Aspect-Oriented Programming and security

Aspect-Oriented Programming (AOP) is a new type of programming that quickly attracts IT developers. One of the common reasons is that because it is branched out from the popularity of Java Spring framework, people begin to understand the key benefits that AOP brings to application development. In this article, u & uac

Rohit Sethi

Aspect-Oriented Programming (AOP) is a new type of programming that quickly attracts IT developers. One of the common reasons is that because it is branched out from the popularity of Java Spring framework, people begin to understand the key benefits that AOP brings to application development. In this article we want to introduce people to AOP. First of all let's ask what AOP is.

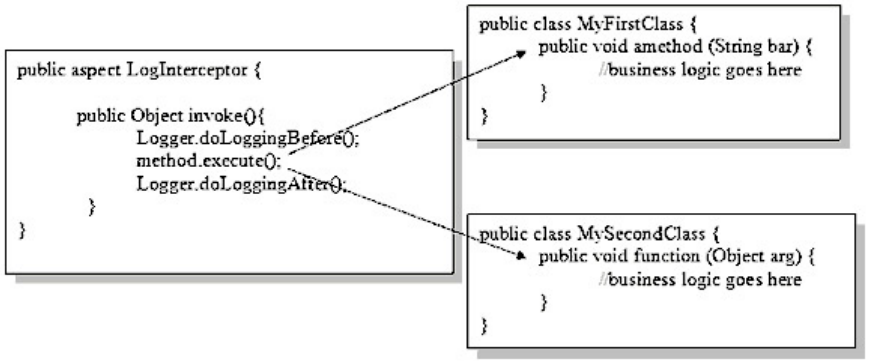
What is AOP?

Many people don't really understand what AOP is and think that AOP is an alternative to object-oriented programming language (OOP), so we need to explain it in depth. This concept is obviously completely wrong: AOP is based on OOP. It focuses on cross-cutting concepts or aspects - common code for different objects. Using an AOP language (such as AspectJ) or libraries (like Spring), programmers can write code for this function, then define the location to weave it into existing objects. An example needs to be given to clarify this issue to you. Suppose we have the following Java classes:

```
public class MyFirstClass {
        public void amethod (String bar) {
                Logger.doLoggingBefore();
                //business logic goes here
                Logger.doLoggingAfter();
        }
}
```

```
public class MySecondClass {
        public void function (Object arg) {
                Logger.doLoggingBefore();
                //business logic goes here
                Logger.doLoggingAfter();
        }
}
```

Here we can see the logging function being duplicated into two different classes. With AOP, we do the same thing (note that the syntax here has been simplified to make the roots clear):



Now we create a LogInterceptor aspect that can be inserted before and after any call to the class (ignoring the details of how the interleaving happens). We need to create a proxy for the call. The important thing to note about this example is that both MyFirstClass and MySecondClass are unknown and independent of LogInterceptor. In a real application, logging code can be copied into hundreds of different class files, so concentrating it like that will significantly reduce the number of source code. This can be done here because we can add security aspects to the application without changing the pre-existing code.

Security suggestions

Some cross-cutting concepts related to security are scattered through application logic:

1. Logging
2. Access control
3. L?i handling
4. Transaction management
5. Session management (in some cases)
6. Input / output validation

Using AOP, we can isolate a large chunk of concepts out of a large piece of code and assemble them. There are some cases where you will not be able to assemble (such as error management), but larger, AOP allows developers to gather on Plain Old Java Object (or whatever language you're using). POJO is essentially a code of developers who have learned to write from the beginning; This code only focuses on logic and there are no other concepts such as input validation, logging, access control, and error management. The more obvious the code, the less likely it is to fail (including security errors).

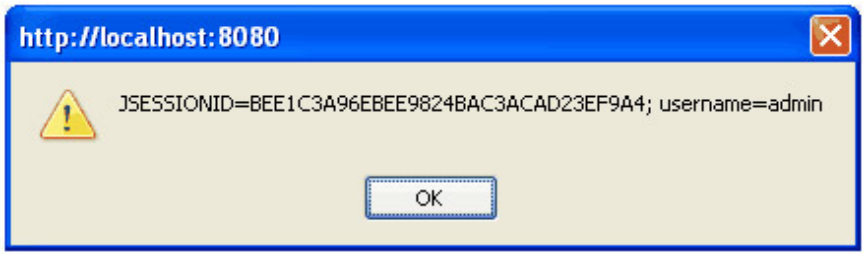
Another implication of this is that expert experts may be responsible for coding security aspects. These are people who have been specially trained and experienced in the field such as session management or access control, can manage the largest part of the code in an entire project or component. of project.

Real example: Add input validation for a previous application

Obviously, using AOP with its full potential on pre-existing applications will need to almost rewrite the application - which is often not feasible. The interesting thing about the security community is how to use AOP in an existing code base for additional security functions.

Let's take a look at an existing Customer Relationship Management (CRM) application, which has some XSS vulnerabilities (Cross-site Scripting). When entering an input the new goods one of the web forms with description.

We get:



This form is an XSS vulnerability. We know that some parts of the code are involved in this vulnerability. Therefore, we decided to protect the NewLead object in the enterprise class by adding input validation at any time that the setter method on the String property is called. This is a small slice of the NewLead class that has several setter methods:

```
public void setSalesStage(String SalesStage) {
  this.SalesStage = SalesStage;
}

public void setCampaignType(String CampaignType) {
  this.CampaignType = CampaignType;
}

public void setName(String Name) {
  this.Name = Name;
}

public void setCategory(String Category) {
  this.Category = Category;
}

public void setProductsOfInterest(String ProductsOfInterest) {
  this.ProductsOfInterest = ProductsOfInterest;
}

public void setDescription(String Description) {
  this.Description = Description;
}
```

We created the following code in the AspectJ programming language, which is easily integrated into existing Java applications - to reduce XSS in the application. You don't need to worry if they look a bit messy. We will explain the example in detail:

```
public aspect InputValidator {
    private Pattern pattern;
    private final String patternString = "[a-zA-Z0-9@:\\/$.]*";

    pointcut myPointcut(String argument): execution(* NewLeadBean.set*(String))
        && args(argument);

    before (String argument): myPointcut(argument){
        Matcher matcher;
        if (pattern == null)
            pattern = Pattern.compile(patternString);
        if (argument == null)
            return;
        matcher = pattern.matcher(argument);
        if (!matcher.matches())
            throw new IllegalArgumentException("application security attack thwarted!");
    }
}
```

public aspect InputValidator-

An Aspect in AspectJ is almost like a class. It may have case variables and internal methods. There are some key differences here: pointcut and instructions.

mycutcut pointcut (String argument): execution (* NewLeadBean.set * (String)) && args (argument) -

The pointcut in AOP defines where we want the aspects to interact with the rest of the code. In this case, pointcut **myPoint ()** will be invoked whenever someone calls **setName ()** , **setSalesStage ()** . from the **NewLeadBean** class. The syntax **NewLead.set * (String)** is intuitive: any method that is part of the **NewLead** class starts with "set", and has a separate string parameter.

&& args (argument) means that we want to save the string parameters to a variable named argument (ie if someone calls **myNewLeadObject.setName ("rohit")** then **rohit** will be saved to the argument variable). Here pointcuts can be a bit complicated.

before (String argument): myPointcut (argument)

This is what we call 'instructions'. I mean, before a certain call of **myPointcut** (that is, before **NewLead.set * ()** is called), execution follows the code. Note that we can distinguish after each round (ie both before and after)

each call of a pointcut. The argument is a string passed as a parameter. This example demonstrates that we are taking a string argument (in our case "rohit") and checking whether it passes whitelist validation by the regular expression checker. If we have entered that line as the name of the merchant input, the next page of the screen will look like this:

This indicates that the aspect has successfully stopped attacking.

In addition, there are some better issues that you will have part of the central code that can be changed easily. Suppose that when a white list is discovered in a XSS form, it is possible to adjust this list in place and automatically populate it. However, if you find that other parts of the code also have XSS vulnerabilities, you can simply define additional pointscut, fast or slow, depending on the attacks. In addition, if you discover that there are a number of whitelists needed in your application, you can define regular regular expressions within the same aspect.

There are several structures that increase the use of AOP for security. For example, one of the biggest challenges with input validation today is that there are too many interfaces in a complex application. Many large enterprise applications have multiple single HTML web interfaces, each single Web application can have entries through:

1. Web services
2. Client diversity (eg applications, applets)
3. Enterprise-Java Beans
4. CORBA connections
5. Other middle floor connections

Building the issue of input validation control in such a interface does not prevent attacks when entered into another interface (such as Web services). At the same time, attacks like cross-site scripting only affect certain interfaces (such as Web browsers), not all (for example, rich applications). With AOP, you can define prevention as regular expressions in aspects, then define where they will be added via pointcut. This means that some prevention can be pointed to the interface code, while other methods point to business logic (as we have done in the NewLead example).

Add details and impact on performance

How exactly does AOP work? That depends on the platform you are using. We have discussed AspectJ here for a number of specific reasons:

• Very easy to integrate AspectJ into existing Java applications. AspectJ simply adds Aspect to Java binaries by using a technique called "weaving", essentially changing the compiled code to add advice at specific pointscut. At runtime for Web applications you just need to add the library **aspectjrt.jar** to the **WEB-INFlib** in your project.

• If you are using Eclipse Integrated Development Environment (IDE), it is perfectly normal to convert an existing Java application to AspectJ with AspectJ Development Tools.

• Since AspectJ works at compile time, it is very efficient and does not suffer from adverse effects on performance.

As we mentioned at the beginning of this article, Spring Framework also has AOP support. There are some major differences in design with Spring AOP and AspectJ, most notably Spring working on top of Java and not requiring a different compiler. Spring AOP is designed for Spring-controlled beans, meaning integrating it into an existing non-Spring application can be difficult. That said, Spring's AOP is still very valuable when AspectJ is not feasible or in applications that use Spring. Spring developers intentionally provided support to integrate Spring's Inversion of Control capability with AspectJ's AOP power.

What is next?

The ability to use AOP in application security is huge. There are a number of applications that AOP is used for security:

• Execute access control independently of the application logic. Instead of having clear checks like **checkAccess (User)** in each sensitive function, you can accomplish this through aspects and allow developers to focus on business logic.

• Implement application security policies such as prohibiting programmers from calling dynamic SQL libraries (eg **executeQuery ()**). Whenever that library is called, you can use aspect to include an exception and correctly record the place where the dangerous calls appear.

Software developers are turning to voting for AOP. JBoss, WebSphere, and WebLogic have integrated AOP or have future implementation announcements. Now the application security community needs to follow a complete set by providing instructions on how AOP can be used in security.

We can do this by making sure to add AOP to the application security training program, besides conducting more research on how AOP works in securing applications. used in production (including benchmarking for performance impact) and provides more examples of code for developers.