Secure Method Calls by Instrumenting Bytecode with Aspects

Xiaofeng Yang and Mohammad Zulkernine

School of Computing, Queen's University
Kingston, Ontario, Canada, K7L 3N6
{yang,mzulker}@cs.queensu.ca

Abstract. Today most mobile devices embed Java runtime environment for Java programs. Java applications running on mobile devices are mainly MIDP (Mobile Information Device Profile) applications. They can be downloaded from the Internet and installed directly on the device. Although the virtual machine performs type-safety checking or verifies bytecode with signed certificates from third-party, the program still has the possibility of containing risky code. Inappropriate use of sensitive method calls may cause loss of personal assets on mobile devices. Moreover, source code is not accessible for most installed applications, making it difficult to analyze the behavior at source-code level. To better protect the device from malicious code, we propose an approach of bytecode instrumentation with aspects at bytecode level. The instrumentation pinpoints the location of statements within methods, rather than at the interface of method calls. The aspects are woven around the statement for tracking. The weaving is performed at bytecode level without requiring source code of the program.

Keywords: Bytecode instrumentation, aspects, code security.

1 Introduction

Mobile technology is very popular nowadays. More and more applications are running on mobile devices such as cellphones and PDAs, etc. Mobile code can easily migrate from one site to another, and even to a personal cellphone. Java applications are popularly running on mobile devices that embed Java Runtime Environments. User's information may be at risk as they are not usually aware of the programs they are running on their devices, for example, the unauthorized SMS (Short Message Service) sending programs [1] can be executed without any authorization of a user. Such sensitive information may also include personal data, geo-locations, and passwords [18].

The MIDP (Mobile Information Device Profile) is the core profile for the development of applications on Java mobile devices. The MIDlet is the application developed with MIDP. Many cellphone manufacturers already have MIDP enabled devices available. The security vulnerabilities shown in [1] indicate that the incompatibilities between different versions of MIDP caused security issues

E. Gudes, J. Vaidya (Eds.): Data and Applications Security 2009, LNCS 5645, pp. 126–141, 2009.

by unprotected use of sensitive method calls. It is necessary to track such sensitive method calls to avoid abuse of APIs before the MIDP specification becomes perfect. Much research have been using static analysis techniques to assess the quality and the security of downloaded Java mobile applications [10,11,17]. They present approaches of validating downloaded MIDlets by verifying source code, method properties and signatures with certifications using Point-to analysis [10] or dependency analysis [11, 17]. Although these static analysis techniques are helpful, they cannot fully address the problems occurred during runtime. By static analysis, it is hard to tell which method call is at risk. For example, `Display.setCurrent()` is a common method used to update the screen on mobile devices. However, it was also used by hacker group to run malicious code on devices. The current screen is obscured with other items asking a user for permitting SMS. Then the user approves the permission for the request shown on the obscured screen rather than the real screen. Although this vulnerability is prevented in Sun RI (Reference Implementation) of MIDP by performing appropriate locking and unlocking access to the display, as there could be other sensitive method calls, we believe that it is necessary to control such method calls. Moreover, by static analysis, the method call `RecordStoreFile.deleteFile()` is allowed for use. However, this deletion operation can cause data loss during runtime with the inappropriate use of the developers as reported in [1]. Both examples show that such method calls should be controlled even though they are declared valid in the static analysis phase.

Aspect-oriented technology has been widely used to separate cross-cutting concerns (like logging mechanisms) from functionality modules. Currently, most implementations of aspect languages have been using the recompilation of the original source code and the instrumentation code to perform the instrumentation. However, in real world, most programs are delivered without source code. Therefore, we consider the instrumentation at bytecode level. Bytecode is an intermediate machine language that instructs virtual machines how to execute operational instructions. When the bytecode is loaded into JVM, the JVM verifier only validates the correctness of the bytecode's structure, data type, and symbolic references. BCEL (ByteCode Engineering Library) is used to manipulate and modify bytecode [20] at bytecode level, but it does not support aspect features for instrumentation and requires good expert knowledge to instrument bytecode at appropriate location. AspectJ is a popular AOP (Aspect-Oriented Programming) language [6] that implements aspects in Java. It provides the flexible aspect constructs and expressive pointcut patterns. It requires the source code of the program for re-compiling. Binder *et al.* [14, 15, 16] propose a series of approaches of bytecode re-engineering by instrumenting Java's standard class libraries. However, it is not realistic to modify the implementation of system libraries. We cannot rely on the platform's upgrade since it is difficult to recall and patch up all platforms. For mobile platforms, the bytecode instrumentation was not performed using aspects for security concerns at bytecode level.

In our approach, we modify the program with aspects at bytecode level without accessing source code and without knowing the context of sensitive method

calls in advance. We manipulate the delivered bytecode without recompiling the original programs, and we instrument the bytecode instructions during class loading. We use a bytecode library, Javassist [3], to manipulate bytecode instructions from class files. We also define the corresponding aspects in modules using `expression editor` provided by Javassist. While loading `classes` into the virtual machine, security concerns are instrumented into the place where insecure method calls might happen. Our approach does not require access to source code and does not change the original class. Moreover, rather than weaving aspects at the interface level, we weave directly around the statement of the method call for tracking. It does not change the implementation of the method call but it changes the behavior of invoking the method call. Our approach determines the context of a sensitive method call dynamically, while the others have to know the names of methods that are using the sensitive method call in advance.

The rest of the paper is organized as follows. Section 2 presents the Java ME (Java platform, Micro Edition) technology, security risks on Java ME platform, bytecode instrumentation, and aspect-oriented technology. Section 3 demonstrates our approach of bytecode instrumentation with aspects to address sensitive method call concerns. Section 4 presents the implementation of our approach and experiments. Section 5 compares our work with the existing related work. Section 6 concludes the paper.

2 Background

2.1 Security Risks on Java ME Platform

Java programming language was initially invented for using in embedded systems. It is becoming more and more popular for mobile embedded systems and mobile devices. Java ME (Java Platform, Micro Edition) is a platform for the development of Java applications on mobile devices. Today, most mobile phones have a Java runtime environment embedded in the device. The overview of the layers of Java ME architecture for resource-constrained devices is shown in Fig. 1.

CLDC HI (Connected Limited Device Configuration Hotspot Implementation) is Sun's new high-performance Java virtual machine for embedded devices compared to KVM (K Virtual Machine), which in the past was widely deployed on Java ME [5]. CLDC (Connection Limited Device Configuration) defines the configuration for "constrained" devices with low memory, limited network connectivity, and user-interface capabilities. MIDP (Mobile Information Device Profile), on top of the configuration, provides rich APIs for the development of mobile applications, such as file handling, networking, and persistent storage. The MIDlet is the application designed with MIDP profile. The VM (Virtual Machine) on mobile devices provides a security model that is different from conventional Java. It does not control access through `SecurityManager` to enforce security policies as standard Java does. Java class files are properly pre-verified in a pre-verification process of the MIDP application's life cycle. Developers or testers are not allowed to design customized class loader during run-time, and

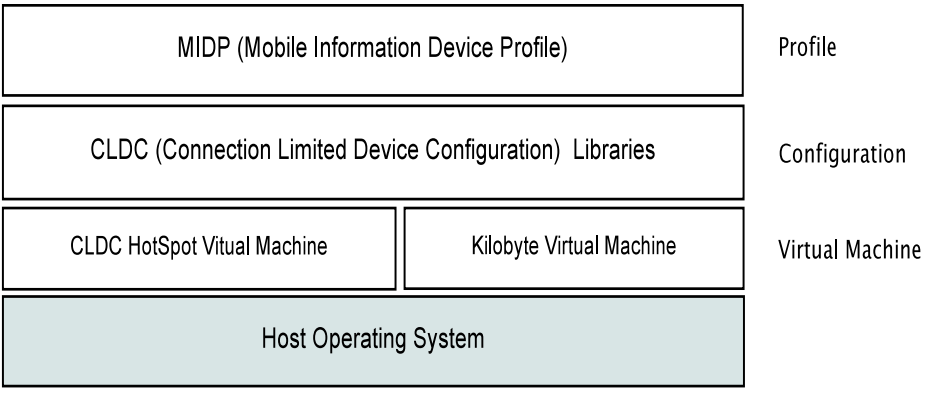

Fig. 1. Java ME Platform

it is also not allowed to download any new libraries containing native interfaces that are not supported in MIDP or CLDC.

From application-level considerations, the study of security vulnerabilities [1] reveals that the platform suffers from many problems caused by poorly written code, containing unauthorized method calls. As mobile devices hold very important information, it can be exploited easily as the devices roam. Applications running on Java ME platform are implemented with the core profile of MIDP. In MIDP 1.0 specification, the application is restricted into a sandbox model. The sharing between MIDlet suites is strictly prohibited. In MIDP 2.0, the access to sensitive resources such as method calls and sharing between MIDlet suites are allowed under granted permissions by the MIDlet. This can cause security problems by inappropriate usage of APIs. The program can access shared resources and facilities that contain sensitive method calls.

In MIDP 2.0, some low-level APIs should be protected from application developers. The application should not call such APIs via higher-level interfaces. For example, the class `RecordStoreFile` provides the direct interface of performing actions on data records in the storage system. In the transitional period of specification implementation, such APIs exist for the purpose of compatibility with old versions. However, with malicious designs, the data from other suites can be easily exploited by the methods provided in `RecordStoreFile` class [1]. Therefore, it is developers' responsibility to take care of all data protections or rely on the enhancement of MIDP security specifications.

Although manufacturers are implementing features complying with MIDP specifications, some conform to the 1.0 version, while some are already designing with the 2.0 version specification. Therefore, there still exists risks on devices because of the varieties of specifications. Meanwhile, MIDP specification has been enhanced with many security features [4]. Currently, the MIDP 2.0 is still the main profile for the implementations of applications on Java mobile devices. Considering these issues, we monitor related sensitive method calls in MIDP 2.0 applications.

2.2 Bytecode Instrumentation

Bytecode instrumentation is a technology used to manipulate and modify bytecode instructions with additional concerns integrated without impacting the original system behavior. It inserts user-defined classes or methods in the format of bytecode. Bytecode is a set of instructions that can be interpreted and executed by virtual machines. Once a Java file is compiled into the class file, the bytecode instructions in the class file can be transferred across the network to other platforms for executions. Java programs are compiled into a generic intermediate format called Java bytecode. A method in bytecode is a sequence of instructions. Each instruction consists of a one-byte operational code specifying the operation to be executed followed by one or more arguments. An instruction can be expressed in the following format [9]:

```
1 <index><opcode>[<operand1>[<operand2>...]][<comment>]
```

The `<index>` is the index of the opcode (operational code) of the instruction in the array that contains the bytes of Java virtual machine code for the method. It can be thought of a bytecode offset from the beginning of the method [9]. The following example is a simple constructor from a MIDlet application with a simple method function of `String.valueOf()`.

```
1 public MainMidlet() {
2     String number = String.valueOf("H");
3 }
```

The corresponding bytecode is shown as below. It consists of index and instruction code in each line. The opcode `aload_0 [this]` gets the reference of the class from the stack. The line 5 and line 7 are two instructions of method calls. The `invokespecial` invokes the method of an instance of class, `this`. The `invokestatic` invokes the static method of a class. Opcode `ldc` pushes the value of the String `'H'` onto the stack. The `astore_1` stores the value returned from the method call into the variable `number`.

```
1  // Method descriptor #10 ()V
2  // Stack: 1, Locals: 2
3  public MainMidlet();
4  0  aload_0 [this]
5  1  invokespecial javax.microedition.midlet.MIDlet() [12]
6  4  ldc <String "H"> [14]
7  6  invokestatic java.lang.String.valueOf(java.lang.Object) : java.lang.String [16]
8  9  astore_1 [number]
9  10 return
```

Bytecode instructions are checked by a bytecode verifier in JVM before loading to virtual machines. However, the bytecode verifier mainly checks the type safety in VM (Virtual Machine). It cannot predict the runtime behaviors of the instructions. Therefore, it is necessary to check additional properties of bytecode instructions and prevent them from imposing any harms on systems. Bytecode instrumentation uses the structural reflection to return the relevant information from the bytecode instructions. It is a good way to know what the instructions do at run-time.

Javassist is a tool that makes the manipulation of Java bytecode simple [3]. Javassist is a class library that deals with reading and writing Java bytecode. It takes great usage of Java reflection features to modify bytecode retrieved from class files. In Javassist, each Java class is represented by an abstract `CtClass` object. The `ClassPool` is the repository of all loaded classes. The typical process of the instrumentation by Javassist is as follows.

```
1 ClassPool pool = ClassPool.getDefault(); CtClass cc =
2 pool.get(targetClass);
3 //targetClass: String name of the class
4  ....
5 /* Modification Process */
6 cc.writeFile();
```

`ClassPool` represents the repository of all loaded classes. The `CtClass` represents the instance of the loaded class. All modifications on the class, such as inserting fields, changing method bodies, modifying constructors, are performed against the `CtClass` object. The method `writeFile()` commits the changes into the loaded `CtClass` object.

As the reflection features are not allowed on the Java ME Platform, the instrumentation is performed in Java standard virtual machine with MIDP libraries and verified before delivering to Java ME Platform.

2.3 Aspects

Aspect technology is used to separate crosscutting concerns from functionality modules. The most common cross-cutting concerns in a system is the logging mechanism. An *aspect* is a code snippet that scatters across multiple modules of the whole system. It aims to address crosscutting concerns by providing advices for systematic identification (*pointcut*), separation (*join point*), representation (*aspect*), and composition (*weaving*). A *join point* is a location where crosscutting concerns are attached to the original functionality. A *pointcut* is an expression language that specifies the patterns of locations where the code for cross-cutting concerns are required.

Crosscutting concerns are encapsulated into separate modules, known as *aspects*, so that security concerns can be developed separately from the existing programs [7]. Combining aspects and existing programs requires the *recompilation*

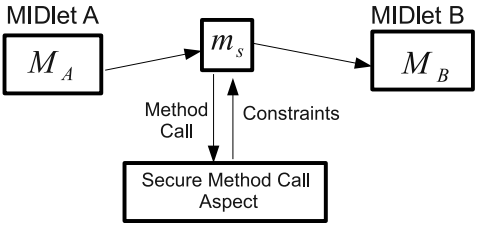

Fig. 2. Secure Method Call

process to form the final system. Using bytecode instrumentation, additional concerns and the original core program can be combined together without impacting the logical functionality of the original system.

3 Secure Method Call

We use aspect-oriented programming to track sensitive method calls during class loading. The join points are evaluated for the intercepted sensitive method calls. Currently, many existing approaches instrument security concerns by intercepting methods that use sensitive method calls in applications. However, a sensitive method call can be used everywhere by the application. The name or method signature can be assigned differently. Method context and class context should be known in advance to locate the secure method call. In our approach, the method context and class context are determined dynamically during runtime by the instrumented aspects. We insert tracking code directly around the method call without knowing which class or method will be using it in advance.

When a sensitive method call is requested, the appropriate checking is performed around the call. On both "caller" and "callee" sides, it should be verified whether the "caller" and the "callee" are in the granted permission of the request according to the policy by "callee" side. Secondly, the depth of method calls is evaluated to locate the origin of the method calls. In this way, we compare the chain of a method call with the specification to verify if the method call is in a safe state.

Let us consider a MIDlet suite containing two MIDlet applications, M_A and M_B. Assume that M_A retrieves the reference to one field of M_B, for example, the reference to the `record management system` in M_B. Then M_A can call method m_s, which is the sensitive method call as Fig. 2 shows.

The MIDlet M_A is performing actions defined by method m_s on MIDlet M_B. The M_A is the initiator of the method call. The sensitive method call invocation is intercepted by Javassist tool. The method call m_s is passed as a parameter to the secure method call aspect module. In the aspect module, the context of the method call is determined. The security state is evaluated by the constraints.

The security state of the method call is determined by the constraints returned from security state checking aspect. If the method call m_s passes the security

checks, the invocation is continued on the target M_B. Otherwise, the security warnings are prompted to user indicating failures of the security check. We encapsulate the "security state check" module into aspects. The aspects are instrumented around the sensitive method call. Bytecode instructions are instrumented into the context of the method call, rather than at the interface. We locate the context of the method call instead of changing method bodies, and we instrument concerns around the statement of the method call at the caller side.

The aspects are woven around the call and invoked when it is intercepted. At runtime, when the method call is invoked, the corresponding aspects are executed to check the security state of the call. The constraints returned from the aspect module decides whether to continue the method call. The security policies (permission check, domain check) are included in the aspects. The policies are returned as the constraints imposed on the execution of the method calls.

3.1 Bytecode Instrumentation Using Aspects

We design the corresponding aspects for concerns of sensitive method calls. The framework of the instrumentation process is shown in Fig. 3. The `ClassPool` is a repository representing all objects of the loaded classes. The `ClassLoader` is used to load classes from the pool into virtual machines. It takes the parameter of an instance of `ClassPool` to retrieve and manage properties of classes. The `Translator` is added to `ClassLoader` instructing how the classes from `ClassPool` should be loaded and when the instrumentation should be performed. It is implemented in the `onLoad()` method. The `onLoad()` method is invoked when the class is being read from `ClassPool`. The `ExprEditor` is the module containing aspect implementations. It implements how to modify the bytecode instructions when the `Translator` is going to instrument the class. In our framework, the target `Translator` performs instrumentation upon is the context (class and method) of the method call, rather than the body of the method call. As the custom classloader is not supported on Java ME platform, we use Java Standard Virtual Machine assisted with MIDP libraries to load the classes, manipulate the bytecodes, and instrument aspect codes. The modified bytecode are delivered to the mobile platform for final execution.

We use Javassist to encapsulate cross-cutting concerns. We design a subclass of `ExprEditor` class representing an `Aspect`. However, the description of an aspect library written with Javassist is not declarative but procedural whereas an aspect library in AspectJ is declaratively written with Java-like syntax [6].

A pointcut is defined by condition checks. In Javassist, it is specified by a function implemented in a class inherited from `ExprEditor` class: `edit(Expr e)`. The `edit()` method takes different parameter types as inputs including `FieldAccess`, `MethodCall`, and `ObjectConstruction`. For example, if we instrument on one field of an object, then the method `edit(FieldAcess f)` is defined to implement the condition checks on the field variables and specifies when the aspect is invoked. To design the pointcut with the combination of multiple types of fields, multiple `edit()` methods are defined in the class with different parameter types.

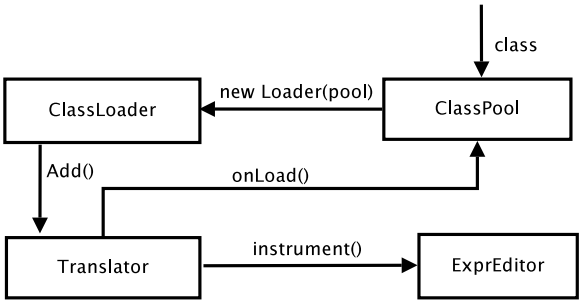

Fig. 3. Instrumentation Process

The join point is a process of determining where to instrument bytecode. Javassist uses Java reflection features to manage Java bytecode instructions. Unlike expressive patterns in AspectJ, the joinpoint determination has to be performed using procedural programming with reflection interfaces. After we intercept the method call, we determine the method and the class from which the method call is invoked. We further locate the position of the call statement within the method context and class context. Javassist instruments the statement when it is reached during loading.

The advice is the action performed at join points. We use Javassist to retrieve bytecode information and modify bytecode during loading them to the class loader. The advice is wrapped as a string of Java statements to replace the original statement. The original statement is the statement of executing the method call. The aspects are defined as separate method functions in `ExprEditor` class for modularity. All aspect codes are appended into a `StringBuffer` object, including the execution of the original method call.

3.2 Secure Method Call Aspect

We use the example shown in Fig. 2 to demonstrate the process of instrumentation in aspects at bytecode level. The method call is initiated from class M_A. If it invokes the `deleteRecord()` method call to access the object entity specified by `rs`, which is a `RecordStore` object in class M_B, the identity of the calling class M_A needs to be verified upon the call. In this case, the method call `deleteRecord()` needs to be intercepted. Class M_A is the class context containing the use of the method call. If the method call is used within a method in M_A, for example, `call()` method defined in class M_A, then `call()` is the method context of `deleteRecord()`. The context of the method call `deleteRecord()` is determined dynamically by aspects without knowing the name of context (class and method) in advance. This is different from other traditional approaches [7], in which the intrumentation actually happens in the body of the method call of `deleteRecord()`. Determing the context of a method call is to locate the position of statements of invoking the method call, that may be in the middle of the method context. The statement of the method call is then replaced with the instrumented code.

Pointcut. Sensitive method calls are intercepted statically by the Javassist tool during reading classes from `ClassPool`. The `Pointcut` specifies the interceptor patterns, indicating where sensitive method calls are happening. In our approach, we define the corresponding `edit(MethodCall m)` method to capture predefined sensitive method calls. When the method call `RecordStore.deleteRecord` is encountered, an `aspect` is instrumented into the context of the call. The `pointcut` is implemented by the method `edit()` in the class inherited from `ExprEditor`. The method function `edit()` contains the determination of point cut. It invokes the corresponding instrumentation containing join points and advice defined in `ExprEditor`. It takes different parameter types including `FieldAccess` and `MethodCall` to process different types of objects. If it targets on the type of method call, it then takes the `MethodCall` as the parameter to intercept method calls filtered by condition checks.

```
1  public void edit(MethodCall m) throws CannotCompileException{
2      if(matchMethod(m)){
3          try{ rmsInstrument(m);
4          }catch(Exception e){ e.printStackTrace();}
5      }
6  }
7
8  public boolean matchMethod(MethodCall m){
9      return (m.getClassName().equals(m_CName) && m.getMethodName().equals(
           m_MName));
10 }
```

The interception of the method call of `RecordStore.deleteRecord` is shown as below. The field member variable `mName` specifies the name of the method call. The method `matchMethod()` is a function to verify the signature of the method call. The variable `m_CName` refers to the class name defining the method call. In our example, it is `RecordStore`. The variable `m_MName` refers to the method name, which is `deleteRecord()`. The method `rmsInstrument()` is the aspect containing join points and advice to weave. It is defined in `ExprEditor`.

JoinPoint. The `JoinPoint` specifies the place where aspects are woven into. The process is to determine the context containing the statement of the intercepted method call. When the context is located, we instrument tracking aspects into it around the statement of the call. The method call is passed as a parameter to assess its context. If the intercepted method call is invoked from another method, then this calling method is the method context of the call, and the class that the calling method belongs to is the class context of the call. All the context information such as calling method, calling class, and calling MIDlet determine whether the caller is authorized for the call.

We first retrieve the method and the class containing this method call. Below is the function to retrieve the information of class context and the method context. The `CtClass` represents a class object, and the `CtBehavior` represents a block object that containts the use of `Expr` object.

```
public String getContextMethod(Expr expr){
    CtBehavior source = expr.where();
    return source.getName();
}

public String getContextClass(Expr expr){
    CtClass cc = expr.getEnclosingClass();
    return cc.getName();
}
```

After we get the class context and method context, we further locate the position of the statement of the method call. The advice is invoked as a method defined for the determined context, which is `CtMethod` shown in the example below:

```
final String className = getContextClass(m);
final String methodName = getContextMethod(m);
ClassPool cp = ClassPool.getDefault();
CtClass cc = cp.get(className);
CtMethod cm = cc.getDeclaredMethod(methodName);
cm.instrument(editor);
```

The `editor` is the "aspect" module instrumenting bytecode into the context of the method call `cm`. When reaching the method context `cm`, the `editor` further locates the position of the statement of method calls. The aspects defined in `editor` are instrumented when the statement is reached.

Advice. The `Advice` is the action performed as a response to the events of the sensitive method call. It is defined in instrumented code. In our example, it is encapsulated in the sub class of `ExprEditor`. After the class context and method context are determined, the next step is to construct the bytecode to weave into appropriate locations.

We design aspects around the statement of sensitive method calls appearing in their class and method context. We replace the method call statement by new bytecode containing logging concerns and the original statement. The new statements are encapsulated into `StringBuffer` object, including the executions of original statements represented by `$proceed()`.

```
    StringBuffer codes = new StringBuffer();
    codes.append("Log("before invoking the method call");");
    codes.append("$_ = $proceed();}");
    codes.append("Log("after invoking the method call");");
    m.replace(codes.toString());
```

Aspect Rule. To better refine the control flow of sensitive method calls, we adopt an aspect rule to control how the method call is granted access. The access rule is defined to confine the use of sensitive method calls that are risky to execute. The aspect rule is used to filter the class or the method to be instrumented.

```
1 <aop>
2     <pointcut name="rms" expr="execution(public * *.RecordStore->
          deleteRecord(int))"/>
3
4     <bind pointcut="rms">
5         <interceptor class="RmsEditor"/>
6     </bind>
7 </aop>
```

Each aspect is specified by an `aop` element. The `aop` element contains an element of `pointcut` and an element of `bind`. The `pointcut` specifies the pattern of the pointcut. It contains an attribute of `expr` that defines the expression of intercepted method call. In this example, it indicates an expression to intercept the method call of `RecordStore->deleteRecord(int)`. The method call can be initiated from any classes. The advice is defined in `bind` element. The `bind` element binds the `pointcut` and `interceptor`. The interceptor specifies the instrumentation class of `ExprEditor`. In this example, `RmsEditor` is a subclass of `ExprEditor` containing aspects implemented by its method `edit()`. When the method call specified by `pointcut` is captured, the advice specified by `interceptor` element, `RmsEditor` is invoked to instrument the method call.

4 Implementation

The instrumentation framework is written in Java. The UML diagram of the implementation is shown in Fig. 4. We use one MIDlet application (`RmsApp`) of RMS (Record Management Store) as a target application. The target application contains the usage of the sensitive method call of `deleteRecord()`. We instrument the designed aspects into the application around the sensitive method call for tracking purpose. The modified bytecode instructions run on an emulator device. Whenever a sensitive method call is invoked, the event is logged with the source of initiator. This approach provides a way to track and locate the source of malicious code on the device.

The `RmsTranslator` is constructed to control how class files are interpreted. The translator class is attached with the `RmsEditor` which contains the instrumentations in aspects. The `RmsSecureController` is the entrance point of the instrumentation process.

Class `RmsSecureController` contains three members. ClassLoader is a user-defined class loader which is used to load classes. `ClassPool` represents the repository of all objects read from class files. `ClassLoader` reads the classes from `ClassPool`, as we have shown the process in Fig. 3. `RmsTranslator` provides instructions to `ClassLoader` about how to interpret and load classes at runtime.

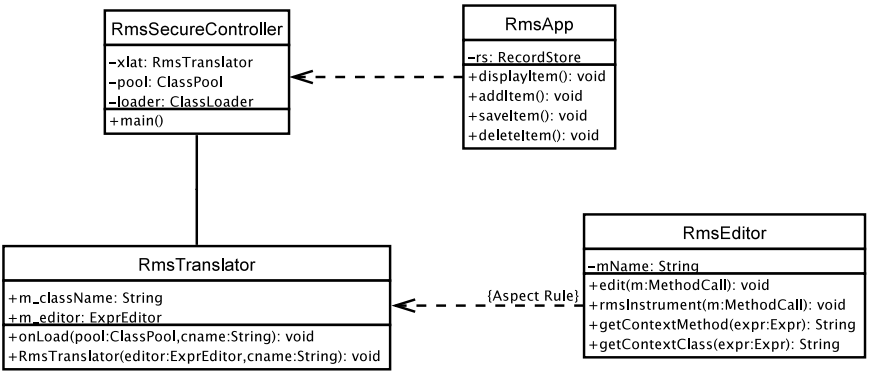

Fig. 4. UML of Instrumentation Class

Class `RmsTranslator` defines the `onLoad()` method. The method implements the function of loading the required method calls. It has two parameters, `ClassPool` object, and the string indicating patterns to match the required method call. The constructor of `RmsTranslator` has two member attributes: the class name variable indicating the target name to intercept and the `ExprEditor` specifying expressions of patterns for identifying join points.

Class `RmsEditor` encapsulates aspects into method implementations. The `edit()` method is designed to intercept method calls specified by its parameters. When the desired method call is intercepted, the aspects are invoked by `rmsInstrument` around the positions of statements. The positions of statements are calculated by `getContextMethod()` and `getContextClass()` to retrieve the context information including the class and method contexts.

We verify whether the instrumentation is successful in two phases. In the first phase, we check the bytecode instructions statically. In instrumentation process, modified bytecode are written into temporary files. Using `javap` program and the Eclipse IDE to disassemble the class files, the bytecode of `Log()` methods are instrumented successfully around the `deleteRecord()` method call. In the second phase, we conduct the run time checking. The modified bytecode is delivered to the emulator toolkit at run time. When a user invokes the `deleteRecord` operation, the logs are recorded specifying where the operation is initiated from and where it is reached. The bytecode is instrumented for tracking the source of the call.

Our instrumentation is conducted in Java Standard Environment and tested using Sun's Java Wireless Toolkit 2.5.2 for CLDC on Debian GNU/Linux operating system. During the instrumentation, we set the `bootclasspath` to Java ME library version to make it deliverable to mobile devices. As the instrumentation is performed in Java Standard Environment assisted with MIDP libraries, we use the `preverify` command tool provided by Java Wireless Toolkit to statically verify the validity of modified classes. Since the instrumented aspect code are only limited to APIs allowed only on Java ME platform, the classes pass through a pre-verification process.

5 Related Work

Static analysis technique is commonly used to assess the quality and the security of downloaded programs. The MATOS (Midlet Analysis TOol Suite) [10] automatically validates MIDlets with Point-to analysis. It analyzes properties of JAD (Java Application Descriptor) conformity, signature certificates, allowed usage of classes and methods, and acceptable ranges of argument values. Bian *et al.* [11] also perform bytecode analysis and combine the technique of dependency analysis with the information flow analysis to predict secure information flow from bytecode's composition. Avvenuti *et al.* [17] propose a security mechanism with the static analysis to indicate the state of secure information flow. All of these static analysis techniques are helpful in static checking phase, but they cannot address all the issues that may be found at runtime. In [19], a formal specification of dynamic semantics of Java bytecode is presented in the format of operational semantics for the Java Virtual Machine, giving each instruction a rule describing its effect based on the machine state. These approaches provide a static view of assessment of bytecode security. Our approach protects the use of sensitive method calls and tracks their behavior during runtime. It is more practical because most mobile applications are deployed in the format of bytecode.

Bytecode instrumentation has been used in the area of debugging and profiling. Binder *et al.* [14, 15, 16] propose a series of approaches of bytecode re-engineering by instrumenting Java's standard class libraries. The bytecode instrumentation has also been used to adapt the legacy software to distributed execution on multiple JVM [13] migrated from the version running on single JVM. However, it is not realistic to modify the implementation of system libraries, although source code is available for some systems. We cannot rely on platform's upgrade since it is difficult to recall and patch up all platforms currently in use. Our approach neither modifies the platform of applications, nor changes the original programs. We instrument the bytecode instructions while loading applications into virtual machine. The bytecode is modified during loading. After the execution is completed, the program remains as it was when downloaded.

Many tools assist in flexible bytecode instrumentation. BCEL [20] (ByteCode Engineering Library) provides libraries to manipulate low-level bytecode. It directly operates on the instructional operands. It requires proficient skills on bytecode. Win *et al.* [7] illustrate the effectiveness of instrumenting separate application security concerns using AspectJ. However, they either require the source code for instrumentation, or instrument bytecode instructions in a procedural way. Our approach uses Javassist to modularize concerns in aspects. Javassist operates on Java bytecode with flexible expressions. It combines the benefits from both BCEL [20] and AspectJ [7]. Hence, we can integrate more concerns flexibly. It does not require too much knowledge of low-level bytecode instructions or the access to source code.

The applet filter was proposed [12] by substituting the references of potential classes and methods. The authors provide a solution to change the references of restricted classes and methods stored in the `CONSTANT_POOL`, where the references

of all loaded method calls are stored. However, if there are more unsafe classes or methods, more subclass and method references should be designed. They also change the standard class libraries provided by JVM environment. It is not realistic to modify the platform. In our approach, we do not modify the platform and only focus on the application-level. We do not modify the implementation of a program, but weave the additional security concerns without changing the functionality of the original program.

6 Conclusion

Without the source code of applications, it is difficult to manipulate bytecode in a flexible way. We present an approach of bytecode instrumentation using aspects. The process does not require access to source code. We design corresponding aspects to track secure method calls by weaving it into executable applications without having too much knowledge about bytecode. The bytecode instrumentation is performed during class loading. Therefore, we do not need to change the original classes. The bytecode instructions are inserted before and after the intercepted method call successfully to track the call event. Determining the contexts of the sensitive method calls dynamically is another benefit of our approach, while existing approaches require the names of methods that use the sensitive method calls in advance.

In future, we will try to make aspects more flexible to instrument. Javassist is a procedure-based facility for bytecode instrumentation. The modularized aspects are still not quite flexible and obscure to understand. Therefore, the AOP features provided by Javassist is kind of limited. However, the principle of AOP for bytecode instrumentation is the same. Currently, we address only the security issues related to sensitive method calls. We plan to use more security rules to integrate more flexible security concerns.

Acknowledgments

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Debbabi, M., Saleh, M., Talhi, C., Zhioua, S.: Vulnerability Analysis of J2ME CLDC Security. US DoD Information Assurance Newsletter 9(2), 18–23 (2006)
2. Debbabi, M., Saleh, M., Zhioua, S.: Java for Mobile Devices: A Security Study. In: Proceedings of the Annual Computer Security Applications Conference, ACSAC 2005, Tucson, Arizona, USA. IEEE Press, Los Alamitos (2005)
3. Javassist, `http://www.csg.is.titech.ac.jp/~chiba/javassist/`
4. JSR 271: Mobile Information Device Profile 3, `http://jcp.org/en/jsr/detail?id=271`

5. Sun Java ME CLDC HotSpot Implementation White Paper, http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf
6. AspectJ Programming Guide, http://www.eclipse.org/aspectj/doc/released/proggui-de/index.html
7. Georg, G., Ray, I., France, R.: Using Aspects to Design a Secure System. In: 8th Int'l Conf. on Engineering of Complex Computer Systems, pp. 117–128 (2002)
8. Using Javassist for bytecode search and replace transformations, http://www.ibm.com/developerworks/java/library/j-dyn0302.html
9. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison Wesley, Reading (1999)
10. Crégut, P., Alvarado, C.: Improving the security of downloadable Java applications with static analysis. In: BYTECODE. ENTCS, vol. 141. Elsevier, Amsterdam (2005)
11. Bian, G., Nakayama, K., Kobayashi, Y., Maekawa, M.: Java Mobile Code Security by Bytecode Analysis. ECTI Transactions on Computer and Information Technology 1(1), 30–39 (2005)
12. Chander, A., Mitchell, J.C., Shin, I.: Mobile code security by Java bytecode instrumentation. In: DARPA Information Survivability Conference & Exposition (DISCEX II) (June 2001)
13. Tatsubori, M., Sasaki, T., Chiba, S., Itano, K.: A bytecode translator for distributed execution of legacy java software. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 236–255. Springer, Heidelberg (2001)
14. Binder, W., Roth, V.: Security Risks in Java-based Mobile Code System. Scalable Computing: Practice and Experience 7(4), 1–11 (2006); SWPS
15. Binder, W., Hulaas, J., Moret, P.: Advanced Java Bytecode Instrumentation. In: 5th International Conference on Principles and Practices of Programming in Java, Lisbon, Portugal, pp. 135–144 (2007)
16. Binder, W., Hulaas, J., Moret, P.: Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation. In: Seventh IEEE International Working Conference, September 30, pp. 91–100 (2007)
17. Avvenuti, M., Bernardeschi, C., De Francesco, N.: Java bytecode verification for secure information flow. ACM SIGPLAN Notices 38(12) (December 2003)
18. Resource and Information Flow Security Requirements for MOBIUS (Mobility, Ubiquity and Security) (2006), http://mobius.inria.fr/twiki/pub/DeliverablesList/We-bHome/Deliv1-1.pdf
19. Bertelsen, P.: Dynamic semantics of Java bytecode. In: Workshop on Principles on Abstract Machines (September 1998)
20. The Byte Code Engineering Library (BCEL) manual, http://jakarta.apache.org/bcel/manual.html