RC24232 (W0704-030) April 6, 2007
Computer Science

IBM Research Report

Classloading and Build Issues for Secure and Reliable Java Systems

Sam Weber, Paul Karger
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Classloading and Build Issues for Secure and Reliable Java Systems

Sam Weber
T.J. Watson Research Center
samweber@watson.ibm.com

Paul Karger
T.J. Watson Research Center
karger@watson.ibm.com

Abstract

Current research on software security has tended to focus on directed techniques such as detecting known flaws, finding coding anomalies that might indicate problems and protecting against known attacks or vulnerabilities. In contrast, process-based approaches such as the Common Criteria have the potential to discover or prevent unknown flaws. Unfortunately, such process-based approaches offer no benefit to pre-existing software, or software developed under a different methodology.

In this paper we investigate the security issues raised by dynamic classloading, as related to large, high-assurance Java products. We describe a previously unknown vulnerability in Java which allows remote attacks. We also derive a set of build requirements, based upon the Common Criteria, that can be applied to non-Common Criteria Java products. We conduct an experiment on two large, mature, widely used Java systems, one commercial and one open-source, evaluating them against our requirements. This resulted in the discovery of previously unknown security and reliability issues in both products, related to dynamic classloading.

1 Introduction

Although security has recently received larger attention from the general community, due to the prevalence of internet-based attacks, much of the focus has been on "flaw-centric" techniques, such as identification of known types of flaws in code, as in [7, 15, 5, 9, 12], and in finding anomalies in code which might signify problems, as in [6].

In contrast, the Common Criteria [1, 2] and Correctness By Construction [4] are process-based methodologies. They do not define specific features applications must contain, nor specific flaws that must be prevented. Instead, they both emphasize development methodology and procedure, attempting to ensure that the design and development processes have been done in a rigorous fashion.

Whereas flaw-based approaches can only protect against known flaw classes, process-based techniques have the potential to avoid even undiscovered types of vulnerabilities. Unfortunately, process-based techniques are often thought to be impractical. In particular, they are considered to offer no utility to software that was not produced by that process.

Our work is based upon the observation that software developed by a high-level Common Criteria evaluation will have certain characteristics, which can be measured in software that was *not* developed according to that methodology. Deviances from these characteristics in analyzed software can reveal flaws, including ones that flaw-based techniques would not reveal.

In particular, we concentrate our attention upon classloading issues in large Java systems. Java loads all of its classes dynamically, and dynamic linking has long been known to present security issues.

Additionally, the Common Criteria requirements implicitly assume static linking, and therefore it is unclear what requirements are appropriate for Java software.

Our work began when the authors were asked to assess the security of a certain large Java-based commercial product, in addition to the assessment it was already undergoing as part of a low-level Common Criteria evaluation. This product was *not* originally designed in accordance with the Common Criteria. Rather, a Common Criteria evaluation was being performed on an already existing large Java-based product. Furthermore, the product in question actually included several other already existing large products as subsystems, and was under continuing development by different groups around the world.

We conducted the following experiment: A set of build process requirements for Java software was developed, designed to reflect the results that would be expected from following a high-level Common Criteria evaluation process. Dynamic and static analysis techniques were used to inspect the commercial application against these requirements. In order to ensure the general validity of our results, an open-source application was similarly inspected. The course of our investigations led us to investigate Java's Remote Method Invocation (RMI) subsystem. Finally, the security and reliability impact of our discoveries was evaluated.

Significant and previously unknown security and reliability issues were discovered in both applications, including the cause of long-outstanding defects in the commercial application. Also, we found that the interaction of dynamic loading and the RMI subsystem resulted in a flaw that allows remote parties to execute a two-part attack on Java server applications, causing JVM crashes or other failures.

This paper is *not* primarily about the static analysis tools we developed. Its focus is on the process of developing large-scale, Java-based applications and on RMI.

Section 2 of this paper provides background on the Common Criteria and Java classloading. Section 3 describes the build criteria derived from the Common Criteria. Section 4 describes the analyses of the two applications, followed by Section 5 which covers security issues found in Java RMI. Section 6 describes the security and reliability impact of our findings. Finally, we draw conclusions and suggest future work. First, though, we provide details about our target applications.

1.1 Target Applications

In this work we examined a large commercial application, written in Java, that was undergoing a Common Criteria evaluation at level EAL 4+, which is a low-level evaluation, but the highest level widely used by commercial products. The developers allowed us access to their Common Criteria documentation and source code, under the condition that we not identify the product in any resulting publications. For this reason, we will refer to this software system as "$\mathcal{C}$". One source for their support of this work was that $\mathcal{C}$ incorporated code from third-party sources, both commercial and open-source, over whose development processes they had little control, and thus they were concerned about the resultant risks.

In order to ascertain how universal our results were, after completing our analysis of system $\mathcal{C}$ we then inspected an open-source application with near-equivalent functionality to $\mathcal{C}$. Since revealing the identity of this open-source software would serve to reveal the identity of $\mathcal{C}$, we also anonymize this second application, and refer to it as "$\mathcal{O}$". $\mathcal{O}$ is a software suite comprising four open-source projects. Our analysis of $\mathcal{O}$ suffered somewhat from the lack of Common Criteria documentation, but we were still able to achieve comparable results.

Both $\mathcal{C}$ and $\mathcal{O}$ are mature applications: $\mathcal{C}$ was initially released more than eight years ago, $\mathcal{O}$ more than seven and both have been continuously updated since. Details about their size and number of classes are in Table 1. The $\mathcal{C}$ application provides its own JVM, while $\mathcal{O}$ does not. Although the JVM used by $\mathcal{C}$ is not identical to any publically available release, in order to provide a fair comparison we used with $\mathcal{O}$

	$\mathcal{C}$ JVM	$\mathcal{O}$ JVM
jar files	40	50
total size of jars	49M	50M
total class files	17,069	17,310
distinct classnames	16,920	17,129

(a) JVM statistics

	$\mathcal{C}$	$\mathcal{O}$
total jar files	804	224
total size of jars	315M	70M
total class files	151,227	62,819
distinct classnames w/o JVM	83,930	27,323
distinct classnames with JVM	84,622	42,216

(b) Application statistics

Table 1: Size of analyzed JVMs and applications

a JVM with the same vendor and release number as that provided with $\mathcal{C}$. The characteristics of these two JVMs are shown in Table 1a. The main difference between them was that $\mathcal{O}$'s JVM included several demonstration programs not included in $\mathcal{C}$'s. As can be seen, both applications are large, and their sizes limit the amount of manual inspection that can be done.

Both $\mathcal{C}$ and $\mathcal{O}$ are Java-based middleware that have the ability to process requests received from remote machines on behalf of other, client, applications. Additionally, plugins that are specific to the client applications might be installed into $\mathcal{C}$ and $\mathcal{O}$ in order to add functionality to the middleware layer. Both $\mathcal{C}$ and $\mathcal{O}$ must ensure that client plugins cannot interfere with each other, even if they contain identically-named classes, which necessitates the middleware's use of separate classloaders.

2 Background

2.1 Common Criteria

The Common Criteria (CC) is an international standard for evaluating the security of both hardware and software products. It emphasizes development methodology, attempting to ensure that the design and development of the evaluated product has been done in a rigorous fashion, and the security features have been properly implemented. The result of a successful CC evaluation is one of seven Evaluation Assurance Level (EAL) ratings, from EAL 1 to EAL 7.

In this work, we are interested in examining the build process of modern Java software in a manner consistent with, and motivated by, the Common Criteria process.

Vulnerabilities due to dynamic linking have long been known in UNIX. Garfinkle and Spafford [8, pp. 151-153] describe some simple examples of such vulnerabilities. Multics, which first intoduced dynamic linking in the late 1960s, avoided these problems by pre-linking all privileged programs. However, the UNIX setuid facility did not provide such isolation and that resulted in problems.

Many CC requirements either assume static linking or fail to account for complications arising from dynamic linking. Java not only supports dynamic linking, but dynamically loads and links every class and therefore we have had to infer from the statements regarding statically linked programs what equivalent requirements on dynamically linked programs would be. Furthermore, Java, like UNIX setuid and unlike Multics, does not avoid dynamic linking for privileged programs.

Although space restrictions do not allow us to describe the Common Criteria's build and configuration goals and requirements in any detail, the aspects of a medium-level evaluation (EAL 5 or higher) which most affected this work were:

- All parts of the product must be strictly tracked and managed, including all source code, generated binaries, development tools and build scripts.
- Which binaries will be executed must be precisely determined.
- The build process must be completely automated.

- The programming language must be completely defined, including the semantics of every statement.
- Every call from one "subprogram" in one "portion" of the source code to another "portion" must be shown to be consistent.
- The system must be modularized, and there must be no unnecessary interactions and interdependencies between modules.

2.2 Java Classloading

One of the key foundations of Java security, and also a major build process challenge, is Java's classloading mechanism. The main purpose of this mechanism is to partition the class namespace into sections that cannot interfere with each other. The initial rationale behind this was to support applets, which require some classes to be downloaded from the internet with the assurance that these classes are not able to override or subvert local classes and thereby act maliciously. Classloading is being used increasingly to enable independently developed software to co-exist, despite the fact that they might use different versions of the same library. Both $\mathcal{C}$ and $\mathcal{O}$ use classloading to isolate each of their plugins, as these plugins might otherwise conflict.

In Java, each class is dynamically loaded at runtime, when needed. Every class is loaded by a classloader, and a running Java system contains multiple classloaders. Each classloader maintains its own namespace of classes, and mechanisms are in place to ensure that less-trusted versions of classes don't override more-trusted ones, and that classes with the same name can be loaded by different classloaders and kept distinct. Each Java classloader maintains a separate set of classes, obtained from different sources, and possibly delegates to another classloader.

If the classloader used by a certain part of an application obtains its classes from the local filesystem, then the term "classpath" is colloquially used to refer to the set of jar and class files that are available.

Before a Java object can be created, its class must be successfully loaded, linked and initialized. Although "loading" is often colloquially used to refer to this entire process, strictly speaking loading only means obtaining the sequence of bytes that defines a class. The linking process verifies the structure of the class, creates certain data structures and resolves references to other classes. It is only possible to do type-checks on classes after they have been linked. The final step is initialization, during which possibly programmer-defined code is executed to initialize the state of the class. This is the first step in this process which involves the execution of untrusted code.

3 Build Requirements

Based on the aims above, we propose the following build requirements, or "sanity conditions" for a Java-based application. These requirements provide the same guarantees for a Java application that the Common Criteria documents imply for a statically-linked program.

1. Each classloader used by the application shall be clearly identified.
2. The delegation relationships between classloaders shall be clearly identified.
3. For each classloader,
   (a) each class available from that classloader (including via delegation to another classloader) shall be identified. The namespace defined by each classloader shall be documented.
   (b) each available class shall be uniquely and unambiguously obtained from exactly one class or jar file.
   (c) any dependencies between classes shall be satisfied by the classloader
   (d) each available class shall be loadable and initializable at any time.

As discussed above, the CC documents clearly indicate that all calls between modules or "portions" of the code shall be consistent and well-defined, and that all the code in the system be explicitly defined. In Java all references between classes are made symbolically, and only linked when the classes are loaded. In a Java program it is not possible to be sure that the target of a symbolic reference in one class even exists, much less is consistent, unless one knows what classloader loaded the class and what classes are available to that classloader. Therefore, our requirements 1, 2 and 3a force the existing namespaces and what they contain to be explicitly documented. Requirement 3c enforces consistency between classes.

The requirement 2b states that each class in a classloader's namespace should be defined exactly once. For instance, if a classloader searches for its classes in the files "a.jar" and "b.jar", there should be no classname that is found in both jar files. If there were such ambiguities, then it would not be known which version of the class will actually be executed, or whether one class's reference to another was consistent without knowledge of the runtime behaviour of the classloader. In some existing Java applications the resolution of which copy of a class will be loaded is dependent upon under which operating system the application is running. Such a situation would greatly increase the complexity of analysis, and therefore we disallow it.

Java does not define the order in which class initialization will occur, other than that every class will be initialized before it is used. There are some circumstances in which one class can affect another's initialization without using it, by altering system state elsewhere. Since Java does not define the order in which such classes will be initialized, this would be a violation of the CC requirement that the programming language defines precisely the semantics of each statement. For this reason, requirement 3d disallows this situation.

4 Analyses and Results

In this section we describe the analyses (including results) that we performed upon both $\mathcal{C}$ and $\mathcal{O}$.

It should be noted that all the analyses operated on binary object code (JVM class files), not Java source code. This was for two reasons. First, many JVM class files are not generated from Java source code, but are rather the output of various other tools such as "rmic". Java source for these generated classes does not exist. Second, using the available Java source code would not tell us whether the build process correctly produced the object code.

Our focus upon class files has several implications. First, there is not a one-to-one relationship between Java source files and JVM class files. For instance, a Java class that contains two inner classes will be compiled into three class files. Since we are analyzing JVM classes, we will consider these to be three classes, not one. Second, both interfaces and classes are compiled into class files, and the JVM makes little distinction between classes and interfaces. For example, both class and interface objects are children of `java.lang.Class`. For the sake of brevity, unless explicitly stated otherwise, we will make no distinction between class files that contain class objects and those that contain interfaces.

Our sanity conditions explicitly depend on the structure of our application's classloader hierarchy. Both of our applications documented their hierarchy, and in both cases, this featured a main classloader which loaded the main application functionality. All our analyses, therefore, concentrated upon this main classloader.

Three analyses were performed:

Class Loading Test:

The code of both applications was instrumented, and their run-time behaviour was monitored to discover which classes were made available by the main classloader and which classes were successfully loaded. This test also investigated whether there were any dependencies on load order.

Class Duplicate Examination:

This static analysis investigated the main classloader's classpath, and determined whether each available class was uniquely defined.

Static Dependency Analysis:

This static analysis checked for inconsistencies between classes, and for classes whose static dependencies were not satisfied from the same classloader.

Although these tests do not use novel techniques, they effectively allowed us to test for failures of our build requirements. We will now consider each of these analyses in depth.

4.1 Class Loading Test

Recall that our build requirements included that every class available from a classloader has its dependencies on other classes satisfied and is loadable and initializable in any order. In order to test this, the following process was followed:

1. The names of all class files found in all jars in each application, including from the JVM, were extracted to obtain a list of all possible classnames that could be loaded. This resulted in a list of 84,622 names for $\mathcal{C}$, and 42,216 for $\mathcal{O}$.
2. The code to one of the main application components of each application was altered to:
   - read from a file a list of classnames to be tested,
   - attempt to load and initialize each classname, and
   - write to a file whether each initialization succeeded, and, if it failed, the error returned.
3. Testing a large number of classnames at once destabilized the JVM and gave unreliable results. Therefore, the set of potential classnames was sorted alphabetically and divided into small batches. Each batch was tested by starting the application, waiting for it to quiesce, and then stopping the application and JVM. During this process, the application was monitored for normal behavior.
4. If testing a batch of classnames resulted in abnormalities, the batch was subdivided to isolate the problematical name.
5. Steps 3 and 4 were then repeated with the classnames in reverse alphabetical order.

This process effectively determined which classes were available from the main classloader, which classes could not be successfully loaded and initialized, which classes, if loaded, seriously impaired the application, and, finally, an estimate of the classes whose initialization success was dependent upon the order of loading.

A complication was encountered on $\mathcal{C}$. The application persisted information in the filesystem that survived JVM restarts. This resulted in a significant performance degradation after a large number of runs which, in turn, caused $\mathcal{C}$ to enter an error-recovery state. In order to eliminate the possibility that this recovery state affected the loading of classes, the experiment was conducted inside a virtual machine (not to be confused with a Java virtual machine) which simulated an entire computer, including the hard disk and operating system. After every batch of classnames was run, the virtual machine state was reset back to its initial state, so that each batch was tested starting from a clean, normal environment.

What we expected, and what we should obtain from a system developed according to our interpretation of the Common Criteria, is that each classname would either load successfully, or would fail with a `ClassNotFoundException`, indicating that the class was not on the path. This was not the case.

To our great surprise, two classes were found that, when loaded, would crash the JVM.[1] These classes were shipped with the JVM and loaded native code during their class initialization that was meant to be executed on a different operating system. This affected both applications.

Additionally, on $\mathcal{O}$, nine classes were found that, when loaded in one particular order, caused $\mathcal{O}$ to cease to respond to its administrative commands.

[1]This was reported to the vendor, and these classes have now been fixed.

	$\mathcal{C}$	$\mathcal{O}$
Classes on classpath	55,364	29,673
Not on classpath	29,258	12,543
App failures		9
JVM failures	2	2
Initializable classes	54,884	29,196
Sometimes init.	27	5
Never initializable	451	470

(a) Loading Statistics

	$\mathcal{C}$	$\mathcal{O}$
ClassFormat		3
ExceptionInInitializer	26-37	
IncompatibleClassChange	2	
Instantiation	0-2	
NoClassDefFound	403-421	461-466
UnsatisfiedLink	18-24	
Verify	2	6

(b) Breakdown of Initialization Errors

Table 2: Class Loading Results

The complete statistics on the initialization attempts are found in Table 2. As indicated, 27 $\mathcal{C}$ classes and 5 $\mathcal{O}$ classes were initialized successfully during one run but not the other. Additionally, a number of classes which failed during one run failed with a different error when the order of class loading was reversed. To indicate this in the table, a range of numbers is shown, where the low number is the number of classes whose initialization attempt always returned that error and the high number is the number of classes which returned the error once among the two runs. Space limitations prevent us from describing each of these errors in detail, but we now discuss the most significant ones.

The vast majority of the initialization errors in both applications were NoClassDefFound errors. The cause of most of these errors was that one of the class's superclasses failed to initialize, usually because it was not on the classpath. In other words, loading of a class A was attempted, but one of A's interfaces or its parent was not available. These errors can also be thrown as a cascade effect of another classes' initialization failure.

It could be argued that UnsatisfiedLinkErrors and ExceptionInInitializerErrors are normal errors to expect in abnormal circumstances such as generated by our test; it is reasonable for loading native code to fail if the system is running under the wrong operating system, and similarly throwing an exception is understandable if the system state is incorrect in same fashion. However, under no circumstances can classes that cause either VerifyErrors or ClassFormatErrors be executed, so these should never be found in a working product. The remaining errors are also indicative of an abnormal build state, as linking errors can't be explained by the fact that the code was initialized unusually.

4.2 Class Duplicates

As Table 1 shows, the fact that in both applications the number of class files contained in the distributions is greater than the number of distinct classnames implies that the same classname must be defined in multiple jar files. However, this does not imply that any particular classloader will have multiple class definitions because the jars containing duplicate classes might be used by utility programs or be associated with different classloaders.

In order to determine whether classes are uniquely defined, we need to know the classpath used by the classloader. With $\mathcal{C}$, this information was provided by the Common Criteria team and the results of the previous analysis were consistent with it. In addition, $\mathcal{C}$ defined a set of "core" jar files which, due to the implementation of the main classloader, were guaranteed to be on its classpath. There was no implied ordering between these jars, so any conflicts between the classes defined by them was unintentional. We therefore analyzed both $\mathcal{C}$'s main classpath and its core jars.

Unfortunately, none of $\mathcal{O}$'s documentation included a statement of its main classpath. Fortunately, the results of the previous analysis were sufficient to uniquely determine the effective classpath of its

	$\mathcal{C}$ core	$\mathcal{C}$ rt	$\mathcal{O}$ rt
jar files	267	318	131
class names	36,129	55,364	29,673
repeated classes	348	577	2,711
distinct binary	19	155	1,556
distinct source	8	144	1,239

(a) Duplicate classes

	$\mathcal{C}$ rt	$\mathcal{O}$ rt
conflicting defns	10	199
missing superclasses	100	45
incomplete classes	275	359

(b) Dependency Check

Table 3: Results of Static Analyses

main classloader. (This is not true for $\mathcal{C}$.)

We created a program that, given a classpath, does the following: (1) Examines the classes defined by each jar, and determines which, if any, classes are defined in more than one file. (2) For each of the duplicated classes, extracts each version, and determines whether they are byte-wise identical. (3) For every duplicate class definition that is not byte-wise identical, it attempts to decompile each version to Java source in order to determine whether the duplicates correspond to different source code.

Notice that the decompilation step has two sources of error. First, some classes cannot be decompiled successfully: compiler optimizations or incompletely handled language features are frequently the source of this problem. Second, different compilers might compile the same source code in such a way that the decompiler will produce different but semantically equivalent source code. For example, a compiler might rearrange the order of exceptions declared to be thrown by a method – this does not change the semantics, but could result in the decompiler producing different results. For these reasons, the decompilation results are useful but not definitive.

The results of this analysis on the three classpaths: the core and runtime classpath of $\mathcal{C}$, and the runtime classpath of $\mathcal{O}$, are shown in Table 3a.

The cause of the source differences was investigated manually. In the $\mathcal{C}$ core, one difference was a false positive, generated by a different compiler re-arranging exceptions. Four classes only differed in embedded RCS control strings, indicating that a new source control branch had been created, but no changes had yet to be made in it. The remaining three classes were automatically generated by tools, and it appeared that different versions of said tools were used. Of the $\mathcal{C}$ runtime, most of the distinct source reports are false positives, generated by different compilers, and the same source code control string and tool versioning issues that were encountered with the core files. In addition, however, there were significant differences that were caused by two component groups independently incorporating a certain library and obtaining different versions of it.

Of the many $\mathcal{O}$ runtime source differences, 109 were clear false positives caused by compiler differences. On the other hand, 438 classes had differences of more than 40 lines each. A number of these appeared to be from different versions of libraries. In particular, greatly divergent copies of an XML library and an XPath library appeared.

4.3 Static Dependency Analysis

Although our first analysis discovered which classes could be loaded by the main component of our target applications, two major issues remain.

First, if multiple different versions of a single class exist in a classpath, then being able to load the class successfully does not imply that all versions are loadable, or equivalent. Although significant differences between versions would be detected by our second analysis if the versions could be decompiled, many classes are not amenable to available decompilers. We would like to be able to detect unloadable versions of classes even if they are "shadowed" by loadable ones.

Second, our first analysis required run-time testing. A static analysis is preferable, because such analyses have the potential to be integrated into build processes.

For these reasons, a prototype tool was constructed that, given a set of jar files, detects whether multiple class versions exist with inconsistent dependencies upon other classes, and which classes do not have their dependencies on other classes satisfied.

This tool maintains, for each class file processed, the object's name, whether it is a class or interface, whether or not it is abstract, the name of its superclass, if any, and the names of all interfaces it extends. With this information, it can detect and report inconsistencies such as two different versions of the same class with different parentage, or a class that is used as an interface by another class. It can also detect and report classes whose superclasses are not found in the given jar files.

Table 3b shows this tool's results against the main classpath of $\mathcal{C}$ and $\mathcal{O}$. In each application we find that the main classpath contains multiple definitions of the same class with different ancestries. In $\mathcal{C}$ there are 100 classes that are used as a superclass for a class on the classpath, but which are not themselves found on the classpath, and this accounts for 275 of the NoClassDefFound errors that we observed in our first analysis. In other words, 274 classes could not be loaded because they have a superclass that is not on the classpath. For $\mathcal{O}$, 45 classes are missing and this accounts for 359 of the NoClassDefFound failures.

5 Java RMI Issues

In this section, we describe our investigation of Java's Remote Method Invocation (RMI) [16] mechanism in relation to classloading issues. We selected the RMI component specifically because RMI accepts remote requests and therefore offers the possibility of remote attacks.

5.1 RMI Details

The RMI subsystem implements a distributed object model for Java systems. It provides a distributed object model in which code in one JVM can invoke methods on objects that reside in different JVMs, different physical machines, or even non-Java object systems, like CORBA. It aims to preserve Java's type-safety guarantees, and maintain Java security properties.

A remote object is one whose methods can be made available to be invoked remotely. Technically, this is indicated by the class implementing the `java.rmi.Remote` interface. Remotely available methods can take arguments. These arguments have to be either other remote objects or serializable objects.

A serializable Java object is one whose state can be converted or *serialized* into a series of bytes, and then converted back or *deserialized* into another Java object with equivalent state to the original. Serializable objects are denoted as such by implementing the `java.io.Serializable` interface. Observe that deserialization results in a copy of the original object.

Consider a remote method which takes one parameter of type X. When an invoker on another machine passes it an argument y, then, for obvious security reasons, it is the receiving system's responsiblity to check that y's actual type is a subtype of X. Similarly, if the method returns a value z, then it is the caller's responsibility to verify that z's actual type is a subtype of the method's formal return type.

As the deserialization process involves dynamic classloading, this process will be described in detail.

The arguments and return values can be either remote objects, in which case they are passed via proxies, or serializable objects, in which case they are converted to a byte stream. The proxy mechanism for remote objects is not of concern here. The processing of serializable values, however, involves classloading issues.

According the RMI and Serialization specifications [16, 17], when RMI receives a remote method invocation, it has to instantiate the arguments to the call, invoke the method on those arguments, and then send the return value back to the invoker. For any arguments which are serializable, the following steps take place:

1. Reading the name of the argument's class from the stream.
2. Retrieving the class with the name of the argument's class.
3. Creating an instance of the class.
4. Restoring the fields of the object from values on the stream. User-written code may be called to do this.
5. Optionally, replace the newly created object with another object, possibly of a different type. User-written code may be executed to do this.

Step 3 requires that the class named be dynamically loaded, if it hasn't already been loaded. Also, step 4 is a recursive call: if any fields have values that are objects, then this process must be performed to retrieve them.

5.2 Specification Flaw

We observed that the specifications[2] contain a flaw related to dynamic class loading. The remote method parameters are received through a network connection, and thus should be considered to be untrusted input. This input contains the name of a class which is loaded in step 3 of the deserialization process. In neither specification is there any requirement that the class not be initialized until it has passed appropriate type checks, such as ensuring that the class is serializable or of the correct type for the method parameter.

The following two-part attack is thus possible: Say that "BadClass" is a class available to a server, whose class initializer has undesirable consequences, such as crashing the system. Say also that the server has made available a remote method (call it "`foo`") which has at least one argument. Invoke `foo` with an argument that claims to be of type BadClass. This can be done either by direct manipulation of RMI's wire protocol, or by making another version of BadClass on a client system which is of the correct type to be an argument to `foo` and invoking the method normally. Either way, according to the specification, it would be legal for the server to initialize BadClass before doing any validity checks.

As described previously in this paper, we found JVM-supplied classes that would cause a system crash, and application-supplied classes which would cause application failure. Any of these 11 classes could be used in this attack.

5.3 Implementation Flaws

Even though not mandated by the specification, implementations could implement the proper checks. To investigate this, we tested the above attack in sixteen scenarios: Two different vendor's JVMs, the two currently-supported major releases from each vendor (both with the latest patches), and using two different transports (the default Java implementation, JRMP, and the CORBA-compliant transport, IIOP, intended for enterprise applications). Additionally, we tested both serializable and non-serializable classes as the payload for this attack.

The results were that every JVM, with every transport, incorrectly loaded and initialized classes, if the target classes were serializable. Additionally, one vendor's IIOP implementation would incorrectly load

[2]A revision to the specifications to address the issues described here is planned for the upcoming Java 7 release.

and initialize any class, serializable or not, and thus any class could be remotely forced to be initialized.[3] In particular, servers using this implementation would be able to be crashed remotely by an attacker making use of the JVM-supplied crash-causing classes.

5.4 Underlying Causes and Solutions

As we discovered, all RMI implementations classloaded and initialized serializable class names received as claimed RMI method arguments, even if the class in question did not match the formal argument's type. Because of this, systems that use RMI would still be vulnerable to a two-part attack that makes use of a serializable class whose initialization code can cause system failure (although we do not know of any such classes in product code). It is natural to consider whether this flaw could be fixed.

At the time of a RMI method invocation, the system has available to it the compile-time types of the method's formal arguments. Therefore, it seems possible that, when deserializing the actual arguments a type check could be performed after the arguments' classes have been resolved but before they have been initialized.

Unfortunately, this is intrinsically infeasible without breaking core Java functionality. Although the replacement functionality in step 5 of the deserialization process makes it impossible to determine whether an argument is type-correct before step 3 (because the replacement process can change the argument's type), there is a more fundamental reason: the recursive process in step 4. At this point, the incoming object's fields are being deserialized, possibly under the control of user-defined code. Type information about these fields is therefore unavailable, other than the fact that they must be serializable. Even for non-user-defined field assignments, however, the fact that Java's Collection classes are frequently used and do not provide any information about the types of their contents[4] means that even system-defined field assignments would frequently be unable to prevent attacks that use serializable objects. As a result, even if the system type-checked the argument types, an attack could be performed by sending the name of a dangerous serializable class as a field value.

We observe that this problem is caused solely by the interaction between dynamic classloading, the ability to remotely invoke methods with serialized arguments, and user-defined class initializers. Each of these is a fundamental feature whose absence would greatly reduce Java's functionality. We argue, therefore, that the ability of remote parties to force execution of class initializers of arbitrary serializable classes on the server's class path is an unescapable consequence of basic design decisions, and not the result of a flawed implementation.

5.5 Impact and Remediation

We do not wish to overstate the importance of this flaw; only the class initializers of classes already available locally can be made to be executed. Although theoretically such methods can do arbitrary actions, in reality the results are most likely to be crash failures. In the cases we've seen, most failures have been due to invocation of native code for another platform or explicitly changing the path on which the system looks for native libraries. Furthermore, in response to this work, the above-mentioned IIOP transport was corrected, so that with all tested JVMs, only serializable classes are vulnerable. Our search for problematical serializable classes in product code has been fruitless, and such code seems unlikely.

On the other hand, the fact that code with faulty class initializers has been found shipped with a JVM and in both commercial and open-source applications is disturbing. Also, a $\mathcal{O}$ class, when initialized,

[3] This flaw was fixed by the vendor as a result of our report.

[4] Although Java 5.0 and later releases provide typed Collections, these types are stripped out during compilation and do not exist at runtime.

started a service that seemingly was expecting to handle database activities, which potentially could be further exploited. It should be noted as well that, while denial-of-service attacks on Java systems are not hard to perform simply by sending large amounts of message traffic, this form of attack simply requires the sending of one small message which can be easily varied. Such messages are hard, if not impossible, to filter using a network firewall.

6 Flaw Impact

In previous sections, we have detailed various build practice deviations discovered in our target applications. In this section, we will evaluate their impact on security and reliability, particularly in light of the RMI flaw discussed above.

In summary, we have found classes which were, in rough order of increasing importance: (1) uninitializable, (2) duplicated, (3) susceptable to initialization order effects, and (4) caused application or JVM failure.

Uninitializable classes pose a low risk, as their code cannot be executed. The existence of these classes, however, raises questions about the application's test process. If these classes had some functional use, then their failure should should have been detected during test. The benign possibility is that these classes simply represent now-dead code or the accidental inclusion of unnecessary code.

Duplicated classes have a formal impact on Common Criteria evaluations. The Common Criteria (Part 3)[1] includes the following requirements, all of which are adversely impacted by duplicated classes:

- Basic Modular Design - ADV_TDS.3.8.C: The design shall describe each Security Functional Requirement (SFR)-enforcing module in terms of its SFR-related interfaces, return values from those interfaces, and called interfaces to other modules.
- Use of a CM system - ALC_CMC.2.3C: The CM system shall uniquely identify all configuration items.
- Implementation representation CM coverage - ALC_CMS.3.1C: The configuration list shall include the following: the TOE itself; the evaluation evidence required by the Security Assurance Requirements (SARs); the parts that comprise the TOE; and the implementation representation.
- Implementation representation CM coverage - ALC_CMS.3.2C: The configuration list shall uniquely identify the configuration items.

One could argue that if the software is properly compiling and loading, that these strict requirements could be waived by an evaluator. However, these duplicated classes have a practical reliability impact that shows the importance of these Common Criteria requirements. The most likely reason for having the same class appearing in multiple jars visible to the same classloader is that the application was divided into independently developed components, and two or more components either incorporated the same library, or used a conflicting naming scheme. Since a classloader can return only one version of a given class, this means that if the versions are different, only one component will be run against the code it was intended to.

Clearly, this situation invalidates all the component unit-tests. Furthermore, this has a pernicious effect upon the defect tracking process. Consider two components, A and B, which accidentally have a class version conflict. If B's version is chosen by the classloader, then A will run against code it wasn't tested against. If B's version of the class is defective, or at least not suitable for A's use, then the resulting defect will be entered against the failing component — A. Unfortunately, A's developers will inspect their version of the class which is, in fact, correct. $\mathcal{C}$'s developers believe that some long-unresolved defects in their product were the result of this problem.

Even if multiple versions of the same class are identical, if all versions are not simultaneously and identically updated during maintenance, then bug fixes have a large risk of being ineffective or being reverted.

Severe risks are exposed by the classes which can be made to fail by altering the order of classloading, and by the classes which cause application or JVM failures if loading is attempted.

There are two methods by which a potential attacker can cause a classloading attempt: through a child classloader (i.e., via a client plugin) or by means of a remote method invocation.

It is easy for an application plugin in $\mathcal{C}$ or $\mathcal{O}$ to attempt to load and initialize any class. Even if a plugin isn't deliberately malicious, it is subject to attack itself and any flaw whereby a party can cause it to attempt to load a class of the party's choice would be fatal. Therefore, this means that all application plugins must be trusted, which not only greatly increases the amount of trusted code, but since plugins aren't under the control of the $\mathcal{C}$ or $\mathcal{O}$ developers the responsibility for ensuring the trustworthiness of the plugins is transfered to the plugin developers, who may not be qualified or aware of the system's security requirements.

Our discovery of the RMI flaw raises the importance of these failures. Even when all Java releases are remedied to the point of only being susceptible to attacks involving serializable classes, this still is an area of concern.

7 Related Work

The challenge in conducting our experiment was to extract information about a product's build which was either missing or inaccurate. There are a number of tools which, if they had been used by either $\mathcal{C}$ or $\mathcal{O}$, would have eased various portions of our work.

Maven [3] and Ivy [10] are two Java project management tools which include jar-level dependency tracking features. Both tools allow the user to specify versions of jar files and declare that one jar depends upon another, or a specific version of another. Although such tools are highly useful, they do not provide any mechanism for discovering dependencies between jar, nor are they able to detect conflicts between jars. Frequently in $\mathcal{C}$ and $\mathcal{O}$ classes were found to be repackaged into multiple jar files, with different associated content. This seemed to be the result of subcomponents attempting to minimize the number of jar files contributed by repackaging their dependencies into their output. These tools aren't able to detect or manage such cases.

OSGi [14] is a Java system in which components can be organized into modules. Each module can state other modules which it depends upon, and what classes it makes available to others. At runtime a classloader heirarchy is built which allows each module to access its dependencies and be isolated from others. Unfortunately, due to runtime overhead, it enforces only the dependancies that it knows about. Also, the dependency information must be given to the system. However, if a project was developed using OSGi and was careful to declare strict dependencies (ie, not importing or exporting "*", which matches all classes), then most of the issues found in our experiment would be avoided. Interesting further work would be tooling to reverse-engineer existing code into appropriate OSGi modules.

Our experiment included determining whether classes had their dependencies satisfied, by dynamically loading them and by statically extracting the class hierarchy information. It would be interesting and useful to do this entirely statically, and to use information about which other classes a given class interacts with. Unfortunately, this is in general incomputable, because Java can load classes given a dynamically created name. However, various work has been done to approximate this information, although to our knowledge none of it takes into account classloader hierarchies.

The product "Jar Jar Links" [18] from the now-defunct company Tonic Systems claimed to discover dependencies between classes by extracting both explicit class references and strings which are of the proper form to be Java class names. Theoretically, this approach is subject to missing classes whose names are constructed from strings outside the class and through such mechanisms as property files. It can also falsely decide that there are dependancies, due to such things as strings which happen to be of the correct format. As the product no longer exists, it is hard to evaluate the effectiveness of this approach, although the fact that they listed libraries which are "Jar Jar friendly" implies that they had a large number of errors.

For the purposes of constructing call graph information, Livshits, Whaley and Lam [13] take a multistep approach to determining what other classes are used by a given class. First, for every dynamic class load, they use points-to information to attempt to determine the actual names of classes created dynamically. If this fails, then they use casting information. This process assumes that when a dynamically created class instance is cast to a specific type, then an unsuccessful cast represents a program failure and the object will not be further used. Therefore, cast information can determine the types of these objects. Although they achieved good results, they note that in many cases casts were made only to high-level, general-purpose classes, which resulted in too many possible target classes to be suitable for their purposes. It is not clear how effective this would be for our purposes. For security purposes, assuming objects that fail casting are never used is dubious, particularly since unexpected control-flow paths are notorious for containing security flaws. On the other hand, we are not attempting to construct call graphs, and therefore can deal with a large number of possible targets.

Finally, Kozen and Stillerman, in [11], describe an algorithm for determining whether Java class initializers can be executed eagerly, or in other words, if the operation of a class initializer depends upon other classes having been initialized first, or other prior computation. Such a determination would statically discover the many inconsistent results of our dynamic check. Unfortunately, their algorithm is only sound when no exceptions are thrown during initialization, there is no concurrency, no reflection, and no native methods. All of these conditions fail to hold for both $\mathcal{C}$ and $\mathcal{O}$, and probably most large Java systems. Although we are curious as to how well this algorithm performs despite these issues, their implementation is no longer available.

8 Conclusions and Future Work

In this paper we have derived Java build-process requirements, based upon the Common Criteria, taking into account the challenges of Java's dynamic classloading and the namespace partitioning created by its classloader hierarchy mechanism. Two similar and mature Java applications. one commercial and one open-source, were analyzed against these requirements. To do this, tests were done with instrumented versions of the applications, and two custom-built tools were used to do static analyses. Similar, previously unknown, flaws were found in both applications. These flaws included multiple versions of classes available to the same classloader, classes whose superclasses were missing, classes whose references to other classes were inconsistent, dependencies upon the order of classloading, and application or JVM crashes upon class loading. Finally, the impact of these flaws was evaluated. These flaws not only affected the testing, debugging and maintenance processes, but developers of the commercial product believe that these flaws are the cause of some long-outstanding unsolved bug reports. Most notably, we discovered that these problems can be used as the second part of a two-part remote attack in which attackers can disable the JVM or application by sending a small number of specially-crafted messages. This attack cannot be fully addressed, as it is the intrinsic result of core design decisions.

Given our findings, we conclude that our methodology is useful in detecting flaws in Java software, and that Java's classloading raises security concerns. The flaws we found were previously unknown, even though the applications we inspected were both mature, and the commercial application had a history of inspections by other techniques. Therefore, we also conclude that our methodology detects a different class of vulnerabilities than flaw-based techniques. Since these flaws would have been avoided had the Common Criteria or similar development processes been used, this provides evidence that such development processes avoid issues that are to otherwise hard to detect or prevent.

Our methodology is of particular importance in the construction of very large Java-based applications. Had either product been developed by a single team of people, many of the class duplications might have been detected by manual procedures. However, these problems were very hard to otherwise identify, because the products were very large, incorporated other already existing products, and were under development by multiple teams in multiple distributed locations. It is for precisely these types of very large software developments that configuration management requirements and tools have been developed.

A clear direction for future work is to augment build tools to detect such violations automatically. Our prototype tool would suffice for ordinary Java programs, in which there is only a single classloader. However, those programs which, like $\mathcal{C}$ and $\mathcal{O}$, partition their classes into namespaces pose a challenge, since the classloader hierarchy and each classpath is not explicitly represented anywhere. The use of OSGi, as it explicitly represents associations between sets of Java code, should enable build processes to effectively analyze OSGi applications. Additionally, tools capable of partitioning pre-existing applications into OSGi modules would be very useful.

Finally, although we have focused on the build process and Java applications, we believe that other aspects of the software development process and other languages could also benefit.

References

[1] Common criteria for information technology security evaluation, parts 1, 2, and 3. Version 3.1, Revision 1, CCMB-2006-09-001, CCMB-2006-09-002, CCMB-2006-09-003, Sept. 2006. URL: http://www.commoncriteriaportal.org/public/developer/index.php?menu=2.

[2] Common methodology for information technology security evaluation. Version 3.1, Revision 1, CCMB-2006-09-004, Sept. 2006. URL: http://www.commoncriteriaportal.org/public/developer/index.php?menu=2.

[3] Apache Software Foundation. Apache maven project. URL: http://maven.apache.org/.

[4] M. Croxford and R. Chapman. Correctness by construction: A manifesto for high-integrity software. *Crosstalk*, December 2005.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operation System Design and Implementation (OSDI)*, October 2000.

[6] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in system code. In *Proceedings of the Eighteenth ACM Symposium on Operation System Principles (SOSP)*, October 2001.

[7] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 345–354, Washington, DC, 2003.

[8] S. Garfinkel and G. Spafford. *Practical UNIX Security.* O'Reilly & Associates, Sebastopol, CA, 1991.

[9] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, June 2002.

[10] Jayasoft. Ivy: The agile dependency manager. URL: http://www.jaya.free.fr/ivy.html.

[11] D. Kozen and M. Stillerman. Eager class initialization for java. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 71–80, London, UK, 2002. Springer-Verlag.

[12] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, pages 177–190, 2001.

[13] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for java. In *LNCS 3780*, pages 139–160),, Nov. 2005.

[14] OSGi Alliance. Osgi^{TM} - the dynamic module system for java. URL: http://www.osgi.org/.

[15] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the Tenth USENIX Security Symposium*, pages 201–216, 2001.

[16] Sun Microsystems. Java remote method invocation specification, revision 1.5.0. URL: http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf.

[17] Sun Microsystems. Java serialization specification, revision 1.5.0. URL: http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html.

[18] Tonic Systems. Jar jar links - manual. Google cache of URL: http://www.tonicpoint.com/products/jarjar/manual/.