Eavesdropping Techniques

13

IN THIS CHAPTER

"You can observe a lot by just watching."

Berra's First Law

Eavesdropping Defined

The previous chapters focused mostly on working with the application bytecode and resources. N-tier applications that are dominant on the server side offer additional angles for reverse engineering and hacking. It is common practice to deploy application tiers as separate processes that communicate with each other via network protocols. For instance, a Web browser displaying an HTML front-end on a user workstation uses the Hypertext Transfer Protocol (HTTP) to communicate with the Web server. In turn, the Web server typically uses RMI or IIOP to communicate with the application server. The application server relies on JDBC to communicate with the database. A classic deployment diagram for N-tier distributed Java applications is shown in Figure 13.1.

This chapter presents several techniques that can be employed to eavesdrop on the conversation between the distributed tiers. *Eavesdropping* is intercepting and logging the message exchange between a client and server. It can facilitate the troubleshooting or performance tuning of a complex distributed system, as well as provide insight into the application design and communication principles.

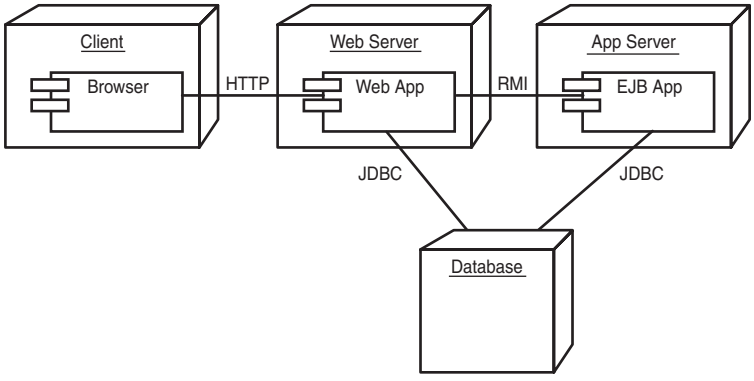

FIGURE 13.1 An N-tier application deployment diagram.

Eavesdropping on HTTP

We will first look at eavesdropping on Web-based applications and Web services that use HTTP as a transport protocol to communicate with their clients. HTTP messages consist of a header and the content that is sent across the network via TCP/IP. For a client to be capable of talking to a server, it must know the server's hostname or IP address and the port on which the server is listening. HTTP messages are sent in plain text, which can be easily read and understood by humans.

To eavesdrop on the client/server communication, you must intercept the message exchange. One way of doing this is by placing an intermediary that traces and tunnels HTTP messages between the client and server. An alternative is to monitor the communication on the network protocol layer, filtering on the messages exchanged by the client and server. We will look at both alternatives in the following sections. Once again, we will use WebCream as the server-side application because it provides static and dynamic HTML content via servlets and JSP pages.

Using a Tunnel to Capture the HTTP Message Exchange

A *tunnel* in this context is a pseudo server placed between the real client and real server to intercept and trace the message exchange. Instead of directly talking to the server, the client is reconfigured to send messages to the tunnel; the tunnel is configured to dispatch requests to the server on behalf of the client and forward the server responses back to the client. The tunnel logs the exchanged messages to the screen or into a file, allowing a hacker or developer to study the details of the conversation. The text-based and stateless nature of HTTP makes it a good choice for tunneling.

We will use the TCPMON utility distributed with Apache AXIS project for tunneling browser requests to WebCream's Tomcat server. Although TCPMON is by far not the most sophisticated tunneling software, it is free and does a good job for most cases. Download AXIS from Apache's Web site and, if you do not have a script to start TCPMON, copy the `CovertJava\bin\axis` subdirectory to the directory where you have installed AXIS. TCPMON takes in three command-line parameters, which are the port to listen on, the host, and the port number of the server to tunnel the requests to. Because WebCream runs Tomcat on port 8040, we will run TCPMON on port 8040 and tell it to forward to port 8040 of localhost.

We need to make a few simple configuration changes in WebCream to support tunneling through AXIS. WebCream generates page forms with fully qualified URLs for submission, which include the host and the port number. We want all traffic to go through the TCPMON, so we need WebCream to always submit forms to port 8000 rather than 8040 as it does by default. We can achieve this by adding the following two lines to `WebCreamconf\WebCreamDemo.properties` (see the WebCream documentation for more information):

```
html.docsURL=http://localhost:8040/webcream
html.submitURL=http://localhost:8000/webcream/apps/WebCreamDemo
```

Start WebCream's Tomcat using `WebCream\bin\startServer.bat`. Open your favorite Web browser and type the following in the address line:

```
http://localhost:8000/webcream/apps/WebCreamDemo
```

Notice that we are using port 8000, which is the port on which TCPMON is running, rather than pointing the browser directly to Tomcat, which is running on port 8040.

The browser should display the WebCream demo main page. Switch to TCPMON and make sure that you see the intercepted request in the top (or left) panel. TCPMON can be rather slow at detecting that the request is fully transmitted, so be sure to wait until the status of the request in the top panel changes to `Done`. Go back to the browser and click the Login Dialog button to open the next page; then enter **`Neo`** as the username and **`Wakeup`** as the password and click OK. (You are free to pick your own username and password, however.) If you switch to the TCPMON screen, it should look similar to Figure 13.2.

We can now examine the requests and the data intercepted by TCPMON. The top panel shows three messages, which corresponds to the number of pages requested by the browser. The most interesting piece of information is shown in the bottom-left and bottom-right panels. The left panel shows the request header and data, whereas the right panel shows the response header and content. Looking at the various attributes and content elements gives us an insight into the data exchange between the browser and server. For example, selecting the third request in the top panel and looking at the request data, we can see the actual values of the login name (`Neo`) and password (`Wakeup`). In the header, we can see a cookie, `JSESSIONID`, that can be exploited to hijack the user session.

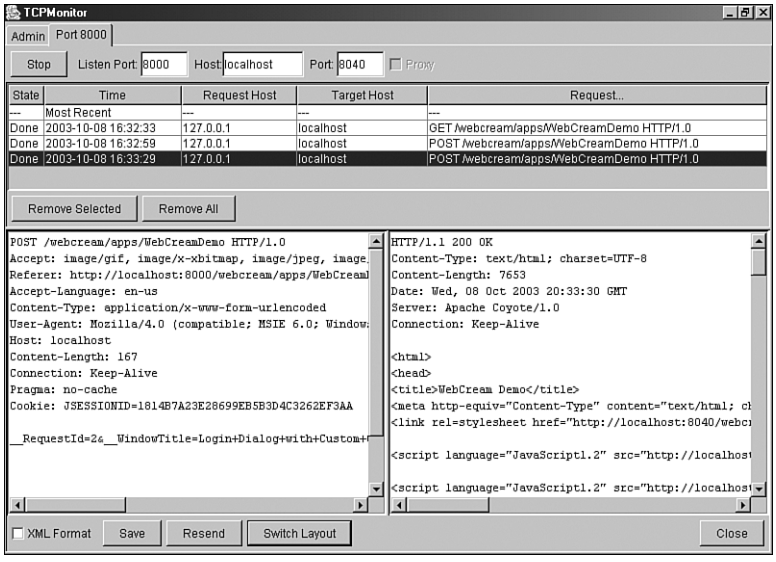

FIGURE 13.2 TCPMON showing intercepted requests.

Using a Network Sniffer to Capture the HTTP Message Exchange

Tunneling is a simple and effective way to see what is actually transmitted between the client and server. It is good for debugging and studying Web-based applications, but it has the drawback of requiring a reconfiguration of the client (and possibly even the server, as we had to do with WebCream) to send requests to the tunneling agent instead of directly to the server. The second technique you will learn allows eavesdropping on the network protocol level, which does not have the limitations of tunneling.

All distributed communications go through a layer of protocols supported by the JVM, operating system, and network driver. Protocols are stacked on top of each other, meaning that higher-level protocols rely on lower-level protocols to perform more basic tasks. Thus, HTTP relies on TCP, which in turn relies on IP. A single HTTP message can be represented by several low-level IP packets, and it's the job of the networking layers to disassemble and then reassemble the packets. Most of the physical networks are composed of interconnected Ethernet segments of workstations. Within a segment, packets from one node are transmitted to all nodes regardless of the target node address. To communicate with a computer outside the segment, routers redirect the packets to other segments. The bottom line of this crude summary is that, when an application on one host communicates with a remote application

on a different host, the protocol packets have to traverse a number of other network hosts before reaching the target. Network sniffing and monitoring takes advantage of this principle to spy on the communications. Normally, a network node accepts only the packets that are targeted to it, ignoring all other packets. However, a node can be running in *promiscuous* mode in which it accepts all packets regardless of the destination address. Then an engineer or a hacker can examine the packets and their contents to gain insight into the application communications.

Working on the protocol level can be hard and time-consuming. Because HTTP is a very common protocol, products are available that simplify eavesdropping on HTTP communications. We are going to look at HTTP Sniffer from EffeTech, which is a Windows-based shareware application. We will try to perform the same task of intercepting the communication between the browser and WebCream's Tomcat and see whether we get a different result than with TCPMON. One of the drawbacks of using a network sniffer is that it does not work when the client and server are running on the same host. No packets are passed to the network driver, so the sniffer is incapable of capturing the protocol requests and responses. Thus, for this and the next section you need to make sure that the Tomcat server is running on a different network host from the browser. If you do not have access to another machine, you can use any Web site instead of WebCream.

Download and install EffeTech's HTTP Sniffer. It installs WinPcap, a packet capture library that is added as a device driver to Windows to capture raw data from the network card. Because WebCream is running Tomcat on port 8040, we need to add that port to the sniffer filter using the Filter item of the Sniffer menu. You might have to switch the current network adapter in the sniffer if the recording will not produce any results, but for now leave it as the default. Start WebCream's Tomcat on the remote server and then bring up the Web browser. Note that the sniffer can run on the same machine as the browser, on the same machine as the Web server, or on any machine connected to the same Ethernet segment as the browser or the server hosts. Start the recording using the Sniffer menu or the toolbar and type the server URL in the browser. If testing with WebCream, open the main screen, click the Login Dialog button to go to the next page, enter **`Neo`** as the username and **`Wakeup`** as the password; then click OK. After stopping the recording in HTTP Sniffer, its screen should look as shown in Figure 13.3.

The way the information is presented and the type of information intercepted is almost the same as in TCPMON. Besides sporting a cleaner user interface, HTTP Sniffer shows other resources that were obtained by the browser from the server, such as GIF and JavaScript files. Once again, looking at the request data we can obtain values of form parameters such as the username and password. The beauty of this approach is that we didn't have to do anything to the target application, yet we have captured a complete log of the communication between the client and server.

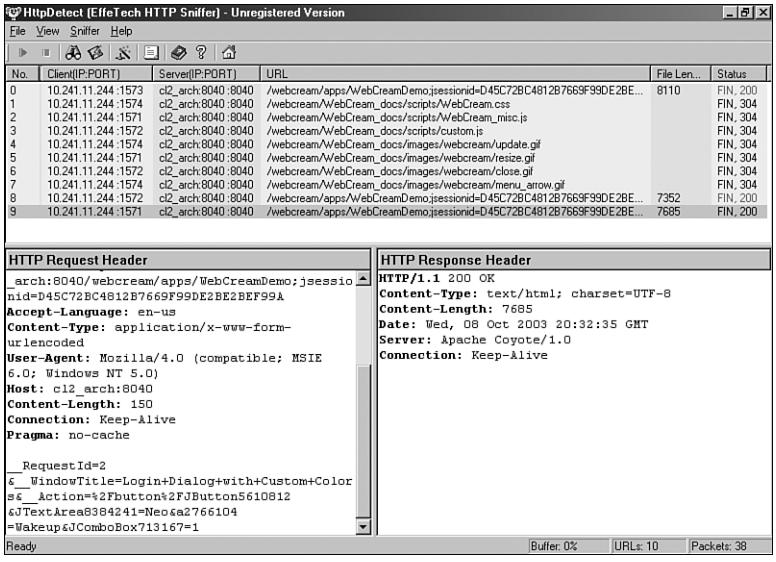

FIGURE 13.3 HTTP Sniffer showing intercepted requests.

Protecting Web Applications from Eavesdropping

In a short amount of time, you have learned how to eavesdrop on browser-based user interfaces. The ease with which you can obtain the values of potentially sensitive parameters is rather alarming. A few precautions can make the job of hacking a Web application much harder. The simplest and most effective way to protect the data integrity and secure the client/server communication is via the use of Hypertext Transfer Protocol Secure (HTTPS). This protocol mandates that a client establish a secure channel to the server using Secure Sockets Layer (SSL) before sending any HTTP data. After the channel is established, the message exchange occurs just as with HTTP. The benefit of HTTPS is that all data is encrypted, so even if someone intercepts a network packet, decrypting it is virtually impossible. SSL does have an overhead that can slow down the server, so for high-performance applications doing all communications via HTTPS, this might not be feasible. A good compromise is to use HTTPS selectively or to continue using HTTP but encrypt the sensitive content on the application tier. If the user interface is a Web browser, JavaScript functions can be used to perform encryption and decryption on the client side.

Eavesdropping on the RMI Protocol

Java Remote Method Invocation (JRMI) uses either Java-specific Java Remote Method Protocol (JRMP) or Internet Inter-Orb Protocol (IIOP) to send binary messages to remote hosts. JRMP and IIOP rely on the Transmission Control Protocol/Internet Protocol (TCP/IP) to transport the messages across the network, which makes the communication channel subject to network sniffing. Theoretically, you can write a tunnel that would receive and log the messages before passing them on to the intended receiver, but this would be a very tedious task. We will therefore rely on network sniffing to eavesdrop on RMI. Unfortunately, no tool has native support for higher-level Java protocols, similar to how HTTP Sniffer supports HTTP. This means we must work at a lower level, studying TCP and IP packets that represent all or parts of RMI calls. We will practice spying on the conversation between two users of Chat applications.

The RMI Transport Protocol

RMI uses a concept of a stream to represent the wire format. Internally, the communication has two associated streams—an out stream and an in stream. The streams are mapped to the corresponding socket streams and used to send and acknowledge messages. The output stream consists of a header followed by a sequence of messages. If HTTP is used, the output stream is embedded in an HTTP message. The header contains the protocol identification and attributes describing the type of protocol used. The essence of an RMI call is contained in the message section. Output messages consist of method calls, remote ping, or garbage collection traffic; input messages can be the return result of a call or an acknowledgement of a ping.

To execute a remote call, RMI uses the Java Object Serialization protocol to format the method name and parameter values into a binary structure that is sent across the wire. Therefore, all remote calls follow the same format on the binary level. Knowledge of the transport specifics is not required for eavesdropping on RMI, but if you would like to get more information on it, go to `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`. For our purpose, it is sufficient to know that when two Java applications use RMI to exchange messages, a low-level TCP/IP connection with input and output streams is created and used.

Using a Network Sniffer to Intercept RMI Messages

One of the most popular network sniffing tools is Ethereal. It is free and has ports to Unix and Windows. It can analyze just about any conceivable protocol (save for RMI) and, even though its user interface is somewhat crude, it will do in the absence of a better tool. Ethereal relies on WinPcap, the same library HTTP Sniffer uses to capture network packets. Download,

install, and run Ethereal. Start recording the network traffic just to get an idea about the type of information it can capture. Open a browser and visit a couple of Web sites; then stop recording and look at the items displayed in the top panel. You'll see several packets. Our first task is to configure the tool to show only the information we are interested in seeing.

We start by opening the Capture Options dialog box, which is displayed when you select Start from the Capture menu. Deselect the Capture Packets in Promiscuous Mode button, select Update List of Packets in Real Time, and make sure that all the Stop Capture After buttons are deselected. Because network sniffing requires the client and server run on different hosts, run Chat on two machines. Click OK to confirm the capture options and start recording. Then switch to Chat and send a `Hi Alex` message to the other host. (I won't be offended if you use your own name instead of mine.) Send one more message and remember the text (my second message is `IT jobs are becoming boring`); you will need it later. Then switch back to Ethereal and stop the capture.

We have captured a lot of packets and, to see the Chat conversation, we need a way of filtering the packets that carry the pieces of the text messages users sent. We can start by searching for a packet that carries the string we know was sent during the recording. Select Find Frame from the Edit menu to bring up the Search dialog box. Enter a part of the message you sent (I would enter `Alex`), and be sure to select the String option for the Find Syntax button group. This tells Ethereal to search for the string in any part of the packet. Click OK and the packet that contained the string should become selected. In the lower panels you should see the contents of the packet, which should include your search string among the binary content. Because one RMI call can be broken down into multiple TCP/IP messages, it generally helps to use the Follow TCP Stream feature that reassembles the stream and shows it in a separate window. Right-click the selected packet frame and select Follow TCP Stream.

At this point, we are looking at the binary version of an RMI packet. It is somewhat cryptic, but with a little bit of patience we can gain some knowledge out of it. It starts with the header `JRMI`, indicating that the JRMI protocol is used for RMI transport. Next is the IP address of the originating host, the object ID of the server, and miscellaneous distributed garbage collection (DGC) information. The message content is at the very end. In my case, I can see the string `Hi Alex` followed by `covertjava.chat.MessageInfo`, which we guess is the message class name. It is followed by other parameters of the message, such as the hostname and username.

After examining the TCP/IP stream representing the RMI message sent by Chat, we can make an assumption that subsequent messages will have the same format. In other words, even though the message text might change, the header and the format of the message will remain the same. If this assumption is correct, we should be able to search for Chat messages based on a substring from the message header or format. To test this theory, let's try searching all frames for `covertjava.chat.MessageInfo` using the Find Frame dialog box. If the filter string is set at the bottom of the screen, reset the filter. Then go to the first frame and execute the search. The first frame I found has the `Hi Alex` string and, after searching for the

next frame, I found a message with `IT jobs are becoming boring` in the content. This supports our assumption about Chat and allows us to follow the conversation between the users.

A similar approach can be used for other applications. Ethereal works at a very low level, but with due diligence and a little bit of analysis, you can eavesdrop on network communication between any applications.

Protecting RMI Applications from Eavesdropping

Based on what we have learned about network sniffing, we can conclude that you cannot really prevent eavesdropping on network communications. At the end of the day, the data has to travel over the wire and go through several network nodes, which makes it susceptible to interception. The only way to protect the data is to encrypt it. SSL offers industry-standard features for securing network channels, and it can be used for RMI communication. Java Secure Sockets Extension (JSSE) is a set of APIs that enable Java applications to take advantage of SSL for data encryption, authentication, and message integrity. RMI enables applications to provide custom socket factories for exported objects that are used instead of the default TCP/IP sockets. Using JSSE SSL factories as custom socket factories for both the client and server effectively secures the channel. An example of an RMI and SSL marriage is provided on JavaSoft's Web site at the following URL:

```
http://java.sun.com/j2se/1.4.1/docs/guide/rmi/socketfactory/SSLInfo.html
```

SSL communication does introduce an overhead, which might be unnecessary for all possible client/server data exchange. It might be a better option to keep using the default socket factories, but to encrypt the sensitive parts of the data using JSSE. An example of using Java encryption is shown in Chapter 19, "Protecting Commercial Applications from Hacking."

Eavesdropping on JDBC Driver and SQL Statements

Most of the server applications use a database to store and retrieve data. Knowing how the application interacts with the database and which SQL statements it employs can be of great help in performance tuning or reverse engineering. The overwhelming technology of choice for persistence is JDBC, and it is not going to change any time soon. To use JDBC, an application must identify and load a driver, obtain a connection, and then execute statements to perform updates and queries. The most interesting logic from the performance tuning and reverse engineering perspective is the structure and the parameters of SQL statements. It is common knowledge that database performance depends immensely on how effective the application SQL statements are. Analyzing the SQL and the execution times can lead to improvements on the application side, database side, or both.

Theoretically, you can go through the application source or bytecode and collect all the SQL statements that are stored as strings. This can, of course, be problematic for applications that have a large number of statements or form the SQL dynamically. You also can rely on the database itself to provide the logging of the SQL statements. Although this is definitely a better option than the application code, it requires administrative privileges to the database and might not be very easy if multiple applications share the same database.

JDBC API provides a method of logging the database operations at the driver level via the `DriverManager`'s `setLogWriter` method. It outputs information such as the registration of drivers, URLs used to create database connections, and connection class names. The problem with driver manager logging is that it does not provide the most important information, such as the SQL statements and values passed to the database. I have redirected the JDBC driver logging to System.out and tested it with a simple program that registers a driver, obtains a database connection, and executes a few `SELECT` statements with parameters. Examining the output of driver logging in Listing 13.1 shows no traces of the `SELECT` statements the program executed.

LISTING 13.1 JDBC Driver Logging Output

```
DriverManager.initialize: jdbc.drivers = null
JDBC DriverManager initialized
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
DriverManager.deregisterDriver: com.sybase.jdbc2.jdbc.SybDriver@1d4c61c
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
DriverManager.getConnection("jdbc:sybase:Tds:he1unx142:2075/pslwrkdb1")
    trying driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
getConnection returning driver[className=
com.sybase.jdbc2.jdbc.SybDriver...]
```

STORIES FROM THE TRENCHES

At Riggs Bank we have started using Wily Introscope for performance monitoring of a cluster of servers. Introscope is a Java application that collects performance metrics at runtime and allows storing them in a relational database for later analysis. Built around good ideas, Introscope was lacking polishing and, with the amount of performance data we had, it was simply failing to work with the historical data. The tables used by Introscope grew to millions of records, and we suspected that inefficient database design and SQL statements were the cause of the problem. The technical support did not have a lot of suggestions other than, "Keep less data or upgrade the machine." To revive the product, we decided to spy on the SQL used to execute the queries. After we logged and analyzed the SQL statements, we were able to add indexes to tables, optimize the database design, and reconfigure the persistence within the product. This resulted in a tenfold increase in query performance and a great win for the development team.

To provide a reliable way of eavesdropping on database calls, you must replace the JDBC driver with a wrapper that logs the statement and then delegates to the "real" driver. There is a very good old maxim that states that a good technology/product makes "simple things easy and complex things possible." P6Spy, which we'll use for JDBC logging, demonstrates just that. In less than 10 minutes and with minimal configuration changes, it enables the logging of database calls from existing applications without any code changes. P6Spy is a free, open source application that can be downloaded from SourceForge.

After downloading and installing P6Spy, the `p6spy.jar` and `spy.properties` files must be placed on the `CLASSPATH` of the target application. To activate the spy, the real driver class used by the application should be replaced with the spy driver class. The name of the real driver should be configured in `spy.properties` so the spy can delegate to it. The rest of the database-related configuration for the application does not need to change. For example, if the application is connecting to an Oracle database using the standard driver, the name of the real driver class is `oracle.jdbc.driver.OracleDriver`. To activate the spy, this name should be replaced with `com.p6spy.engine.spy.P6SpyDriver` in the application configuration file and in `spy.properties` the real driver must be configured as follows:

```
realdriver=oracle.jdbc.driver.OracleDriver
```

Restarting the application produces a log that contains the intercepted database calls. For example, after configuring P6Spy with the standard JDK demo application `TableExample` and typing a few `SELECT` statements in the table, I got the following log:

```
1066073757578¦16¦0¦statement¦¦SELECT * FROM TAB
1066073780718¦0¦0¦statement¦¦SELECT * FROM TAB where tname='Alex'
```

P6Spy uses Log4J and is very customizable. Refer to the product documentation for further details.

Quick Quiz

1. What are the general approaches to eavesdropping?
2. Why is it easy to eavesdrop on HTTP communication?
3. How can HTTP communication be protected?
4. What makes network sniffing possible?
5. How can you eavesdrop on RMI communication?
6. What is the most reliable way of intercepting JDBC SQL statements?

In Brief

- *Eavesdropping* is intercepting and logging the message exchange between a client and server. It is possible because the communication has to traverse the process or machine boundaries.
- *Tunneling* is a technique that is based on placing an intermediary between a client and server and reconfiguring the client to send messages to the intermediary. The intermediary dispatches the requests to the server and forwards the response back to the client, logging the communication in the process.
- Eavesdropping on HTTP is easy due to the simple text-based nature of the protocol and a wide variety of tools.
- Network sniffing is possible because the networking protocol packets have to travel between nodes. A promiscuous network node chooses to receive all packets regardless of the destination address, which enables the sniffer to intercept the messages traversing the network.
- RMI eavesdropping is possible using a network sniffer such as Ethereal. It requires analysis of lower-level network packets that carry the binary contents of RMI messages.
- JDBC eavesdropping can be easily achieved with the help of wrappers around the real driver and the connection objects. The wrappers log every statement before forwarding it to the real driver classes.