside the firewall, which is a breach of both, security and privacy. Creating multiple security policies, however, is error-prone for naïve users as it requires a full understanding of how this is done.

Rationale

The security policy pattern resolves the forces mentioned above as follows:

- Users may feel their systems are protected. If users know that their web browser enforces a security policy that protects their files and applications, they may not mind visiting homepages with applets embedded in them. Major Web browsers (e.g. Netscape Navigator and Microsoft Internet Explorer) devise and implement a security policy by subclassing the `SecurityManager` class.

- People might be willing to contribute their idle CPU cycles to be part of a global compute engine if they are guaranteed that their files and applications will not be altered. Such people can define their own security policies to state what foreign code, running on their machines, can and cannot do.

- Allowing applets to read and write files and open network connections increase the usefulness of applets. With the security policy, users can establish a policy that states what applets can and cannot do.

- Furthermore, users can specify what applets coming from a particular site are allowed to perform. For example, they can establish a security policy which states that applets coming from site X can read files only and applets coming from site Y may read and write files. The same can be applied to communication channels and other system resources.

Known Uses

The pattern described in this paper has been used in a number of systems. The `java.lang.SecurityManager` abstract class serves the same purpose as this pattern, and it can be used to implement this pattern. A number of distributed frameworks implement their security policies by subclassing the `SecurityManager` class, use the new policy-based security features in Java 2, or define their own security policy-based architecture. For example:

- Netscape's Navigator and Microsoft's Internet Explorer implement the Security Policy Pattern by subclassing the `SecurityManager` class.

- Java Remote Method Invocation (RMI) implements a security policy and provides a default security policy (the `RMISecurityManager`) that must be installed by the server application; otherwise no class loading for RMI classes is allowed.

- ObjectSpace's Voyager customizes the `SecurityManager` class by providing the `VoyagerSecurityManager` that implements the security policy pattern.

- A Web-based Compute Engine[Mah99] implements its security policy by subclassing the `SecurityManager` class.

- In the Aglets Workbench[KLO98], a secure aglet system should implement the overall effect of all security policies involved that have been defined by principals (e.g. AgletOwner, ContextMaster, etc.). A policy database represents the security policy defined by the context master; security preferences represent the security policy defined by the aglet owner.

- Jini provides security by relying on a security policy that is defined in a text-based file that describes what actions mobile code can and cannot do.

- Safe-Tcl[OLW98], a mechanism for controlling the execution of programs written in the Tcl scripting language, uses this pattern to allow a variety of security policies to be implemented within a single application, and it supports both policies that authenticate incoming scripts and those that do not.

Related Patterns

Most of the Java-based distributed frameworks employ a security policy by either implementing a custom `SecurityManager` (as in JDK1.0 and 1.1) or writing a text-based security policy (as in Java 2). The Security Policy pattern uses the adapted `SecurityManager` class inspired in the Strategy pattern[GHJV95].

Acknowledgments

I would like to thank Federico Balaguer for shepherding this paper; his constructive comments helped me improve the pattern. I also want to thank Dwight Deugo and Robert Hanmer for their comments, which helped me to improve the presentation of this pattern.

References

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[KLO98] G. Karjoth, D. B. Lange, M. Oshima. *A Security Model for Aglets*. In G. Vigna (editor) Mobile Agents and Security. Springer-Verlag, Germany, 1998.

[Mah99] Q. Mahmoud. *The Web as a Global Computing Platform*. In Proceedings of 7th International Conference on High Performance Computing and Networking Europe, Amsterdam, The Netherlands, April 1999, Lecture Notes in Computer Science, pp. 281-290.

[MF99] G. McGraw, E. W. Felten. *Security Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

[OLW98] J. K. Ousterhout, J. Y. Levy, B. B. Welch. *The Safe-Tcl Security Model*. In G. Vigna (editor) Mobile Agents and Security. Springer-Verlag, Germany, 1998.