**ANALYSIS FROM THE WIRE**

In Chapter 2, I discussed how to capture network traffic for analysis. Now it’s time to put that knowledge to the test. In this chapter, we’ll examine how to analyze captured network protocol traffic from a chat application to understand the protocol in use. If you can determine which features a protocol supports, you can assess its security.

Analysis of an unknown protocol is typically incremental. You begin by capturing network traffic, and then analyze it to try to understand what each part of the traffic represents. Throughout this chapter, I’ll show you how to use Wireshark and some custom code to inspect an unknown network protocol. Our approach will include extracting structures and state information.

**The Traffic-Producing Application: SuperFunkyChat**

the test subject for this chapter is a chat application I’ve written in C# called superFunkyChat, which will run on Windows, Linux, and macOS. Download the latest prebuild applications and source code from the GitHub page at <https://github.com/tyranid/ExampleChatApplication/releases/>; be sure to choose the release binaries appropriate for your platform. (If you’re using Mono, choose the .NET version, and so on.) The example client and server console applications for SuperFunkyChat are called ChatClient and ChatServer.

After you’ve downloaded the application, unpack the release files to a directory on your machine so you can run each application. For the sake of simplicity, all example command lines will use the Windows executable binaries. If you’re running under Mono, prefix the command with the path to the main mono binary. When running files for .NET Core, prefix the command with the dotnet binary. The files for .NET will have a .dll extension Instead of .exe.

**Starting the Server**

Start the server by running ChatServer.exe with no parameters. If successful, it should print some basic information, as shown in Listing 5-1.

**NOTE**

*Pay attention to the warning! This application has not been designed to be a secure chat system.*

Notice in Listing 5-1 that the final line prints the port the server is running on (12345 In this case) and whether the server has bound to all interfaces (global). You probably won’t need to change the port (--port NUM), but you might need to change whether the application is bound to all interfaces if you want clients and the server to exist on different computers. This is especially important on Windows. It’s not easy to capture traffic to the local loopback interface on Windows; if you encounter any difficulties, you may need to run the server on a separate computer or a virtual machine (VM). To bind to all interfaces, specify the –global parameter.

**Starting Clients**

With the server running, we can start one or more clients. To start a client, run chatClient.exe (see Listing 5-2), specify the username you want to use on the server (the username can be anything you like), and specify the server hostname (for example, localhost). When you run the client, you should see output similar to that shown in Listing 5-2. If you see any errors, make sure you’ve set up the server correctly, including requiring binding to all interfaces or disabling the firewall on the server.

**Communicating Between Clients**

After you’ve completed the preceding steps successfully, you should be able to connect multiple clients so you can communicate between them. To send a message to all users with the ChatClient, enter the message on the command line and press ENTER.

The ChatClient also supports a few other commands, which all begin with a forward slash (/), as detailed in Table 5-1

**A Crash Course in Analysis with Wireshark**

In Chapter 2, I introduced Wireshark but didn’t go into any detail on how to usewireshark to analyze rather than simply capture traffic. Because Wireshark is a verypowerful and comprehensive tool, I’ll only scratch the surface of its functionality here.When you first start Wireshark on Windows, you should see a window similar to the oneshown in Figure 5-1.

The main window allows you to choose the interface to capture traffic from. To ensure we capture only the traffic we want to analyze, we need to configure some options on the interface. Select **Capture** ▸ **Options** from the menu. Figure 5-2 shows the options dialog that opens.

The Conversations window shows three separate TCP conversations in the captured traffic. We know that the SuperFunkyChat client application uses port 12345, because we see three separate TCP sessions coming from port 12345. These sessions should correspond to the three client sessions shown in Listing 5-4, Listing 5-5, and Listing 5-6.

**Reading the Contents of a TCP Session**

To view the captured traffic for a single conversation, select one of the conversations in the Conversations window and click the Follow Stream button. A new window displaying the contents of the stream as ASCII text should appear, as shown in Figure 5-5.

Wireshark replaces data that can’t be represented as ASCII characters with a single dot character, but even with that character replacement, it’s clear that much of the data is being sent in plaintext. That said, the network protocol is clearly not exclusively a text that based protocol because the control information for the data is nonprintable characters. The only reason we’re seeing text is that SuperFunkyChat’s primary purpose is to send text messages.

Wireshark shows the inbound and outbound traffic in a session using different colors: for outbound traffic and blue for inbound. In a TCP session, outbound traffic is from the client that initiated the TCP session, and inbound traffic is from the TCP server. Because we’ve captured all traffic to the server, let’s look at another conversation. To change the conversation, change the Stream number ➊ in Figure 5-5 to 1. You should now see a different conversation, for example, like the one in Figure 5-6.

**Identifying Packet Structure with Hex Dump**

At this point, we know that our subject protocol seems to be part binary and part text, which indicates that looking at just the printable text won’t be enough to determine all the various structures in the protocol. To dig in, we first return to Wireshark’s Follow TCP Stream view, as shown in Figure 5-5, and change the Show and save data as drop-down menu to the Hex Dump option. The stream should now look similar to Figure 5-7.

**Viewing Individual Packets**

Notice how the blocks of bytes shown in the center column in Figure 5-7 vary in length.Compare this again to Figure 5-6; you’ll see that other than being separated by direction,all data in Figure 5-6 appears as one contiguous block. In contrast, the data in Figure 5-7might appear as just a few blocks of 4 bytes, then a block of 1 byte, and finally a muchlonger block containing the main group of text data.

What we’re seeing in Wireshark are individual packets: each block is a single TCP packet, or segment, containing perhaps only 4 bytes of data. TCP is a stream-based protocol, which means that there are no real boundaries between consecutive blocks of data when you’re reading and writing data to a TCP socket. However, from a physical perspective, there’s no such thing as a real stream-based network transport protocol. Instead, TCP sends individual packets consisting of a TCP header containing information, such as the source and destination port numbers as well as the data.

In fact, if we return to the main Wireshark window, we can find a packet to prove that wireshark is displaying single TCP packets. Select **Edit ▸ Find Packet**, and an additional drop-down menu appears in the main window, as shown Figure 5-8.

**Determining the Protocol Structure**

To simplify determining the protocol structure, it makes sense to look only at one direction of the network communication. For example, let’s just look at the outbound direction (from client to server) in Wireshark. Returning to the Follow TCP Stream view, select the Hex Dump option in the Show and save data as drop-down menu.

**Testing Our Assumptions**

At this point in such an analysis, I stop staring at a hex dump because it’s not the mostefficient approach. One way to quickly test whether our assumptions are right is to exportthe data for the stream and write some simple code to parse the structure. Later in thischapter, we’ll write some code for Wireshark to do all of our testing within the GUI, butfor now we’ll implement the code using Python on the command line.To get our data into Python, we could add support for reading Wireshark capture files,but for now we’ll just export the packet bytes to a file.

**Dissecting the Protocol with Python**

Now we’ll write a simple Python script to dissect the protocol. Because we’re just extracting data from a file, we don’t need to write any network code; we just need to open the file and read the data. We’ll also need to read binary data from the file—specifically, a network byte order integer for the length and unknown 4-byte block.

**Performing the Binary Conversion**

We can use the built-in Python struct library to do the binary conversions. The script should fail immediately if something doesn’t seem right, such as not being able to read all the data we expect from the file. For example, if the length is 100 bytes and we can read only 20 bytes, the read should fail.

**Handling Inbound Data**

If you ran Listing 5-9 against an exported inbound data set, you would immediately get an error because there’s no magic string BINX in the inbound protocol, as shown in Listing 5-11. Of course, this is what we would expect if there were a mistake in our analysis and the length field wasn’t quite as simple as we thought.

**Changing Protocol Behavior**

Protocols often include a number of optional components, such as encryption or compression. Unfortunately, it’s not easy to determine how that encryption or compression is implemented without doing a lot of reverse engineering. For basic analysis, It would be nice to be able to simply remove the component. Also, if the encryption or compression is optional, the protocol will almost certainly indicate support for it while negotiating the initial connection. So, if we can modify the traffic, we might be able to change that support setting and disable that additional feature. Although this is a trivial example, it demonstrates the power of using a proxy instead of passive analysis with a tool like Wireshark. We can modify the connection to make analysis easier.

**Final Words**

In this chapter, you learned how to perform basic protocol analysis on an unknown protocol using passive and active capture techniques. We started by doing basic protocol analysis using Wireshark to capture example traffic. Then, through manual inspection and a simple Python script, we were able to understand some parts of an example chat protocol.

We discovered in the initial analysis that we were able to implement a basic Lua dissector for Wireshark to extract protocol information and display it directly in the wireshark GUI. Using Lua is ideal for prototyping protocol analysis tools in Wireshark. Finally, we implemented a man-in-the-middle proxy to analyze the protocol. Proxying the traffic allows demonstration of a few new analysis techniques, such as modifying protocol traffic to disable protocol features (such as encryption) that might hinder the analysis of the protocol using purely passive techniques. The technique you choose will depend on many factors, such as the difficulty of capturing the network traffic and the complexity of the protocol. You’ll want to apply the most appropriate combination of techniques to fully analyze an unknown protocol.