A Posteriori Defensive Programming: an Annotation Toolkit for DoS-resistant Component-Based Architectures

Valerio Schiavoni
valerio.schiavoni@inrialpes.fr

Vivien Quémé
vivien.quema@inrialpes.fr

ABSTRACT

Denial-of-Service (DoS) attacks are a major concern for modern distributed applications. They exploit weakness in the software in order to make it unavailable to well-behaved users. Building DoS resistant software is still an issue. Solutions relying on the use of annotations have been proposed. Nevertheless, they require modifying the source code of the application, and must thus be applied at design time. In this paper, we propose an annotation toolkit that allows building DoS resistant component-based systems. The solution we propose does not require any modification of the source code of the application. Moreover it can be applied at deployment time. Its implementation relies on the use of Aspect-Oriented Programming techniques together with Java 1.5 annotations.

Keywords
DoS attacks, components, aspect-oriented programming, annotations

1. INTRODUCTION

Denial-of-Service (DoS) attacks make a system unavailable to well-behaved users by consuming a disproportionate amount of resources on the target system. DoS typically exploit weakness in the software, for instance consuming the bandwidth of the victim’s network or overloading the computational resources of its system.

A lot of factors are involved when trying to build DoS resistant software. To make things worse, most of them are typically unknown during the development of the application. Reasonable hypotheses can facilitate the prevention of such attacks, but new ones can always arise and be exploited to damage systems.

Researchers have studied the nature of these attacks [11, 7] and possible defense methodologies [12, 10], but building DoS resistant software still remains a challenging task. Building robust software can be achieved by adopting different techniques. For instance, in procedural or object-oriented applications, a possibility is to write ad-hoc functions to constantly monitor the application’s execution flow. A different approach has been proposed in [10]: the source code of the application is annotated with macros by experienced programmers. Then, during the program’s execution, macros control how a function is used and, in case of abuse, deflect the execution path to some other function.

Placing the annotations in the right point is a sensible task, on which all the rest of the system relies. Moreover, this is the only possibility to spot out potential weaknesses in the program.

In this paper, we propose an improvement over this low level approach. We focus on component-based systems and show, in this context, how it is possible to provide a general mechanism to detect DoS attacks. The proposed mechanism is able to handle the robustness concern as a separated and modularized but yet integrated aspect of the system. In typical component based applications, each component exposes a service. The key idea is to annotate such services and use the annotations as a mean to detect an attack. For modularizing and separating the implementation of annotation processing, we make use of Aspect-Oriented Programming techniques [6]. We argue that the annotation step should be as transparent as possible to the developer of the base components. As a consequence, in the solution we propose, the developer work only at the model of the system. We avoid the developer the burden to scan line by line the application source code to find out potential deficiencies. We also stress the fact that annotations can be applied after the design and the implementation of a component, and without modifying neither of them. In such perspective, we call our approach a posteriori.

The paper is structured as follow. Section 2 explains what defensive programming is, and what is the nature of detectable DoS attacks. Section 3 illustrates our intentions and a possible use case. In Section 4 we show the initial implementation of the toolkit, and a possible application to the above usecase. Related works and conclusions complete the paper.

2. BACKGROUND

A number of different techniques exist nowadays to perform a DoS attack. It is possible to characterize two distinct kinds of resources that could be attacked: renewable resources, which include CPU cycles, network bandwidth, disk space usage, and bus usage; and non-renewable resources, such as the number of running processes, the available hardware ports, the CPU registries, PCBs. These two
families of resources are victims, respectively, of two different kinds of attack. Renewable resources can be attacked by flooding them with massive number of requests: this kind of attacks are called busy-attacks. Non-renewable resources can be attacked by a malicious user who claims them and never release them. This kind of attack is called claims-and-hold [10].

Both these attacks will eventually cause a degradation of services, and possibly a total blackout of the system.

The different nature of these attacks lead to different defensive strategies. For instance, when a busy attack is performed against a component of the system, a good defensive strategy is to isolate the component from the rest of the system. On the other hand, if an attacker is using a claim-and-hold technique, a possible defensive strategy is to monitor clients to detect those that do not making any progress in their actions.

3. CONCEPTS

As explained in the introduction, this paper presents a general mechanism to detect DoS attacks in component-based systems. A component-based system is made of an assembly of components interacting through bindings. Bindings are established between components requesting a service and components providing a service. Component-based systems have two characteristic features that ease the detection of DoS attacks. First, it is easy by looking at bindings between components to distinguish between components that provide resources and components that act as resource consumers. Such distinction helps in identifying the components to be defended. Second, component architectures ease the isolation of services, which we mentioned as a typical defensive strategy against busy-attacks.

Recall that our goal is to provide a mechanism that be as transparent as possible from the developer point of view. We consider that this goal is reached if it is possible to plug the mechanism inside a system without having to modify its source code and its design. In the rest of this section, we explain the concepts of our mechanism on a concrete example. The example system, depicted in Figure 1, is a component-based HTTP server\(^1\). Components are depicted by boxes\(^2\). Provided services (also called server interface) and required services (also called client interfaces) are both represented by T-like structures. Bindings are represented by arrows going from client to server interfaces.

Consider the issue of monitoring the current workload at the Receiver component (on the left in Figure 1). Without other options, developers should write ad-hoc functions to limit the maximum number of requests, and integrate these functions into the implementation of the Receiver component. A side-effect of receiving a lot of requests could be discovered looking at the activity of the Logger component (on the right corner in Figure 1). In our example, its default behaviour is to write on a log file. In case of busy-attacks, such log file would grow possibly over some limit imposed by the system. In such unlikely situation, the Logger would stop working, causing the logging service of the system to be available (not to mention that if not properly designed, the whole application would stop working). Once again, an ad-hoc mechanism is necessary for the Logger component.

In the general case, for any of those situations where there's a risk of breaking some rules imposed by a system, there's also a risk of being victim of an attack, and a specific intervention from the developers is needed. We argue that such ad-hoc mechanisms can be replaced by general and so reusable ones.

Our proposal is to use annotations at the design level. Annotations are metadata that allow expressing a semantics about a given object or a given component. Annotations can be used to mark the interfaces of components. We call overlay of components a set of components marked by the same annotation. A component can obviously belong to several overlays. A defensive strategy can be applied on the overlay. All the services belonging to an overlay are thus protected by the same defensive strategy. A defensive strategy is implemented by an entity called annotation consumer, whose role is to detect which semantics has to be applied to the components annotated with the annotation it is in charge of.

Relatively to the HTTP server example, our toolkit allows the designer to mark the server interface of the Logger component as being member of an overlay of components. Then, when an annotation consumer is deployed together with the rest of the application, it will monitor the activity of this component, and also the activity of all other components in the same overlay.

Note that once an overlay of components has been defined, the annotation consumer implementing the policy to deploy in it is completely independent from it. Moreover, note that contrary to other approaches based on annotations, the application deployer only has to annotate each service. This effort is greatly simplified by the fact that in a component-based architecture, required and provided services are made explicit.

4. IMPLEMENTATION

This section describes the implementation of the proposed solution. We first describe an extension to an Architecture Description Language that allows annotating component interfaces. Then, we show how Aspect-Oriented Programming techniques can be used to easily develop annotation consumers.

\(^1\) Source code for the Comanche HTTP server is available at \url{http://fractal.objectweb.org}.

\(^2\) Components encapsulating other components are called composite components. Others are called primitive components.
4.1 Annotating component interfaces

The annotation toolkit we propose has been implemented within Fractal [2, 1], a Java-based component model. Fractal provides an Architecture Description Language (ADL) [4] allowing the description of component configurations.

Listing 1 gives the description of the Receiver component described in section 3. This description contains the definition of the component interfaces and the definition of the component content (i.e. the class implementing the component).

```xml
<definition name="Receiver">
  <!-- server interfaces -->
  <interface name="r">
    signature="Runnable"
    role="server"
  </interface>
  <!-- client interfaces omitted -->
  ...
  <!-- component content -->
  <content class="RequestReceiver"/>
</definition>
```

Listing 1: ADL definition of the Receiver component

We have extended Fractal ADL to allow annotating component interfaces. This extension consists in adding an annotatedby attribute to the interface element. This attribute is used to define which annotation has to be used for that interface. A straightforward usage of this attribute is to mark an interface as being part of those to protect against an attack. Consider listing 2: lines 6 shows how to use the introduced attribute to extend the definition of a server interface; it is specified that interface r should be annotated with annotation InBufferOverflowOverlay.

```xml
<definition name="Receiver">
  <!-- server interfaces -->
  <interface name="r">
    signature="Runnable"
    role="server"
    annotatedby="InBufferOverflowOverlay"/
  </interface>
  <!-- omitting the rest -->
</definition>
```

Listing 2: Receiver.fractal

The InBufferOverflowOverlay annotation is depicted in figure 3. Java 1.5 annotations are first class elements, themselves annotateable with other annotations, called meta-annotations. Through meta annotations, it is possible to set two parameters:

- the target parameter is used to specify at which granularity a given annotation can be used. Possible values are methods, method’s parameters, class or interface declaration, etc. Listing 3 at line 2 specifies that the granularity is at class/interface declaration;
- the retention policy indicates how long the annotation with the annotated types have to be retained. We need an annotation to be readable at runtime: it is possible to achieve it as shown in Line 3;

```java
//@import statements...
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InBufferOverflowOverlay {
}
```

Listing 3: The InBufferOverflowOverlay annotation

The ADL file is parsed by a factory that produces components whose interfaces are annotated using annotations specified in the ADL description.

4.2 Consuming annotations

Annotation consumers are developed using Aspect-Oriented Programming (AOP) techniques. We start this section by a brief presentation of AOP. Then, we present how this programming technique can be used to develop annotation consumers. Finally, we describe the deployment of aspects within components.

4.2.1 Background on AOP and AspectJ

Aspect-Oriented Programming (AOP) aims to handle cross-cutting concerns in software applications. These concerns typically affect several classes or modules; as a consequence, their implementation is not well localized and modularized.

AOP allows the implementation of crosscutting concerns in modular and well-localized entities, called aspects. An aspect is made of dedicated constructs which mirror well-defined points both in the program flow and in the program structure (e.g. where a method is invoked, when an exception is caught). This model is the so-called joinpoint model. Briefly, a pointcut construct lets you pick out join points that match certain criteria, and an advice construct lets you add code that should be executed at matching join points.

AspectJ [5], an aspect-oriented extension to the Java programming language, provides an implementation of the above constructs that allows expressing crosscutting concerns in a straightforward way. The typical AOP example is about logging. Consider listing 4: the Logging aspect matches against all calls on the open method made on instances of the PersistenceManager class. The before and after advices specify that meaningful messages must be logged before and after calls to the open method.

```java
aspect Logging {
  pointcut pmCall():
    call(void PersistenceManager.open(*));
  before pmCall(): {
    log("Opening a connection to the db");
  }
  after pmCall(): {
    log("Connection to the db opened");
  }
}
```

Listing 4: Example of a Logging aspect
4.2.2 Developing annotation consumers using AspectJ

Annotation consumers are developed using aspects. They make use of one of the latest features introduced in the AspectJ language: the possibility to declare joinpoints based on the presence of annotations in given objects. Consider the example in Listing 5.

```
1 aspect BufferOverflowOverlayManager {
2   pointcut underAttack():
3   execution (* (@InBufferOverflowOverlay *)) +.*(..);
4   before(): potentialVictim()
5   declare warning: annotatedResourceInvocation();
6   potentialVictim underAttack();
7   (...)}
```

Listing 5: Example of a Logging aspect using annotations

The scope of the aspect implemented in Listing 4 is to capture the execution of methods on instances of any classes annotated by the annotation InBufferOverflowOverlay. The definition of the potentialVictim named pointcut at Line 4 is used to capture the execution of methods on annotated classes. A potential victim is one of the interfaces marked by the annotation, considered to be attackable by malicious users. If such interfaces are used to attack the system, we expect to get an unpredictable usage of the services exposed by that interface: such behavior will be caught by the advice specified in Line 5. The aspect using an annotation to define an interesting point in the execution flow of a program is called annotation consumer.

4.2.3 Deploying aspects and components

Once aspects implementing annotation consumers have been developed, the only task that remains to be done is the deployment of aspects and components. The Fractal component model provides a runtime environment that allows creating components from their ADL definition. Because part of the code of Fractal components is dynamically generated, it is not possible to weave aspects and components source code. To solve this problem, we made use of the load-time weaving mechanisms [8] introduced in AspectJ 5.0. Figure 2 illustrates the integration of AspectJ load-time weaving mechanism with Fractal runtime environment. Components are generated by Fractal, and their binary form is made available to the JVM (upper boxes in Figure 2). Aspects are compiled by the AspectJ default compiler (AJC) (lower boxes). Then, an alternative class loader provided with the AspectJ API is able to instrument the class files of the components accordingly with the declared pointcut and advices of the just compiled aspects. The woven binaries are finally loaded into the JVM class repository.

5. RELATED WORKS

There are two kinds of works related to the work presented in this paper: those about defensive programming and those about the integration of AOP with component-based programming.

Defensive Programming. [10] proposes the use of annotations for building DoS resistant software. The solution they describe consists in annotating the source code of the application with macro functions that are used to monitor and control the execution of the application at runtime. By focusing on component-based systems, our proposition can easily been integrated with existing applications and does not require the source code of the defended system to be modified. They introduce the concepts of sensors to detect anomalies and actuators to change the control flow of a program when an anomaly is detected. There is an implicit relation between the concepts of sensors and actuators and those of pointcuts and advices. Whenever it is required to place a sensor into the source code, our approach permits the same operation without modifying it. Moreover, note that our proposition can be used a posteriori, i.e. it is possible to dynamically add DoS detection to an already deployed application.

AOP and Component-Based Programming. Several research works have been conducted on the integration of AOP with component-based programming. None of these works has been applied to the construction of DoS resistant software. Nevertheless, we cite them and compare the combined use they do of AOP and components to the one we propose.

[3] presents a programming model to implement non-functional services using aspects that are bound to components using Java 1.5 annotations. Doing so, they guarantee that the annotated components expose a given behaviour. Compared to our work, we use annotations to create overlays of components to easily deploy defensive strategies over them. In fact, the members of the overlay will be subjects to the same policy implemented using a specific aspect.

In [13], AOP is applied on the JavaBean component model. Compared to our work, they choose to extend the underlying component model, introducing a new “aspect-enabled” component model backward compatible to the JavaBean component model. Instead, we didn’t extend the Fractal model. We just had to modify its runtime and ADL in order to allow deploying components together with aspects.

In [9], an extension to the Fractal component model is presented, allowing some kind of AOP support for crosscutting concerns. Compared to our work, they allow weaving aspects through bindings to special components. Instead, we didn’t extend the Fractal component model, we exploit the standard weaver provided with the adopted framework, and implement crosscutting concerns striving for obliviousness.
6. FUTURE WORKS
We are in the middle of the process of testing the proposed solution against more DoS attacks (syn-flood attacks, Xmas-tree attacks, etc). We are also studying the relation between aspect-based and interceptor-based solutions to face DoS-attacks.

7. CONCLUSION
Denial-of-Service (DoS) attacks are a major concern for networked applications. Indeed, DoS attacks exploit weakness in the software, thus making systems unavailable to well-behaved users. Researchers have proposed the use of annotations for building DoS resistant software. The source code of the application is annotated with macros that control the system usage. Drawbacks of this approach is that it requires modifying the source code of the application and it needs to be done at design time. In this paper, we have proposed an improvement over this low level approach. We have shown that by focusing on component-based systems, it was possible to provide a general mechanism to detect DoS-attacks. The proposed mechanism is able to handle the robustness concern as a separated and modularized but yet integrated aspect of the system.

Our solution has two major advantages: first, it does not require to modify the source code of the defended system. Second, it can be applied at deployment time. Our solution relies on the use of Aspect-Oriented Programming techniques together with Java 1.5 annotations. We have implemented it within Fractal, a Java-based component model available at http://fractal.objectweb.org.

8. REFERENCES