Global Information Assurance Certification Paper

Joe Barrett
GIAC Security Essentials Certification (GSEC) Practical Assignment
Version 1.4b, Option 1
May 29, 2004

Making Stand-Alone Java Applications More Secure

Abstract
Java has made a name for itself for its cross-platform abilities and ease of use for network applications. Its object-oriented features and extensive application programming interface (API) also make it well suited for stand-alone applications which only run on a single workstation or isolated local area network. The majority of business applications are not web-enabled. This paper explains the structure of the Java security model, and ways this model can be used to enhance the security of stand-alone Java applications. The Java security model is composed of the language syntax and semantics, Java Virtual Machine and the Java API, and additional security mechanisms used for authentication, authorization and cryptographic services. Java developers can produce more secure applications by having a thorough understanding of the Java language, knowing how the Java Virtual Machine can prevent malicious code from damaging their applications, fine-tuning application security with policy files, using Java security mechanisms correctly, testing the application before release and staying educated on the latest Java security bugs and patches.

1 Introduction

Java is an object-oriented programming language, developed by Sun Microsystems and released in 1995. Java is popular in part because of its cross-platform capabilities – a compiled Java program can be run on multiple operating systems. Java has been used extensively in both stand-alone and web applications. Java is also popular for client-server applications, such as interaction with databases. New technologies, such as Web Services, also use the Java language. Java programs are compiled into "bytecodes", a low level machine language. A Java Virtual Machine (JVM), which is installed either on top of the underlying operating system or a web browser, executes a Java program by reading the bytecodes. The language syntax and semantics (meaning) was designed with security in mind. Also, several security mechanisms are available for developing an even more secure application.

This paper will concentrate on giving developers some ideas and methods on how they can make their stand-alone Java applications more secure. This information can be extended to other types of applications. However, applications that run over a network or access a server have a host of other security concerns, such as network sniffing and protecting the integrity and confidentiality of databases. Despite the popularity of network applications, the majority of business applications is not web-enabled and could be considered stand-alone applications (Taylor, Buege, and Layman).

Stand-alone applications run on a single workstation, which may be connected to an isolated local area network (LAN). These applications are vulnerable from attackers outside the organization, if the workstation has Internet access either through a dial-up connection or dedicated line. If the workstation has a modem (with or without Internet capabilities), it can be accessed from the outside. Modems should have "auto-answer" turned off.

So, if a stand-alone application is mostly protected from the outside, it is safe from attack right? Wrong! Aside from intruders that manage to hack their way into your system, applications can be compromised by regular users, either accidentally or on purpose. Users do not always have the best intentions in mind while using an application. Attackers can be grouped into the following: external attackers, deliberate internal attackers and accidental internal attackers. External attackers include hackers and competitors. Hackers generally want to destroy data, use the system to attack other systems, or deface public information. This is usually done for their own personal satisfaction. Unlike external attackers, internal attackers usually have permission to access and use the application. Deliberate internal attackers include disgruntled, malicious or unethical employees, contractors, or espionage specialists who have infiltrated the organization (Taylor, Buege, and Layman). They may want to do the following: steal or modify confidential information, reverse-engineer the application, damage the application or run the application as another user. Accidental internal attackers make up the largest group. This group is made up of novice or untrained users, overworked system administrators and software developers who release applications without the proper testing and security controls. On these stand-alone applications, internal attacks are more common than external attacks (Taylor, Buege, and Layman).

If network and operating system security is in place, application security is unnecessary, right? Wrong again! Network and operating system security is very important, but imperfect. It will not stop legitimate users or even hackers that manage to gain access to the application's workstation. Application security should be part of a defense-in-depth strategy, which can include firewalls, intrusion detection systems, virus software, and the latest security patches to software and operating systems. Developers should consult with their organization's security policy, to decide what security mechanisms are necessary and appropriate for their applications. They need to be knowledgeable of who is going to have access to the application, what the application will be used for, how sensitive and valuable the information is that the application generates and stores, and how expensive it is to develop the application. This will help the developer to estimate the risks involved with using and developing the application; the risk is computed by multiplying the vulnerabilities, threats and impacts together. After the risk is determined and the organization's security policy has been consulted, the developer can then start writing source code. Not every Java security mechanism will be needed. For example, if users do not

needed to sign in/sign off an application, there is no need to use the JAAS (Java Authentication and Authorization Service).

So how does a Java developer go about writing a secure stand-alone application? First of all, the developer needs at least a basic understanding of the Java security architecture. Second of all, general coding guidelines should be followed (such as declaring a class's methods and fields private unless absolutely necessary). A good understanding of the Java language is a must. Thirdly, the developer should implement any required security mechanisms (such as access control and encryption) correctly. Before the application is released to the users, it needs to be thoroughly tested. Finally, the developer should keep abreast of the latest Java bulletins, which will describe the latest security-related bugs. The JVM should be updated or patched if the correct version is a security risk. Developers should know the version number and source (Sun Microsystems, Microsoft, Netscape, etc.) of the JVM their system is running.

This paper will list and describe some fairly easy to implement coding guidelines that will enhance the security of an application. Java's security architecture will then be explained. After that two, important Java security mechanisms will be introduced - authentication/authorization and cryptography. At the end, the importance of testing and logging will be emphasized.

2 General Coding Guidelines

A Safer Language

Java was designed to be "safer" than some other languages, like C and C++. So what does it mean that a language is safer? A language is safe when it disallows the program from misusing data types, overwriting protected memory or using variables before they have been initialized properly. A safe language also allows the developer to hide implementation details and control access to system resources, such as network connections and input/output. Java is considered safe in these regards, though it is not perfect. Java developers still need to use the language features correctly. Java also has exception handling built in, allowing the application to detect and handle errors easier. Use of the String data type helps prevent buffer overflows, a major security concern.

In C, C++, and some other languages, the programmer is tasked with making sure that data types are used properly. Casts from one type to another must be legal. This includes both primitive data types (integer, float, etc.) and classes. Whenever a variable or object is cast to another data type, this casting operation is checked to make sure that it is valid. Otherwise, a ClassCastException is thrown. This helps prevent memory from being illegally accessed and access control mechanisms from being violated.

Numerous bugs in C and C++ programs can be traced to poor memory management. Developers may try to delete objects from memory too early (the program then tries to access objects that no longer exist) or too late (the

references to the objects are no longer in scope). With complicated programs, using multiple developers, it may be difficult to keep track of all the dynamic memory that is allocated and must eventually be deleted. The incorrect use of pointers and arrays in some languages can corrupt a program's internal data or access another process's memory. Thankfully, Java comes with its own automatic memory manager, known as the garbage collector. Java developers generally do not need to worry about managing memory, since the garbage collector occasionally deletes objects that are no longer needed or referenced. Objects can only be accessed indirectly through references instead of pointers, so that memory cannot be accessed directly. In Java, an out-of-bounds array index will throw an exception, also disallowing illegal memory accesses. Invalid array indices are a frequent source of bugs in other languages.

In C and C++, the programmer must initialize all variables or objects before they are used, or else "garbage" values should be expected. Java objects are guaranteed to be initialized, at least to default values. Primitive fields of a class are always initialized when an object of the class is created, however local variables in a class's methods are not. If you forget to initialize a local variable before use, the Java compiler will at least give an error. A Java array of objects is an array of references, which are initialized to null. A Java array of primitives is initialized by assigning the value zero to all of the elements in the array. Despite all these built in precautions, it is still best to explicitly initialize all objects and local variables before use. That is because default values may not be correct or even valid for the particular program.

Like C++ and some other object-oriented languages, Java enables implementation hiding. This means hiding details from the client programmers – the ones using the classes. Variables (objects and primitives), methods and classes can be declared with private, protected, public or package access levels. Private access means access only by the containing class. Protected access is access by the containing class, the class's subclasses, and classes in the same package. Public access is access by all classes. Package access is access by classes in the same package that it is in. Related classes are often grouped together into packages.

Java controls access to system resources in part by ensuring safe data types, use of namespaces, and with variable, method and class access levels. Also, whenever a system resource is requested by a method, the program checks to make sure that the method has the proper permissions, based on the code's owner, origin and user.

Exception handling is built into Java. Invalid operations and security violations can be caught and handled before they cause any damage. Some exceptions should be masked under certain situations. When authenticating a user, an exception may be raised when the user is not found. In this case, a more general exception should be rethrown to indicate a general login failure. If this is

not done, an attacker could use the exception output to determine if a user is valid.

String objects use UTF8 for their internal representation. Every String object has a length and a table of characters storing current information. Because of the UTF8 format and runtime checks, String buffer overflow attacks cannot occur as long as String operations are done at the Java language level (Last Stage of Delirium Research Group). However, native method calls are not necessarily safe. Also related to buffer overflows, runtime checks and the Java bytecode verifier help to prevent Java programs from illegal stack frame accesses. String objects are immutable, so a hacker could potentially read String objects from memory. When storing sensitive text such as passwords and user names, it may be best to use an array of characters (char array). That way, the array can be overwritten when the sensitive information is no longer needed.

Much of the rest of this section, General Coding Guidelines, is taken from Sun's "Security Code Guidelines" document and Java World's "Twelve Rules for Developing More Secure Java Code" article by Gary McGraw and Edward Felten.

Privileged Code

Normally, code cannot access system resources without the proper permissions. Code that lacks the proper permissions yet needs access to system resources, can be put inside a privileged block. As one can imagine, this can be a dangerous thing if not used carefully. Privileged code should be as small as possible and privileged blocks should only be used if security exceptions would otherwise be thrown (Sun, Security Code Guidelines).

Public methods that use tainted variables should not wrap around privileged code (Sun, Security Code Guidelines). A tainted variable is a variable passed in as a parameter and not controlled by the privileged code. This might allow anyone to use the privileged code and its associated permissions to access sensitive system resources. Instead, make sure these methods are private. Further information on privileged blocks is available at http://java.sun.com/j2se/1.4.2/docs/guide/security/doprivileged.html.

Methods and Fields

All variables should be made private, except for good reason. Use "get" and "set" methods instead, to insure data always stays in a valid state. Especially avoid using non-final public static variables, since code can change the values without first going through the proper permission checks (Sun, Security Code Guidelines).

Package Access

It is possible for malicious code to gain access to a class's fields and methods by defining new classes of its own within the class's package. The Security Code Guidelines document from Sun explains how to prevent this:

1. The package can be protected from insertion of rogue classes by adding the following line to the java.security properties file:
   …
   package.definition=Package#1 [,Package#2,…,Package#n]
   …
   This causes a class loader's defineClass method to throw an exception when an attempt is made to define a new class within these packages, unless the code has been granted the following permission:
   …
   RuntimePermission("defineClassInPackage."+package)
   …
2. Another way to protect against package-insertion is by putting the package's classes in a sealed JAR file. (see http://java.sun.com/j2se/1.4.2/docs/guide/extensions/spec.html)
   By using this technique, no code can be granted permission to extend the package and hence there is no need to modify the java.security properties file.

Access to a package's fields and methods can be restricted to only specified code. Sun's Security Code Guidelines document also explains how to restrict package access:

> This can be done by adding the following line to the java.security properties file:
> …
> package.access=Package#1 [,Package#2,…,Package#n]
> …
> This causes a class loader's loadClass method to throw an exception when an attempt is made to access a class from these packages, unless the code has been granted the following permission:
> …
> RuntimePermission("accessClassInPackage."+package)
> …

Initialization

Before using an object, there is a way to verify that the constructor has initialized the object. McGraw and Felten (1998) show how to do this:

> Make all variables private. If you want to allow outside code to access variables in an object, this should be done via get and set methods. (This keeps outside code from accessing noninitialized variables.)…Add a new private boolean variable, *initialized*, to each object. Have each constructor set the initialized variable as its last action before returning. Have each nonconstructor method verify that *initialized* is true before doing anything. (Note that you may have to make exceptions to this rule for methods called by your constructors. If you do this, it's best to make the constructors call only private methods.) If your class has a static initializer, you will need to do the same thing at the class level.

Make classes and methods final

Classes and methods should be made final unless necessary. This prevents an attacker from extending the code through inheritance.

Inner Classes

Do not use inner classes. Inner classes are translated by the compiler into ordinary classes, and can be accessed by any code in the same package (McGraw and Felten). The inner class can access the enclosing outer class's fields, even if they are private. The outer class's private fields are made into package scope, in order for the inner class to access them.

Class Names

Avoid class names in code. Unfortunately, Java forces the programmer to use class names in variable declarations, instanceof expressions and exception-catching blocks (McGraw and Felten, 1998). This opens the application up to mix-and match attacks, in which the attacker either constructs a new library that links some of your signed classes together with malicious classes, or links together classes that were not meant to be used together. Sometimes it is necessary to determine if two objects were created from the same class, or see if an object is a member of a particular class. Different classes can have the same name, so a better way is to compare class objects for equality directly. McGraw and Felten (1998) explain how to do this:

> For example, given two objects, A and B, if you want to see whether they are the same class, use this code:
>
> ```
> if (a.getClass() == b.getClass()) {
>     // objects have the same class
> } else {
>     // objects have different classes
> }
> ```

You should also be on the lookout for cases of less direct by-name comparisons. Suppose, for example, you want to see whether an object has the class "Foo." Here is the wrong way to do it:

```
if (obj.getClass().getName().equals("Foo"))  // Wrong!
        // objects class is named Foo
} else {
        // object's class has some other name
}
```

Here's a better way to do it:

```
if (obj.getClass() == this.getClassLoader().loadClass("Foo")){
        // object's class is equal to the class that this class calls "Foo"
} else {
        // object's class is not equal to the class that
        // this class calls "Foo"
}
```

Making copies of objects

Prevent an attacker from redefining your clone method by making objects noncloneable. An attacker can create new objects with the object cloning mechanism, even if the constructors are not executed. An attacker can define a subclass if the class is not cloneable, and make the subclass implement the java.lang.Cloneable interface. McGraw and Felten (1998) show how to make objects noncloneable and prevent an attacker from redefining the clone method:

```
public final void clone() throws java.lang.CloneNotSupportedException {
throw new java.lang.CloneNotSupportedException();
}
```

If you want your class to be cloneable, and you've considered the consequences of that choice, then you can still protect yourself. If you're defining a clone method yourself, make it final. If you're relying on a nonfinal clone method in one of your superclasses, then define this method:

```
public final void clone() throws java.lang.CloneNotSupportedException {
super.clone();
}
```

Serialization

When an object is in a serialized state, it is outside the control of the JVM environment and its security (Sun, Security Code Guidelines). If an attacker can serialize your objects or obtain your serialized objects, the attacker can read the internal state of your objects. This includes any private fields and other objects

that are referenced. McGraw and Felten (1998) explain how to make objects impossible to serialize, by declaring the writeObject method this way:

```
private final void writeObject (ObjectOutputStream out) throws
java.io.IOException {
throw new java.io.IOException("Object cannot be serialized");
}
```

Classes should also be made nondeserializeable, to prevent an attacker from creating a sequence of bytes that can deserialize to an instance of the class. You then have no control over what state the deserialized object is in. McGraw and Felten (1998) explain that you can make it impossible to deserialize a byte stream into an instance of a class by declaring the readObject method like this:

```
private final void readObject (ObjectInputStream in) throws
java.io.IOException {
throw new java.io.IOException ("Class cannot be deserialized");
}
```

If you must use the Serializable interface, use the transient keyword for fields that contain direct handles to system resources and contain information relative to an address space (Sun, Security Code Guidelines). This prevents others from saving or restoring the fields. In order to guarantee that a deserialized object has a valid state, a class can define its own deserializing method and use the ObjectInputValidation interface. Classes that define their own serializing method should not pass an internal array to any DataInput/DataOutput method that takes an array as a parameter (Sun, Security Code Guidelines). An attacker could then subclass ObjectOutputStream and overwrite the write (byte [] b) method. This would let the attacker access and modify the private array (Sun, Security Code Guidelines). Alternatively, the bytestream produced by the serialization package can be encrypted to prevent another from reading a serialized object's private state.

Signing Code

Avoid signing code, because this will give the code special privileges. Put signed code into one archive file. This will help prevent an attacker from carrying out a mix-and-match attack (McGraw and Felten, 1998).

Reverse engineering code and code obfuscation

Attackers may be interested in either obtaining original source code or reverse engineering bytecode, to obtain secret information inside the source code (such as passwords or keys) or better understand how the application works (to make it easier to attack). Source code may be proprietary or contain proprietary technology, so a competitor may be interested in viewing it. Tools are available for free that can decompile Java bytecode into a readable format. One such tool is JODE, available at http://jode.sourceforge.net. Code obfuscation, modifying

source code to make it more difficult to read, will help but not prevent decompilation. Code obfuscators can do the following: rename variables to cryptic values, rename and repackage class files and make slight modifications in program flow (Taylor, Buege, and Layman).

To really prevent an attacker from decompiling classes they must be encrypted, then decrypted right before they are used. The JVM must be able to read the class files, so in order to do this, one must write a custom class loader by following these steps:

1. Write a program to encode a class file.
2. Extend the URLClassLoader class and override the findClass method.
3. Write a driver program that will load a particular class.

For most applications this is overkill, but Taylor, Buege, and Layman explain how to do it.

Applications can be run with debugging information. This helps the user in case of problems, but also helps an attacker understand the inner workings of an application. Debugging should be turned off when releasing an application to the users. Debugging information can be turned off the Java compiler, javac, this way:
javac –g:none Program.java, where Program is the name of the application.

3 Java Security Architecture

The Java security model has evolved and improved over the years. The first release of the Java Development Kit, JDK 1.0, featured the "sandbox" security model. Mobile code, such as applets, was given severely restricted access to a system's resources. Local applications were given unlimited access. This obviously limited flexibility in enforcing a system's security policy. Security in JDK 1.1 was a slight improvement. An applet could now be signed, giving it unlimited access like a local application, assuming that the signed applet was trusted. Unsigned applets stayed in the sandbox. Before JDK 2, security for applications could be fine-tuned only with substantial programming, by subclassing and customizing the SecurityManager and ClassLoader classes. With JDK 2, the security model is much improved, making it easier to provide fine-grained security. Local applications run unrestricted by default, but can be limited if necessary.

The Java security model can be thought of as composed of four layers, as stated by Herholtz:

> Layer 1 – Java Programming Language (ensures semantics/syntax, memory access protections, strong typing (safety) e.g. no forged pointers, buffer overflow, memory leakage. Segregates name-spaces (memory) of local and network obtained resources.)

Layer 2 – Java Virtual Machine (JVM) normally resident on a Client's Web browser (ensures typed memory access, byte-code verifier, memory garbage cleansing/reallocation).
Layer 3 – Libraries/ClassLoader(s). Three types with 2 basic functions: Internal, Class Loader Objects (applet, RMI, Secure) Roll your own classes (access to files and network resources – implements network classes/objects, methods/functions, disallows unauthorized access, maps names to class objects, invokes security for necessary classes) 1) Instantiates bytecode as classes. 2) Manages namespace
Layer 4 – Security Manager/Runtime environment (defines and implements security policy, centralizes access control). Configurable portion of model, client or platform independent (discretionary access control). Layer 4 must be specifically SA configured to invoke security resident of bottom three layers. Layer 4 allows tailoring of the Java Security Model to specific security policies.

The layers of the Java Security Model do not support each other; they are more like links in a chain (Herholtz). When one link is broken, security is violated.

The Java Security Architecture is not perfect, and has been criticized for lacking a formal proof of its correctness and evolving nature, among other things. Herholtz criticizes the Java model on these grounds:

1. Language and Bytecode Flaws – Use of data types in Java cannot be proven correct, leading to occasional bugs in class verification.
2. Strength of mechanism – The Security Manager is not adequately protected by safe typing, is not always invoked, cannot be verified and is not tamperproof.
3. Simplicity of mechanism – Method call stacks can be traversed by multiple threads of execution with different levels of trust, and byte code can traverse through the Java security model in at least three ways.
4. Complete Mediation – Native code called by untrusted code have circumvented security mechanisms.
5. Auditing – Recording and logging is not automatically done when an exception is thrown or when a security breach occurs.
6. Verification – There is no verification of the Java security model or policy. The security model lacks a formal definition and specification.
7. Ease of use – Proper configuration and management of fine-grained security controls is not easy.
8. Tamper resistance – Flaws in the language have allowed tampering to occur in the past, and the flexible nature of the security mechanisms may inadvertently allow attackers to tamper with the JVM.
9. Ambiguity – There is a lack of security system requirements.

So, what should this mean to the Java developer? A few of these criticisms seem mostly theoretical, in this author's opinion. They do emphasize the following: a sound understanding of the Java language, its security model and

mechanisms is necessary, there is a learning curve to understanding and implementing Java security properly, developers need to know about the latest bugs and successful attacks against Java applications, logging should be used to know who is using the application and what they are doing with it, and finally security is unfortunately not perfect.

Java security will now be described in general terms. Some features of the security model will then be explained in more detail. Java security starts with the bytecode verifier, which examines all untrusted bytecode before it is run through the JVM. It makes sure that the class files are in the correct format and conform to the standards of the Java programming language. The Class Loaders do further checks, to make sure that malicious or damaging classes are not loaded into the JVM. The Security Manager and the Access Controller work together, to make sure that code does not access resources without the proper permissions. Classes are grouped together into protection domains, which have similar permissions. The system domain is given special privileges and includes system code and classes in the CLASSPATH (where local code resides). One or more policy files contain a list of "grant" entries. A grant entry contains an optional list of signers, an optional codebase (source of the code), an optional list of principal (user) class name/principal name pairs, and one or more permissions followed by a target and action. The signers' values are actually aliases for certificates containing public keys, corresponding to private keys used to sign the certificates. The subject that signed the code may not be the same as the author or distributor of the code. The signer just vouches for the code. In addition, Java includes several security mechanisms (JCE - Java Cryptography Extension, JAAS) and tools (keytool, jarsigner, Policy Tool, etc.). Running code is allowed to access certain system resources based on its class, origin, signer and user.

From McGraw and Felten (1999, Chapter 2, section 6), class files contain the following:

- The magic constant (0xCAFEBABE)
- Major and minor version information
- The constant pool (a heterogeneous array composed of five primitive types)
- Information about the class (name, superclass, etc.)
- Information about interfaces
- Information about the fields and methods in the class
- Debugging information

The bytecode verifier attempts to prove that code is in the right format before it is run in the JVM. It is only run for untrusted classes. The class file that is about to be loaded is checked for correct length, magic number (0xCAFEBABE) and format. The verifier ensures that private, protected, public, and package access to classes, methods, and variables is respected. If there is a problem in a class, it is not loaded into the JVM. It performs four passes when checking a class.

The first pass makes sure that the class file format is correct and can be parsed. During the second and third passes, code is checked to see that it conforms to Java programming rules of syntax and semantics. During the fourth pass, symbolic references are resolved into direct references; this pass occurs during runtime.

Class loading in Java follows a hierarchical structure. At the root is the primordial, or system class loader. Class loaders are loaded by other class loaders, except for the primordial class loader. The primordial class loader will load the classes required for all Java programs. A child class loader delegates the task of loading a specific class to its parent. If the parent cannot load a class then it allows the child to. This helps prevents class spoofing attacks. Before a class is loaded, a cache is checked to see if the class has already been loaded. The Security Manager also checks to see whether a class is allowed to run. The creation of class loaders is severely limited; special permission is needed by the Security Manager to create a class loader. Class loading is usually done right before the class is needed in the running program. Class Loaders are responsible for namespaces, so that identical identifiers can exist without interference.

The Security Manager (java.lang.SecurityManager class) works with the Access Controller, and is called to determine whether code can access resources such as files or network connections. The Security Manager calls “check” methods to determine access permissions. For example, a call to the checkRead method will determine whether a file can be read. Anytime a potential dangerous operation is about to be performed, a check is made. If the check fails, a security exception is thrown and the method is not called. The Security Manager is not automatically installed in an application; it can be listed as a command line option. Decisions on access to resources use the security policy in effect, which is based on the contents of the policy file(s) and privileged blocks.

A protection domain is composed of a set of principals (users) and types of permissions. A group of classes is mapped into a protection domain; the mapping occurs before the classes are used and cannot be changed. Exceptions are thrown when a class in a protection domain tries to access a resource it does not have permission to use. A protection domain is associated with a codesource. Classes with the same codesource and signers are placed into the same protection domain. A system protection domain type controls access to system resources, while an application protection domain type controls access to portions of an application. Code that is part of the Java API is trusted, and given all permissions. If a Security Manager is loaded with an application, the application has no permissions to resources other than those granted in the security policy.

Security policy is set by the security policy file(s) in effect. The policy file (sometimes called the policy configuration file) contains entries that grant

permissions to protection domains. This file plays a large part in securing an application. The JVM finds the policy files from the security properties file. Three security policy files can be used – the systemwide policy file, the user policy file in the user's home directory, and any application policy file that the application chooses to load. To specify an additional or different policy file when starting up an application, use the "–Djava.security.policy" command line argument. This set the value of the java.security.policy property. Here is the format: "java -Djava.security.manager -Djava.security.policy=*someURL* SomeApp". The system policy file grants system wide code permissions. It grants all permissions to standard extensions, lets anyone listen on unprivileged ports, and allows any code to read standard properties that are not security sensitive (Sun's Default Policy Implementation and Policy File Syntax). Access to the policy files should be limited. Policy files can be edited by a text editor or created using the Policy Tool, which implements a graphical user interface. It is possible to prevent system properties from being set on the command line. This prevents attackers from loading their own security policy file.

Even though there are a number of Permission classes built into the Java SDK, additional Permission classes can be created. To see a description of the built in Permission classes, along with their security risks, go to http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html. This document also lists methods in the Java SDK that require permissions.

From Sun's Default Policy Implementation and Policy File Syntax, the syntax for grant entries in a policy file is as follows:

```
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 permission_class_name "target_name", "action",
                signedBy "signer_namees";
        …
};
```

Refer to this document for more detail on policy files and several policy file examples.

4 Brief Overview of JAAS and JCE

In J2SE 1.3, JAAS and JEC were optional packages. In J2SE 1.4, these are part of the core security structure. JAAS and JCE are "pluggable" in that the underlying security implementation and algorithms are kept separate from the Java code that serves as an interface. Security implementations can be changed without changing Java code.

The JAAS API does not replace any of the Java security, but provides additional functionality. Authentication is the process of determining that a subject is who they say they are. Authorization determines whether a request from an authenticated user is allowed. After a subject is authenticated, a javax.security.auth.Subject object is populated with its associated identities, or Principals. Subjects can obtain public or private credentials, which are security-related attributes. Any object can represent a credential. Sensitive credentials, such as private keys, are stored within a private credential set. Credentials that are shared, such as public keys, are stored within a public credential set.

The LoginModule class validates a user and assigns principals to the subject. The CallbackHandler class communicates with the user to valid the user's identity. The Subject class is the target of the login process. The Principal class is an entity that is granted access rights.

Authentication can be performed several ways, but the following steps are necessary according to Sun's JAAS Reference Guide:

1. An application instantiates a LoginContext.
2. The LoginContext consults a Configuration to load all of the LoginModules configured for that application.
3. The application invokes the LoginContext's login method.
4. The login method invokes all of the loaded LoginModules. Each LoginModule attempts to authenticate the subject. Upon success, LoginModules associate relevant Principals and credentials with a Subject object that represents the subject being authenticated.
5. The LoginContext returns the authentication status to the application.
6. If authentication succeeded, the application retrieves the Subject from the LoginContext.

According to Taylor, Buege, and Layman (pg38), the following steps are needed for JAAS authorization:

1. Create appropriate security policy file entries.
2. Create a custom Permission class, a subclass of the java.security.Permission class.
3. Create a custom action class, an implementation of java.security.PrivilegedAction.
4. Execute the static Subject doAsPrivileged method, passing the Subject instance containing the principals required and the custom PrivilegedAction, along with an optional AccessControlContext.
5. Within the body of the PrivilegedAction run method, access the SecurityManager and call the checkPermission method using the custom Permission class.

CES uses a "provider"-based architecture. It can be used for encryption/decryption, ensuring the integrity of messages, and nonrepudiation

(proving that the sender is the one who sent the message). The provider refers to a package that implements one or more cryptographic services. These include key generation, digital signatures and message digests. Symmetric, asymmetric, block and stream ciphers can be used for encryption/decryption. The default provider is SUN, from Sun Microsystems. Their package has several implementations, including the following: the Digital Signature Algorithm (DSA), MD5 and SHA-1 message digest algorithms, the SHA1PRNG pseudorandom number generation algorithm, and X.509 certificate generation.

The JCE API uses the factory design pattern to create objects, in which a constructor is not used. Instead, a getInstance method is called to return a reference to the object. A cipher object is set to one of four modes: ENCRYPT_MODE, DECRYPT_MODE, WRAP_MODE (wraps a key into bytes for transportation), and UNWRAP mode (unwraps a previously wrapped key).

5 Testing Programs

For an application to be secure, it must also be reliable. The application's input, output, internal data, and computations should be correct. How much the application needs to be tested depends on its value, complexity, and security concerns. Several methods of testing exist, include unit testing and manual testing by testing specialists. JUnit is a unit testing software program available for Java, available at http://junit.org. One of the most common causes of software bugs is improper input validation. Applications should be able to error check and handle the following bad input: incorrect data types (for example, you expect an integer and the user gives you a floating point number), out of range numbers (too large or too small), input that is too long or short, null values and characters that have special meanings for the operating system or underlying programming language.

After the application has been released to the users, logging can be useful for both application errors and security violations. The extensive Java Logging API is available for just these purposes. It can direct log messages to files, database, network sockets, and the console. It is possible to have separate Logging objects for applications and security, or separate Logging objects for each class. Different levels can be set for logging objects based on their importance, ranging from finest to severe.

Taylor, Buege, and Layman (pg117) recommend logging the following events:

- Successful and unsuccessful login attempts
- Logouts and application shutdowns
- Successfully accessing sensitive functionality
- Unsuccessfully attempting to access any functionality
- Severe application exceptions that could affect the integrity of application data or functionality

It is recommended, but often not followed, that tests should be written before executable code is written. This can improve the design of the software. Testing, or at least automated testing, should be done whenever code is updated.

References

Sun Microsystems documents:

1. Java Security Architecture, 1999, http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html

2. Summary of Tools for the Java 2 Platform Security, October 14, 2002, http://java.sun.com/j2se/1.4.2/docs/guide/security/SecurityToolsSummary.html

3. API for Privileged Blocks, April 30, 2001, http://java.sun.com/j2se/1.4.2/docs/guide/security/doprivileged.html

4. JAAS Reference Guide, August 8, 2001 http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html

5. JAAS Authorization Tutorial, http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnAndAzn.html

6. JAAS Authentication Tutorial, http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html

7. Security Code Guidelines, February 2, 2000 http://java.sun.com/security/seccodeguide.html

8. Default Policy Implementation and Policy File Syntax, April 20, 2002 http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html

9. jarsigner – Jar Signing and Verification Tool, 2002 http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/jarsigner.html

10. Java Cryptography Architecture, August 4, 2002 http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html

11. Java Cryptography Extension Reference Guide, January 10, 2002 http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html

12. Permissions in the Java 2 SDK, 2002 http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html

13. Policy Tool – Policy File Creation and Management Tool, 2002
http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/policytool.html

14. Security Managers and the Java 2 SDK, May 1, 2001
http://java.sun.com/j2se/1.4.2/docs/guide/security/smPortGuide.html

15. Java Logging Overview, November 26, 2001
http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html

SANS GSEC Practicals
1. Ankolekar, Vilas, Application Development Technology and Tools: Vulnerabilities and threat management with secure programming practices, a defense-in-depth approach, November 2003.

2. Herholtz, Matthew, Java's Evolving Security Model: Beyond the Sandbox for Better Assurance or a Murkier Brew?, March 2001.

Other Online Documents:
1. Clark, Mike, JUnit FAQ, April 22,2003,
http://junit.sourceforge.net/doc/faq/faq.htm

2. Last Stage of Delirium Research Group, Java and Java Virtual Machine Security Vulnerabilities and their Exploitation Techniques, version 1.0.0, October 2, 2002,
http://www.lsd-pl.net/documents/javasecurity-1.0.0.pdf

3. McGraw, Gary and Felten, Edward, Twelve rules for developing more secure Java code, December 1998
http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules_p.html

4. Srinivas, Raghavan, Java security evolution and concepts, Part 1: Security nuts and bolts, April 2000,
http://www.javaworld.com/javaworld/jw-04-2000/jw-0428-security_p.html

5. Srinivas, Raghavan, Java security evolution and concepts, Part 2: Discover the ins and outs of Java security, July 2000
http://www.javaworld.com/javaworld/jw-07-2000/jw-0728-security_p.html

6. Srinivas, Raghavan, Java security evolution and concepts, Part 5: J2SE 1.4 offers numerous improvements to Java security, December 2001
http://www.javaworld.com/javaworld/jw-12-2001/jw-1221-jdk4security_p.html

Books:
1. Eckel, Bruce, Thinking In Java, 3rd Edition, Prentice Hall, New Jersey, 2003.

2. McGraw, Gary and Felten, Edward, Securing Java, John Wiley and Sons, Inc., 1999.
Contents available at http://www.securingjava.com/TOC.html

3. Taylor, Buege, and Layman, Hacking Exposed J2EE & Java, McGraw-Hill/Osborne, Berkeley, CA, 2002.