Java Security Extensions for a Java Server in a Hostile Environment

David M. Wheeler
Adam Conyers, Jane Luo, Alex Xiong
Intel Corporation
5000 West Chandler Boulevard, Chandler, Arizona, 85226, USA
david.m.az.wheeler@intel.com

Abstract

The Java (TM) Virtual Machine is being used more frequently as the basic engine behind dynamic web services. With the proliferation of network attacks on these network resources, much work has been done to provide security for the network environment. Continuing work on firewalls, intrusion detection, and even access control have provided numerous insights and capabilities for protecting web resources. Java itself has received much attention in the security arena, and the Java 2(TM) Architecture has provided considerable in-roads to providing security services. However, this research has operated under the assumption that attacks only occur through the network, and not with direct access to the web server through a valid login. Little effort has been placed on securing a Java web server where the attacker has a valid login to the host machine. This paper describes specific security extensions developed for a Java Virtual Machine that provide assurance of correct system operation and integrity even in the presence of successful attacks on the underlying operating system.

1 Introduction

Java 2 security enhancements provide numerous security services, including security policy definition and enforcement, code authenticity protections, and authorization controls. These services are implemented in a way that makes them reliant upon the proper use and functioning of the access control mechanisms of the underlying operating system. If the operating system is attacked from an insider or an attacker who has compromised an account on the server, many of these Java security mechanisms can be easily overcome. Despite proper configuration, the use of firewalls and other network security mechanisms, web servers are compromised every day. Additional mechanisms can be added to Java to add resilience to the server even in the presence of an attack, removing Java's sole dependence on the access controls of the underlying operating system.

This paper explores some of the unique vulnerabilities in the runtime environment of the Java 2 JVM, and describes the details of custom extensions to the standard Java security architecture that can be utilized to provide increased assurance of correct system operation and integrity of a Java application server, even if assumptions of a safe server environment cannot be made.

This paper begins with an explanation of the motivation for this project. Section 3 introduces the vulnerabilities in Java's security mechanisms. Section 4 provides an informal set of security objectives used to describe the underlying security requirements for the Java 2 security extensions. These objectives are used to guide section 5 in describing the important aspects of the extensions. The paper concludes with a summary and areas for future work. A brief review of the security mechanisms available in the Java 2 environment is provided as an appendix for the reader who is not familiar with these concepts.

2 Motivation

Java security was designed to provide common functionality needed in both client and server environments [1]. For this reason, certain trade-offs were made in order to increase the applicability of Java to these differing environments and to meet ease-of-use requirements intended to ensure Java attains a pervasive stance among developers and users alike. The broad assumptions about ease-of-use and system configuration required to satisfy a general-purpose computing environment, do not translate well to a custom-built application server and result in system vulnerabilities adverse to a secure server.

Many of Java's strongest benefits also introduce some of Java's most serious security vulnerabilities. For example, Java's portable byte code, which is the basis for its "write-once, run-anywhere" slogan, is a primary basis for the ease with which Java can be reverse-engineered. And just as easily as intellectual property can be extracted from an application's class files, malicious classes can be inserted into a Java application. Attacks on unsigned JAR files and CLASSPATH modifications are well known, but

a knowledgeable attacker can do far more damage, even overcoming many of Java's security mechanisms.

Although it appears Java has provided several sophisticated security mechanisms to protect itself from these types of attacks, each mechanism is laced with designed-in backwards-compatibility weaknesses or implementation subtleties that can be manipulated to overcome those protections. Signed JAR files are susceptible to keystore replacement attacks, the JVM itself can be modified by adding or replacing trusted classes in the system JAR file (i.e. rt.jar), and even the security checks in the JVM can be disabled by merely setting a null security manager. The Java environment relies on the proper application of the access control mechanisms in the underlying operating system to thwart these attacks. While this is prudent and recommended, there are no secondary mechanisms to protect the system if this configuration is done improperly or fails due to some other system vulnerability. But even with an untampered installation, the security policy is applied at such a coarse granularity level that deployment of real applications with an effective, strong security policy becomes too difficult in practice.

The motivation for this work is derived from the design and implementation of a Java application server product with the requirement to protect the intellectual property in the server while building a system that enforces a known security policy, applies the least-privilege principle, provides resistance to malicious code, and resists tampering from malicious and benign users and administrators. The mechanisms in this paper are intended to satisfy these objectives even when assumptions about a safe environment cannot be made. The aim of this paper is to share the knowledge gained regarding some of Java's vulnerabilities, and describe our team's attempt to mitigate these vulnerabilities.

3 Java Security Vulnerabilities

This section introduces the set of vulnerabilities uncovered by the project team and addressed by the security extensions discussed in this paper. The vulnerabilities covered here, are best classified as ones of choice and design[1] rather than of flaw or defect. And while the descriptions of the mechanisms underlying these vulnerabilities do appear in the literature, discussions regarding their weaknesses and their impact to security are absent.

Many critical components of the Java environment are only protected by the underlying operating system's access control mechanisms. In some installations, the JRE files have no access control restrictions placed on them. This includes the configuration files used to define the security policy and critical security classes (java.security and java.policy), the extension directories that default to give all permissions to classes and JARs (jre/lib/ext and jre/classes[2]), and the JVM system classes JAR (rt.jar). Even the IBM JRE, which otherwise provides a fairly secure default installation, allows the creation of the jre/classes directory and assigns all permissions to any classes placed there.

This loose access control protection extends to many components, and most system administrators are not aware of the potential dangers posed by this situation. For example, the keystore used to hold the certificates for checking signed JAR files are typically provided by an application, or updated during an application installation. Most system administrators may not realize that if the keystore is modified, signed JAR files can be spoofed, and permissions can be granted to malicious code.

The loose access control on the jre/classes directory opens the possibility of all types of malicious code insertion attacks. Some possibilities include insertion of a malicious class loader, replacement of the policy manager, insertion of a weak security manager, or substitution of some application class. Since the jre/classes directory becomes part of the Xbootclasspath [3], it is one of the first places checked by class loaders when looking for classes. This is due to the way in which class loaders search for classes. The standard algorithm for loading classes, which is followed by the URLClassLoader and recommended in the literature [1][2][9], uses a delegation hierarchy in order to ensure system classes are loaded only by the system class loader, and JVM (boot) classes are only loaded by the bootstrap (null) class loader. Using the delegation hierarchy, a class loader will load a class only if its delegation parent has not loaded the class [8]. Effectively, the URLClassLoader will delegate class loading to the bootstrap class loader before attempting to load the class itself. Since the jre/classes directory is hard-coded in the Xbootclasspath by the JVM, any class existing in that directory, that conflicts with the package and class name of an application class, will be loaded from that directory instead of from the application directory or JAR.

As an example, an attack class placed in the jre/classes directory could operate according to the proxy pattern [12] for the class that it is replacing, but also perform other malicious acts. For example, the class could set a new policy manager that grants permissions to other malicious classes; the class could override the main

[1] It should be noted, that in most cases, the mechanisms described here were designed to be extensible, so that better protections could be provided for specific situations. Many of these discussions of planned extensions can be found in [1] and [2].

[2] The jre/classes directory is not technically part of the implementation of extensions, and it is not created by a default installation. Once created, however, the jre/classes directory defaults to the same permissions as the boot class path: all permissions [3 p87].

class of an application and use a malicious class loader to become the application's defining class loader, and then overcome signed JAR protections; the class could strategically insert itself between two classes and simply record data that is passed between them. Once a malicious class is inserted into the system and given system privileges, the attacks are endless.

The primary reason that these attacks are actually viable in the Java system is because reverse-engineering of Java class files is so easy. This is primarily due to Java's portability. The class file format includes human-readable strings revealing the names of class methods and variables [7]. The standard JDK distribution includes a tool[3] to disassemble the byte code of public and private methods and list the variables of any class file. In addition, the marketplace is full other tools to completely decompile any class file into its original source code. Only with the most sophisticated obfuscation tools can one protect a Java class file from reverse-engineering to the equivalent degree of difficulty as compiled C++ code, and even then this results in a modest penalty in runtime performance.

Even assuming an application is protected with radical obfuscation on its class files, Java classes are still easier to analyze than a corresponding C++ application. Java class files encompass only a single class object, are generally well named for human recognition, and are organized logically by function into packages. This makes finding and identifying interesting functionality very easy. However, even if the effort to fully reverse-engineer a Java class is not expended by an attacker, the key components of an application can be easily extracted as class files and reused in other applications. For that matter, the entire application itself can be easily copied and used without a valid license. The techniques used to make a monolithic C++ application unusable without a license do not carryover as simply to the Java environment. The ease with which intellectual property can be lost through reverse-engineering and illegal copying of Java classes is a practical consequence of the conveniences of the Java class file format.

Another deficiency is in the application of the security policy. While the Java literature discusses the fine-grained access control capabilities of Java [1][2][3], in practice this applies only to the specification of permissions, not to the application of those permissions. The mechanism that Java uses to define permissions is fine-grained: any permission that can be conceived can be created as an extension to the permission hierarchy. However, the application of the permissions in the Java 2 security architecture is limited to a directory or a JAR file [3]. This is insufficient, except for the most rudimentary systems. However, it reduces system complexity and eases security policy design at the expense of better access control. Since a single JAR file or a set of classes in a directory defines most applications, all classes in the application typically receive all the permissions granted to that JAR file or directory. This violates the principle of granting least-privilege. Furthermore, this type of single grant is (erroneously) assumed to be proper; the idea for this being propagated by the default policy that grants all permissions to every class or JAR file deposited in the jre/lib/ext directory. Some commercial systems are installed with a security policy that is even worse[4].

A further vulnerability concerning the security policy has to do with the way it is applied. The "java" command allows the security policy to be overridden using the "java.security.policy" option [4]. Using this parameter on the command-line allows any additions the user desires to be added to the security policy. Additionally, if a double equal (i.e. ==) is used with this command-line parameter, the system specified security policy is completely replaced by the file specified [3]. Although this functionality can be turned off using a property in the java.security file, the default JRE installation enables this insecure behavior. Obviously, this is convenient for developers, but it provides an enormous loophole in the enforcement of a system security policy for Java.

The final vulnerabilities discussed in this paper have to do with JAR files. Although "sealing" of JAR files is recommended [5], there is no mechanism in place to ensure that the "sealing" is not tampered. It is far too easy to modify a JAR file and remove the "sealed" attributes of entries, and there is no runtime mechanism to check that elements that were supposed to be sealed are in fact sealed. Signed JAR files can suffer from this same problem. When signed JAR files are used as a single mechanism, there is nothing preventing an attacker from removing the signature file in the JAR and modifying the classes in the JAR file. The class loader will still load and execute the classes. Only when used in conjunction with a policy that assigns specific permissions to the classes from a signed JAR file, can the signature be useful. The trust inherited from the signed JAR file goes unused unless the policy specifically references the certificates that should be associated with classes from the JAR. However, if the class loader cannot be trusted (e.g., the class loader associates a particular certificate to all CodeBases regardless of the location from where they were loaded), the use of the policy accomplishes nothing.

From this discussion, it is evident that there are serious vulnerabilities in Java's security. A knowledgeable attacker can easily leverage these vulnerabilities using a valid account on the system. Even

[3] See Sun's JDK documentation for the *javap* command

[4] See the policy implemented by Jrun 3.0 in the *jrun.policy* and *ejipt.policy* files, which provide grants that gives all permissions to every class, regardless of its CodeBase.

some of Java's strongest security mechanisms (e.g., signed JAR files and policy enforcement) can be circumvented without such an attack becoming evident to the system or the user.

4 Security Requirements

This section introduces a set of assumptions, a set of threats, and a set of security objectives that influence the design of security extensions for the application server. The assumptions are based upon the nature of a Java application server, and the threats are based upon the security vulnerabilities discussed in the previous section. The objectives are an informal statement of the security requirements guiding the design of the security extensions.

4.1 Assumptions

The environment for a typical web server is the basis for the assumptions used to develop the security extensions in this paper. The assumptions are formulated from the perspective of an independent software vendor (ISV) providing a Java application server as a product offering. These assumptions, however, have basic relevance beyond that particular viewpoint.

The application server is a typical networked server, which is accessible from a public network. The operating system provides common authentication (e.g., username-password authentication) and access control (e.g., read-write-execute access lists) protections, and the server is housed in a facility with physical protections (e.g., locked doors). While the network and server host are considered to have standard protections (e.g., firewalls, updated patches, etc.), we specifically assume that these will not prevent an outside attacker (i.e., hacker) from accessing the host server. Additionally, we focus internally on the server, and specifically ignore external communications (e.g., encrypted sockets) between the server and other entities (e.g., users, databases, etc.).

The Java application server is characterized by a static set of Java classes and a stable configuration. The set of class files (including JSP files) in the Java server does not change on a frequent basis. Changes are limited to authorized administrator upgrades to the system. The delivered content of the server changes dynamically due to data changes in the server's database. Also, the configuration of the server changes very infrequently; the server is configured into a stable state and left alone to operate in that state. In addition, the configuration from one server to the next is fairly similar, allowing scripted installation tools to be used for setup and upgrades. This assumption is important, as some of the security extensions discussed here involve the use of digital signatures that complicate the frequent modification of some system components.

4.2 Threats

The threats to the system are addressed from the perspective of vulnerabilities in the Java security system due to configuration, backwards-compatibility, and implementation issues. Each threat is linked to one or more types of attackers that could potentially leverage the vulnerability. The types of attackers considered here include external hackers, system administrators, normal users, and system developers. The threats to the system include intellectual property loss, data disclosure and data corruption, tampering, and unaccountable actions.

A primary threat to an ISV is the loss of intellectual property. This loss results in actual or potential loss of income due to lost sales and/or loss of competitive advantage. Loss of intellectual property can be broken down into two subthreats: the threat of reverse-engineering and the threat of illegal copying. System administrators may have some financial incentive to generate illegal copies of software by installing copies on more servers than the number of purchased licenses. Reverse-engineering may be performed by any of the prospective threat agents, either in response to curiosity, a desire to overcome licensing restrictions or security controls, or for financial incentive from the ISV's competitor (e.g., the competitor may have purchased the software, thereby becoming the system administrator).

A primary threat to the ISV's customer is data disclosure and corruption. From an internal application perspective, these threats come from Trojan horse code, mal-ware, and software defects. These threats include any piece of additional software inserted into the system, or any valid component of the system modified in such a way as to passively collect and disclose, or destroy, private system data. For purposes of this paper, mal-ware is defined as this type of malicious code deliberately injected into the system by one of the system's developers; Trojan horse code is deliberately injected into the system by someone other than a system developer. Defects are similar to mal-ware, however they are inserted accidentally by a developer. This paper does not specifically address the threat of defects other than to separate them from mal-ware.

A common threat to both the ISV and the customer is tampering, especially tampering with the security mechanisms. For purposes of this paper, tampering is defined as the removal of a class or the modification of configuration data. This is specifically to differentiate this type of attack from Trojan horse code and mal-ware. The ISV's concern stems from the removal of security protections as a means to enable copying or as a preliminary step to reverse-engineering. From the customer's perspective this may be a preliminary step to

another attack on the system. The threat of tampering includes Java's configuration files, the security policy, and other relevant files, such as the keystore.

Other threats to the ISV's customer include unaccountable actions. Unaccountable actions are addressed by an audit facility. Although audit is typically a part of an application that is security-aware, certain aspects of the Java security environment provide additional incentives to integrate the audit capability with other Java security features. For this reason, integration of an audit facility into the Java security components will be briefly covered in this paper.

4.3 Summary of Security Objectives

The primary objective of the security architecture described in this paper is to improve the level of trust in the operational integrity of a java application server. This trust must be based upon automated mechanisms that guarantee the integrity of the components that make up the system and the application. Trust is based upon the level of assurance that the Java system and the application are free from malicious code, and that the integrity protection and tamper-detection mechanisms are operating properly. The system should be designed in such way as to safely stop operation as soon as a breech in integrity is detected. This objective is achieved through the following informal requirements:

- Restrict actions that may be performed by any system component to a known set of approved actions, defined by:
  - application of security policy down to individual classes (fine-grained security policy application);
  - security policy defines all sensitive operations (security policy completeness); and automated enforcement of the security policy.
- Ensure the system and application class files are genuine, which means:
  - strong integrity checks are applied to all classes that make up either the system or the application (strong class file integrity);
- Deter malicious code insertion, which includes the following actions:
  - restrict package membership to known classes;
  - enforce known linkages and method-call-patterns between packages;
  - perform minimal obfuscation on all application classes; and
  - disclosure protections are applied to sensitive application classes.
- Deter tampering of supporting system and application files through:
  - application of strong integrity checks to all system configuration files and supporting files; and
  - automatic consistency checks of system installation, including the contents of sensitive directories.
- Deter illegal copying by:
  - link the ability to use application class files to an unforgeable hardware token.

5 Architecture

The system is architected around the security objectives outlined in the previous section. The architecture consists of extensions to the Java 2 security system, including the policy manager, the class loader, and the security manager. Each of these elements participates in meeting different aspects of the security objectives. A graphical depiction of the architecture is shown below.

5.1 Concept of Operation

The security extensions create an inter-locked dependency on each other, with each component satisfying parts of the security objectives. The system is designed such that if a failure in the system integrity is detected by any component, that component shuts down the system. This is shown by a series of dependencies among components in Figure 1. Since the primary motivation in having the application is to actually run it, use of the application's class files is a necessary precondition. Since the class files are encrypted, access to the application necessarily requires the use of the custom class loader.

The class loader will not operate without the self-installing extended security manager. The extended security manager performs system integrity checks during startup and during normal permission checks. These checks include integrity checks on the system environment, including validating manifests of sensitive directories, validating the rt.jar, and checking that the extended policy manager is operating.

The extended policy manager is loaded by the JVM during startup. It checks that the system's configuration files have not been modified by verifying a digital signature across each file. The public key certificate used to verify the signatures is stored on the hardware token. Since the extended policy manager is normally started during JVM boot, it also checks certain elements of the configuration necessary for proper operation of the system. One key element of this is the proper configuration of the Java Cryptography provider.

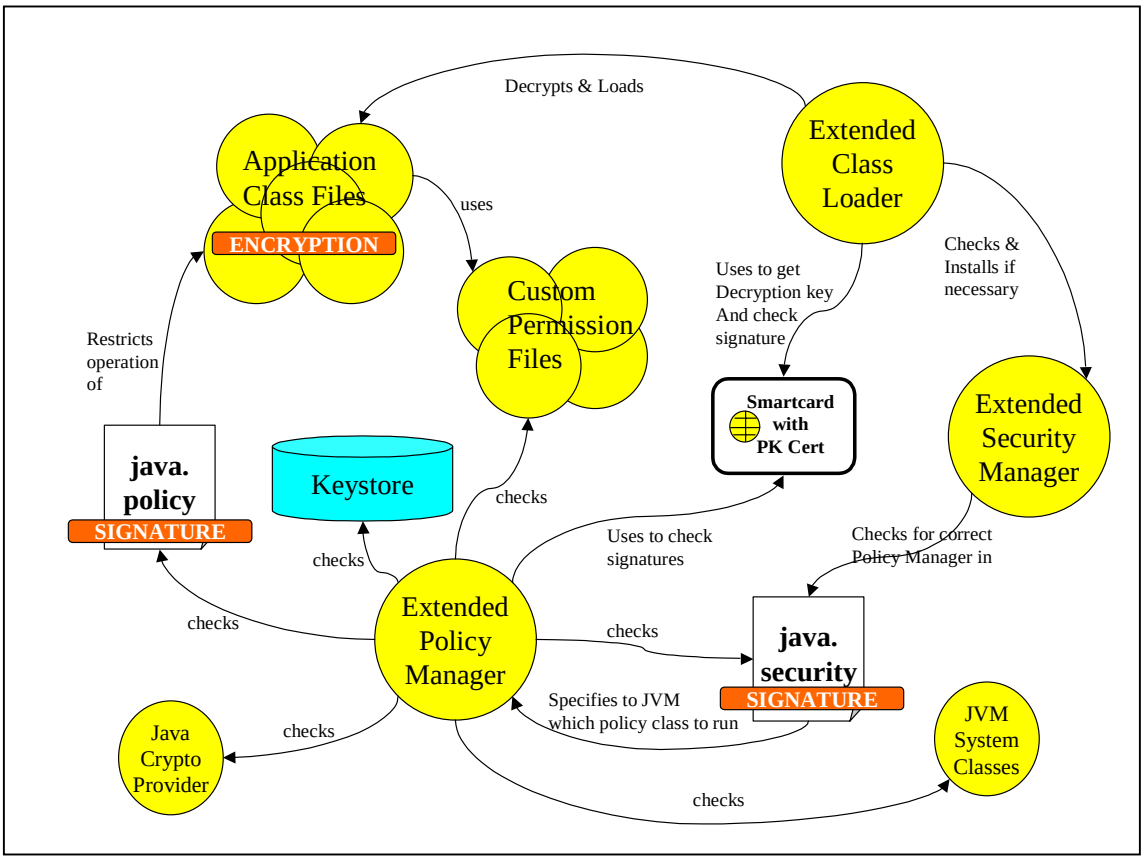

Figure 1: Dependencies between Java 2 Security Extensions

Before the class loader loads any classes, it loads the extended security manager, which checks the system and validates that the extended policy manager is loaded. The extended policy manager should be loaded as part of the JVM start-up process. When the extended policy manager is loaded, it performs additional checks on the system.

If any of the integrity checks fails, each extended component notifies the other components of the failure, and attempts to shut down operation of the system. The extended policy manager denies access to all services by returning the null permission set in response to any query. This effectively prevents any sensitive operation from being performed for newly loaded classes. The extended security manager throws a security exception for every check function. The extended class loader purges the class cache and the ProtectionDomain store and refuses to load any classes by throwing a security exception.

If all the integrity checks successfully pass, the extended policy manager and the extended class loader enforce a fine-grained security policy that assigns permissions down to the class level. The policy file is set up to grant the minimal set of permissions to each class file that is necessary for the class to perform its function. All application class files enforce a permission check in their constructors to verify the calling class has permission to utilize the specific package. This permission check has the effect of installing the security manager (if it is not present), since the extended security manager is self-installing.

Trust can be assigned to the system because, whenever the system is operating, it must be operating correctly. This trust is assigned based upon the cryptographic strength of the encryption used to protect the class files, the assurance that the cryptographic keys remain secret (key storage on the HW token), and the cryptographic strength in the digital signatures across the configuration data. If these elements are trusted, then the system can be trusted because:

- only the extended class loader can access the cryptographic keys and load the encrypted classes;
- only approved classes have authority to perform any sensitive actions because the extended policy manager is started at system boot time, and the extended security manager is installed by the class loader;
- only approved classes can run in the system because the extended class loader loads all non-JVM classes, and the extended policy manager and extended security manager validate digitally signed manifests of all system directories, JAR files, and configuration files; and
- the system cannot be copied to an unapproved

system, since it requires a hardware token to operate.

5.2 Security Policy

The extended policy manager and the extended class loader are responsible for enforcing the security policy. The extended policy manager reads the *java.security* file and *java.policy* file to interpret the security policy to be made active in the system. The *java.policy* file is extended to allow the CodeBase to specify a directory, a JAR file, a class file, or a class file in a JAR file. The class loader was required to be involved in this extension, since the class loader provides the CodeBase of the class being loaded to the policy manager when retrieving the permissions for the ProtectionDomain. The default behavior of the class loader is to strip off the class name and provides only the directory or the JAR file name. The class loader is extended to provide the full pathname, and if the file is a JAR, to provide the package name along with the class name in the CodeBase. This extension satisfies the requirement to apply a fine-grained security policy.

Completeness of the security policy is performed by manual inspection of the policy file, and functional testing of the application server. Review of the text policy file is performed manually, ensuring that each class is individually listed in the policy file and has only the minimal permissions required to run. For example, the database driver class should have permissions to open a socket to the database server, but other classes should not have such a permission. Validation of the policy is easily performed during normal feature validation. Any class that violates this policy will be found while functionally testing the system, since the AccessController will throw a security exception when the policy is violated. Because the policy file is applied down to individual class files, it is easier to see the exact effect of the policy, and which classes are utilizing which permissions. This makes odd or incorrect permission grants stand out in the policy file, and easy to identify by manual inspection. This review of the policy is primarily to defend against mal-ware.

The extended security manager performs the automated security policy checks. The extended security manager is a self-loading singleton class [12]. Access to the extended security manager is provided through a getInstance method. The extended class loader calls this method when it is first instantiated, thus causing security checks to be performed before any sensitive classes are loaded.

5.3 Genuine System and Application Classes

The extended class loader provides integrity checks on all application classes and the extended security manager checks the integrity of all system directories and JAR files. The application classes are protected in signed JAR files, and sensitive classes within those JAR files are protected by a special class file format. The special class file format encrypts the class file content. The extended class loader provides the ability to decrypt these class files using a key stored on a hardware token; this is done during the loadClass method [9].

The test implementation uses a smartcard for the hardware token, and a smartcard reader with secured PIN entry on the reader itself (trusted path). During system startup, when the class loader is first started, after the system has validated the system integrity, the system operator is prompted for the PIN. This PIN is entered directly on the smartcard reader (hardware-only path from PIN pad to smartcard), allowing the class loader to open a secured session to the smartcard.

5.4 Malicious Code Protections

All three extended components participate in the protections against malicious code. The extended class loader checks the integrity of all JAR files. The policy assigns special permissions to all classes that restrict package access and the policy manager provides methods to check package assignments based upon validated system properties. The presence of the security manager ensures that the AccessController is used to check permissions.

The extended class loader is protected from CLASSPATH manipulation by divorcing the search path used by the class loader from a user modifiable system parameter. The class loader uses a digitally signed properties file to list the installation directories of the application. This list can be verified as unaltered using the digital signature. In addition, the extended class loader does not directly follow the recommended algorithm for class loading. Specifically, the extended class loader searches it's own cache, and then tries to load the class from its URL search path before passing off the load request to the delegation parent. This has the effect of preventing code insertion attacks on the application using the system class loader or the Xbootclasspath (e.g., the jre/classes directory and the jre/lib/extensions directory).

The checks performed by the extended class loader provide protection for application classes. The class loader checks the integrity of all JAR files using the standard signed JAR methods. To ensure that the signatures are not completely removed from the JAR files, the security policy must tie its permissions assigned to classes from signed JARs to the certificate used to validate those JAR files. In addition, as discussed above, the extended class loader provides disclosure protection for sensitive classes through the use of an encrypted class file format.

Assignment of a class to a package is always validated by the extended class loader before the class is loaded. This involves a check with the extended policy manager that the class being loaded has a permission associating it with that package. This is different from the normal permissions checks in Java 2, which check the entire call stack. This check merely checks a single permission assigned to a class based upon the class' CodeBase. This check meets the requirement to restrict package membership.

The extended class loader uses the accessClassInPackage technique to enforce known linkages between packages [6]. This requires some work to translate the known package interactions into permission grants in the policy file. For each class that accesses a protected application package, and the class is not a member of that protected package, the policy file must grant that class access to the protected package. This involves a special method in the extended security manager so that the entire call stack is not checked for that permission, only the referring class. This special method in the extended security manager is only available to the extended class loader.

As a final protection, as a deterrent to reverse-engineering attacks, all class files are obfuscated. All private members and local variables are obfuscated; this may actually provide a performance improvement, as many longer strings are shortened. More sensitive classes may be obfuscated more radically, however, this can have impacts on the policy file and other classes if the class names, package names, and/or public members of a class are modified.

5.5 Tampering Protections

The extended policy manager and extended security manager perform integrity checks on all configuration files and system directories and JAR files during start-up and intermittently throughout the operation of the system. The extended policy manager performs checks on the *java.security* and *java.policy* files, as well as over the installation of the cryptography providers, the keystore, and an initial check on the rt.jar and the Xbootclasspath. These checks involve validating the actual files against a digital signature to ensure their contents have not changed. The extended security manager performs additional checks on the rt.jar, and validates a set of manifests for the system directories, including directory contents, file sizes, and checksums on certain files. These manifests are digitally signed to protect them from tampering. As an additional step, the extended security manager checks for the proper configuration of the smartcard subsystem, including native libraries, to ensure that the smartcard system is not altered.

5.6 Audit Facility

A specific premise of the security policy should be that all sensitive operations are checked with a Java permission. This permission check occurs through the security manager, which could provide additional services. In [2], the idea of adding audit to the functionality of the security manager is introduced. This provides a convenient method to localize audit collection in one place, yet tailor the audit messages for each event by providing a method in the permissions object to format an audit record. This has the benefit of allowing a single review of the security policy to check for completion of both policy and audit coverage.

6 Current Status and Future Work

The current project has completed the first development phase of the security extensions discussed in section 5. The system includes an extended class loader and policy manager that implement a fine-grained security policy down to class files and classes in JAR files. The class loader implements a protected class file format that includes encryption of the class file and association of certificates to specially protected classes. The extended policy manager and security manager perform checks on the system integrity of configuration files and other system components using digital signatures. A demonstration implementation of key storage was built on a smartcard, but has not been integrated with the other components of the system. Testing of the current system has focused on the security policy implementation and demonstrated successful association of permissions based upon the fine-grained policy. The integrity protections, while effective, have the drawback of being overly sensitive to whitespace modifications on the configuration files.

Future work includes complete integration of the hardware token, full integrity checking of the environment, and additional testing of the system. Integration of the hardware token requires further analysis to design automated validation of a trusted path through the class loader to the smartcard. Although most integrity checks were implemented, manifest files of sensitive directories have not been integrated into the system. A change to the current configuration file protections involves the use of XML security protections [13][14] to solve the sensitivity to whitespace. Additionally, a mechanism for runtime cross-validation of the active security components (i.e., the class loader, the policy manager, and the security manger) needs to be designed into the system to allow the stronger validation that the security components are actively running in the system. As a final task, penetration testing of the completed system, based upon the vulnerabilities described in

section 3, must be completed. Although this project is formally complete, continuation of this work for integration into a large-scale application server, and addition of the future work items discussed here, is currently under consideration.

7 Appendix: Java 2 Security Background

Java 2 security was designed to provide the basic security mechanisms required for a client environment, a server environment, and an operating system [1]. These mechanisms include security policy definition, code source authenticity, and authorization controls.

7.1 Java Security Policy Definition

The Java security policy is defined by a policy manager class and represented within an active JVM in combination with the class loader. Other objects are involved with the representation of the security policy, including the CodeSource class, the ProtectionDomain class, and the Permissions class hierarchy.

The Java security policy is defined in a text file as a series of grant statements. Each grant statement assigns one or more permissions to Java classes based upon a CodeSource. The CodeSource defines a code location and a set of certificates. The code location is a URL that specifies a local or remote directory or JAR file. The certificates are represented in the policy file by common names that identify the public keys used to sign the class files. The common names are used to retrieve the actual certificates from the local keystore [1]. A local administrator defines the Java security policy in two files: the *java.security* file and the *java.policy* file [4]. Together, these files define the objects and permission sets that comprise the JVM's security policy.

The Java policy manager class, along with the class loader, is responsible for providing the runtime representation of the security policy. The policy manager class reads the security policy from the configuration files and assigns sets of permissions to unique CodeSources. The class loader is responsible for binding a set of permissions to each loaded class [1]. The class loader creates a CodeSource for each class it loads using the location from where the class was loaded, and the set of certificates that were used to sign the class. The class loader uses this CodeSource to retrieve the set of permissions from the policy manager, and then creates a ProtectionDomain from this set of permissions and the CodeSource. This ProtectionDomain is cached by the class loader, which associates those same permissions with every class from the same CodeSource. The AccessController uses the permissions in the ProtectionDomain when verifying whether a particular class has a particular permission.

7.2 CodeSource Authenticity

Code source authenticity is provided through the use of JAR files and special permissions. This functionality is implemented in the class loader as sealed JAR files, signed JAR files, and protected packages.

Java 2 provides a special mechanism, called "sealing", within JAR files to ensure that only one JAR file supplies classes for a specific package. JAR files are "sealed" by adding a text attribute to the manifest file. This attribute can be applied to all packages in the JAR file, or only to specific packages. When a package is "sealed", all classes that claim membership to that package must come from a single JAR file [5].

The class loader enforces special restrictions on classes and packages when these come from signed JAR files. If a class comes from a JAR file that is signed, and the digital signature across the class file is verified, then the class file is given a new security attribute in its CodeBase. This security attribute is the certificate of the signer of the JAR file. Since the certificate is added to the CodeSource, and the CodeSource is used to assign permissions to classes through the ProtectionDomain, the presence of a certificate can be used to assign special permissions to a class or set of classes. Additionally, when the first class for a particular package is loaded, the class loader associates any certificates assigned to that class to that package as well. Then, when additional classes for that package are loaded, they must also have the same certificates associated with them. This enforces equal integrity protections across all classes within a package, and prevents some types of malicious code insertion attacks.

The class loader provides special protection to packages defined in the *java.security* file under the defineClassInPackage and accessClassInPackage properties. These properties are a list of package names, each of which require a special permission in order for the package to be used. For all packages listed in the accessClassInPackage property, class loaders must have a special permission to access classes in that package. In the default installation of the *java.security* file, the "sun" package is listed in the property list for accessClassInPackage. This prevents classes loaded by the system class loader from using classes considered private by Sun and residing in a package beginning with the name "sun." The defineClassInPackage property is intended to work in a similar fashion for the permission to define classes, however the standard class loaders do not implement checks for this property. This mechanism of package protection was deemed insufficient to protect the "java.*" packages, presumably because Sun Microsystems wanted to ensure the base packages of Java could not be supplanted by merely changing a text-based configuration file. Additions to the "java.*" packages are

protected using explicit code within the base class loader [6].

7.3 Authorization Controls

The security manager and AccessController enforce the authorization controls during runtime using permission checks. A permission check uses the CodeSource of the class being checked to reference the set of permissions in the ProtectionDomain, and verifies that the class holds a permission for the action being performed. The permission set in the ProtectionDomain was created from the security policy in force at the time the first class matching that CodeSource was loaded. These authorization checks are performed only if a security manager is currently active in the JVM. This condition is a result of a backwards-compatibility requirement that new versions of the JVM preserve the behavior exhibited by legacy code when the security manager is null. It is for this reason that the JVM does not directly call the AccessController to perform permissions checks, but instead calls the active security manager, or performs no checks if the security manager is null [1].

8 References

[1] L. Gong, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, Addison-Wesley, 1999.

[2] L. Gong, *Java 2 Platform Security Architecture*, Oct 1998, Copyright 1997-1999, Sun Microsystems, http://java.sun.com/j2se/1.3/docs/guide/security/spec/security-specTOC.fm.html

[3] M. Pistoia, et. al, *Java 2 Network Security, 2nd Edition*, IBM, 1999.

[4] G. McCluskey, "Adding Security Features to Applications". *JDC Tech Tips*, July 14, 1999, Copyright 1995-2000, Sun Microsystems, http://developer.java.sun.com/developer/JDCTechTips/.

[5] Sun Microsystems, "Controlling Package Access with Sealed JAR Files", *JDC Tech Tips*, Jan 30, 2001, Copyright 2001, Sun Microsystems, http://developer.java.sun.com/developer/JDCTechTips/.

[6] Sun Microsystems, "Controlling Package Access with Security Permissions". *JDC Tech Tips*, Jan 30, 2001, Copyright 2001, Sun Microsystems, http://developer.java.sun.com/developer/JDCTechTips/.

[7] Sun Microsystems, *The Java Virtual Machine Specification*, http://java.sun.com/docs/books/vmspec.

[8] Sun Microsystems, "Relating Class Loaders to the CLASSPATH", *JDC Tech Tips*, Oct 31, 2000, Copyright 2000, Sun Microsystems. http://developer.java.sun.com/developer/JDCTechTips/.

[9] MageLang Institute, "Fundamentals of Java Security", Copyright, 1998, MageLang Institute, http://developer.java.sun.com/developer/onlineTraining/Security/

[10] R. Srinivas. "Java Security Evolution and Concepts, Part2", *Java World*, July 2000, Copyright, 2000 Itworld.com Inc, http://www.javaworld.com/javaworld/jw-07-2000/jw-0728-security_p.html

[11] Sun Microsystems, "Java Security", http://java.sun.com/docs/white/index.html

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

[13] W3C, *XML Signature Syntax and Processing*, Draft Candidate 2, http://www.w3.org/TR/xmldsig-core/

[14] W3C, *XML Encryption Syntax and Processing*, Work-in-Progress, http://www.w3.org/Encryption/2001/