Jeremy Hall

Protection of Agent Teamwork

Table of Contents

Goal 1
Overview 1
Class Overview 2
Javassist[i] 3
Usage 3
Limitations 3
Java Limitations 4
Changes made to Agent Teamwork 4
Proxy Creation - Logical Flow 4
Proxy Method Execution – Logical Flow 7
Conclusion 8

I. Goal

1. The main goal of this project was to protect a Java program's logic by encrypting a Java class's methods. This is accomplished by first encrypting a Class, and then only decrypting the parts of that Class as they need to be used, after which they are thrown away. An additional goal was to have as little an impact as possible on existing code. This means that the project should require as little change in the classes that it protects as possible. A program using an encrypted class should also have minimal required changes to use the encrypted classes.

II. Overview

1. The reason for protecting a Java program's logic is to prevent sophisticated attacks such as an attacker scanning the memory of a computer while its Java program is running. Another situation is forcing a memory dump to read the contents of a running Java program.
2. This project relies heavily on an open source Java library called Javassist. Javassist allows for manipulation of Java bytecode files, as well as run-time compilation of dynamically created classes.
3. For a class to be encrypted, its bytecode .class file must be loaded by a special classloader. In practice, this means using an instance of the ClassRipper class.

4. When a class is “ripped”, all of its methods are changed to forward their parameters to a single “Invoke” method. All of the class's real methods are encrypted, and stored inside of a class which knows how to decrypt and return an instance of a class that contains the required method (In Java, a method cannot exist on its own, it must exist inside of a Class) . A one-time-use classloader is used to actually create the instance of the class containing the method.
5. The decrypted method is invoked, and all references to the instance of the class containing the method lose scope (including the class loader that loaded the class), so that the instance is garbage collected, and no unencrypted versions of the code remain.

III. Class Overview

1. ClassRipper is the class that a program wishing to use an encrypted class will use. This class will “rip” a Java .class file, and return an instance of an object which contains the same public interface (note that this does not literally mean a Java interface, but in fact means the returned class will have all the same methods, variables, interfaces, and inherited class) as the Class stored in the .class file.
2. ProxyCreator is the class which holds all the actual steps for generating a new encrypted Class from the original Class representation. I refer to the modified classes as proxies.
    1. There are two slightly different proxy types that need to be created. The first is the Class whose instance will be returned to the user. This class contains none of the method information from the original class. It just has skeleton methods which forward all information to an “Invoke” method.
    2. For every method that existed in the original class, a separate Class must be created to store said method. This is the second type of proxy. This proxy actually contains a method from the original class in it for execution. In addition to containing a real method, however, it must also contain skeleton methods just like the first type of proxy. This is because the method that this proxy contains may reference another method inside the original class.
3. ClassServer is essentially a wrapper around a Hashtable. ClassServer accepts a method name, and returns the correct Class wrapped method for a proxy to use. ClassServer uses the method name as a key into the Hashtable, decrypts the Class that is returned to it, and then instantiates an instance of that Class to return to the proxy.

4. DisposableClassLoader is a ClassLoader which is to be used with only one class. ClassServer uses this class to instantiate the Class returned to it from the Hashtable. This is because for the class definition to be garbage collected, the ClassLoader that loaded it must be dereferenced as well.
5. The Crypto class provides the methods to encrypt and decrypt. It also handles the generation of the secret key. Its encryption implementation is a symmetrical cipher.
6. ByteEncrypt extends Crypto and is used specifically to handle the encryption of bytes. This class is used by ClassRipper to encrypt the individual class wrapped methods which are inserted into the Hashtable.

IV. Javassist[i]

1. As mentioned earlier, Javassist allows for the manipulation of bytecode, and the run-time creation of classes. Before modifying a class with Javassist, the class must be loaded into one of Javassist's own class types. Most of the Javassist code is in the ProxyCreator class. This project loads the class from a .class file from the filesystem. This is loaded into a CtClass, which is a Javassist representation of a Class. CtMethod is Javassist's representation of a Java Method.
2. CtClass allows for the direct manipulation of all of a Class's internal members. CtMethods can be created, added to a CtClass, removed from a CtClass, and modified. CtFields are representations of Java Fields, and can be modified similarly to CtMethods.

V. Usage

1. To create an encrypted class, the first thing one must do is to instantiate an instance of ClassRipper.
2. Call the method *Object ClassRipper.StartRip(String location)*. The parameter points to the .class file of the Class which should be encrypted. The returned object is an instance of an encrypted version of the class. If the Class to be encrypted needs to have arguments passed to its constructor, then use *Object ClassRipper.StartRip(String location, Object[] args)* instead, and pass an Object array holding the constructor arguments along with the .class file location.

VI. Limitations

1. One major limitation placed on a Class which wants to be encrypted is that the constructor may not reference another method in the class. This is because of the order in which the

constructor is called, and the ClassServer which contains the encrypted methods is passed to the proxy. When the encrypted version of the class is created at runtime, one of the methods added to it is called "_setClassServer" and its meant to accept the ClassServer so that the proxy can retrieve its methods. This method must by definition be called after the constructor, which means that if the constructor references another method in the class, the proxy won't have its ClassServer yet so it cannot retrieve the method and an exception will be thrown.

2. Encrypted classes are not threadsafe. This is because there is a mechanism to send and retrieve class variables each time a method is called, and this portion has not been synchronized (although it probably wouldn't be hard to do).

VII. Java Limitations

1. There are certain limitations in Java which have made the project harder to implement. One of the biggest limitations is that once a Class has been loaded by a Java ClassLoader, it cannot be replaced, or removed individually. To get rid of a Class, the entire ClassLoader must be garbage collected.
2. Methods must be wrapped inside of a Class, they cannot exist on their own.
3. Once a Class has been loaded into Java, it cannot be modified (there actually is a way to allow this, but it can only be done by running Java in debug mode).
4. Unless the class being encrypted belongs to an interface, the only way to access the methods of the encrypted class is through reflection (unless you cast the object as its superclass). This is because the actual class isn't created until run-time.
5. In Java, its very difficult to remove all traces of something from memory. This is because garbage collection is nondeterministic. Its impossible to know when it will happen, and it's impossible to force it to happen. This means that unencrypted versions of methods can be floating in memory for some time before being removed, thus reducing the effectiveness of the just-in-time decryption method.

VIII. Changes made to Agent Teamwork

1. UWAgent.java
    1. In the makeObjectFromClass() method, the ClassRipper had to be used to instantiate a new encrypted class. Also, the current agent's ClassServer had to be extracted so that the new encrypted class could use it.

2. UWInject.java
    1. Changed the original class loading sequence to use ClassRipper (public void engine()).
    2. Had to get the bytecode of the encrypted Class from ClassRipper and put it into the classesHash, instead of allowing the bytecode from the .class file to be put there.
    3. The classesHash is created earlier than it used to be, because the bytecode must be passed through the multitude of engine() methods.

IX. Proxy Creation - Logical Flow

1. The process begins with a user creating an instance of ClassRipper, and giving it a location of where to find the .class file of the Class that should be encrypted.
2. The next step is to load the Class from the .class file into a Javassist CtClass where it can be freely manipulated.
3. Once the CtClass has been created, the next step is to create the Hashtable of encrypted methods. For every method defined in the Full CtClass, a new proxy must be created to contain that method.
4. The creation of the proxy involves several steps, and takes place in the ProxyCreator Class:
    1. First a new blank CtClass is created with a different name from the original (it has to be different because Java requires it), so "Class" is prepended onto the front of the name of the method that this proxy will contain.
    2. The newly run-time generated class will start out identical to a Java "Object".
    3. Next, if the Full CtClass extends another class, that inheritance is copied so that the new CtClass extends the same class.
    4. Any interfaces that the Full CtClass implements are also copied to the new CtClass at this time.
    5. The next step is to add a field for storing a reference to the ClassServer. This is the Class that stores the encrypted methods. The proxy class needs this in case its one encapsulated method calls another method. Since each proxy only holds one method, it must contain the same Invoke and shell-method architecture as the original Proxy class which holds no methods of its own.
    6. The next step is to create a method to set the ClassServer field created in the last step.
    7. Now a method to extract every field in the proxy and return their values as an array of Objects is added. This is because each time a method is called, a proxy object is

spawned to handle the method. The proxy object's method is called, and thus the fields of the method-proxy are manipulated by the method, and not the original parent-proxy. In order to propagate the changes in the field values back to the parent-proxy, all of the child-proxy's fields must be accessed.

8. Next, a method to set all of a proxy's fields with an Object array is created. This is for the passing of information from one proxy to the other, and is the complement of the previous step's method.
9. The next step is the adding of the invoke method. This is really the heart of the proxy. The invoke method performs the following functions:
    1. It accepts the method name, parameter types, and method arguments from a shell method.
    2. It then asks the proxy's internal ClassServer for a Class object which contains the necessary method.
    3. At this point a handle to the necessary method is created using reflection.
    4. Now an instance of the Class containing the required method is created.
    5. The current proxy now uses the method given to it to put all of its class variables into an Object array, and passes those variables to the newly instanced Class.
    6. Now the method is invoked, and any return value is stored as an Object.
    7. All class variables are now copied back to the current Proxy, so that any changes made by the instanced Class can be seen.
    8. Finally the returned Object is returned to the shell method that invoked the invoke method.
10. Now all the fields from the full CtClass are copied to the runtime generated proxy class.
11. Finally the shell methods are created. All the methods in the full CtClass are cycled through. The shell methods on the runtime generated class will have the same names, parameters, and return type, but the actual method will differ. They just forward the parameters and method name to the invoke method, instead of actually performing the data manipulation that their full CtClass method counterpart would do. This happens for all the methods in the full CtClass, except for the one that this proxy will contain. In that case, the method is copied over in full.
12. Now that the new CtClass has been built, it is output to bytecode, and returned to the

ClassRipper class which can then encrypt it, and store it as a key/value pair in a Hashtable, with the key being the method name, and value being the encrypted bytecode.

5. Once all of the method's have been stored inside of a proxy, encrypted, and put into a Hashtable, the ClassServer is finalized. The ClassServer is a wrapper around the Hashtable of encrypted methods, and handles the decryption and creation of the Class objects from the proxy's bytecode. First an instance of ClassServer is instantiated by passing in the Hashtable as an argument to its constructor, and then the secret key to use to decrypt the Hashtable's encrypted proxies is passed to it. It is here that someone implementing a key distribution system would change things, as the key should not be immediately given to the ClassServer, but should instead be passed to the ClassServer once it has arrived at another UW Place.
6. Now the Parent Proxy which will actually be the encrypted Class used instead of the original class is created.
7. Once again the ProxyCreator Class is used, since the creation steps are so similar to creating the method containing proxies. There are a couple differences however:
    1. The constructors are copied, because this Proxy will not have its values initialized by having all fields copied from a parent proxy.
    2. All the methods are shell methods (except for the constructors), there is no full method inside of this Parent Proxy.
8. Once the Parent Proxy Class has been created, an instance is generated. If an Object array was passed to ClassRipper.StartRip(), then that array will be used to pass constructor parameters to the proxy instance. During this time, the ClassServer is passed to the proxy as well, so that the proxy may begin using the stored methods. After this is complete, the proxy is passed off to the user's program.

X. Proxy Method Execution – Logical Flow

1. First, a method gets invoked on the Parent proxy object.
2. The method that gets invoked is really just a shell method. ProxyCreator.CreateAndAddShellMethods() is where these shell methods are added. The method's name, parameter types, and arguments are passed on to the proxy's invoke method.
3. The invoke method first requests a Class object from the internal ClassServer. It does this

by calling ClassServer.getClass(), and passing in the name of the method that was passed as a parameter. The Class object which is returned has the necessary method inside of it.

4. Using reflection, the method inside of the newly returned Class object is referenced, and an instance of the Class is created. This class instance is also a proxy, with shell methods for all methods in the original class except for the one method that it was created for.
5. The values of all class variables are now gathered from the Parent proxy and passed to the child proxy.
6. The method in the Child proxy is invoked now, and any requisite parameters are passed to it.
    1. If the method in the Child proxy references another method in the class, then this whole process happens recursively again. It will be referencing a shell method in the Child proxy, which will redirect to the Child's invoke method, which will go through this same process.
7. The class variables of the Child proxy are now returned to update the Parent's variables in case any changes were made.
8. Any return object is now returned to the shell method, which returns it to the original caller of the Parent proxy's method.

XI. Conclusion

1. It's possible to create encrypted versions of Java classes with minimal impact on the users of those classes. This method will only be truly effective with a key distribution system however. The encryption of classes is designed to thwart a sophisticated attack on a running Java program. It is only one part is securing such a program from prying eyes.

---

i Javassist web page: http://www.csg.is.titech.ac.jp/~chiba/javassist/