NAMA : M KHOOLLUAH AL KAFI

PRODI : Rekayasa Perangkat Lunak

NIM : 2121700009

MATKUL : Keamanan Perangkat Lunak

**Replaying Existing Captured Network Traffic**

Ideally, we want to do only the minimum necessary to implement a client or server for security testing. One way to reduce the amount of effort required is to capture example network protocol traffic and replay it to real clients or servers. We’ll look at three ways to achieve this goal: using Netcat to send raw binary data, using Python to send UDP packets, and repurposing our analysis code in Chapter 5 to implement a client and a server.

**Using Python to Resend Captured UDP Traffic**

One limitation of using Netcat is that although it’s easy to replay a streaming protocol such as TCP, it’s not as easy to replay UDP traffic. The reason is that UDP traffic needs to maintain packet boundaries, as you saw when we tried to analyze the Chat Application protocol in Chapter 5. However, Netcat will just try to send as much data as it can when sending data from a file or a shell pipeline.

Instead, we’ll write a very simple Python script that will replay the UDP packets to the server and capture any results. First, we need to capture some UDP example chat protocol traffic using the ChatClient’s --udp command line parameter. Then we’ll use Tshark to save the packets to the file udp\_capture.pcap, as shown here:

One difference in extracting the data from the UDP capture is that Tshark automatically tries to parse the traffic as the GVSP protocol. This results in the data field not being available. Therefore, we need to disable the GVSP dissector to create the correct output.

**Repurposing Our Analysis Proxy**

In Chapter 5, we implemented a simple proxy for SuperFunkyChat that captured traffic and implemented some basic traffic parsing. We can use the results of that analysis to implement a network client and a network server to replay and modify traffic, allowing us to reuse much of our existing work developing parsers and associated code rather than having to rewrite it for a different framework or language.

**Capturing Example Traffic**

Before we can implement a client or a server, we need to capture some traffic. We’ll use the parser.csx script we developed in Chapter 5 and the code in Listing 8-5 to create a proxy to capture the traffic from a connection.

Listing 8-5 sets up a TCP listener on port 4444, forwards new connections to 127.0.0.1 port 12345, and captures the traffic. Notice that we still add our parsing code to the proxy at ➊ to ensure that the captured data has the data portion of the packet, not the length or checksum information. Also notice that at ➋, we write the packets to a file, which will include all outbound and inbound packets. We’ll need to filter out a specific direction of traffic later to send the capture over the network

Run a single client connection through this proxy and exercise the client a good bit. Then close the connection in the client and press ENTER in the console to exit the proxy and write the packet data to packets.bin. (Keep a copy of this file; we’ll need it for our client and server.)

**Implementing a Simple Network Client**

Next, we’ll use the captured traffic to implement a simple network client. To do so, we’ll use the NetClientTemplate class to establish a new connection to the server and provide us with an interface to read and write network packets. Copy Listing 8-6 into a file named chapter8\_client.csx.

One new bit in this code is that each script gets a list of command line arguments in the args variable ➊. By using command line arguments, we can specify different packet capture files without having to modify the script.

The NetClientTemplate is configured ➋ similarly to our proxy, making connections to 127.0.0.1:12345 but with a few differences to support the client. For example, because we parse the initial network traffic inside the Parser class, our capture file doesn’t contain the initial magic value that the client sends to the server. We add an InitialData array to the template with the magic bytes ➌ to correctly establish the connection.

We then read the packets from the file ➍ into a packet collection. When everything is configured, we call Connect() to establish a new connection to the server ➎. The Connect() method returns a Data Adapter that allows us to read and write parsed packets

on the connection. Any packet we read will also go through the Parser and remove the length and checksum fields.

Next, we filter the loaded packets to only outbound and write them to the network connection ➏. The Parser class again ensures that any data packets we write have the appropriate headers attached before being sent to the server. Finally, we read out packets and print them to the console until the connection is closed or the read times out ➐.

When you run this script, passing the path to the packets we captured earlier, it shouldconnect to the server and replay your session. For example, any message sent in the original capture should be re-sent.

Of course, just replaying the original traffic isn’t necessarily that useful. It would be more useful to modify traffic to test features of the protocol, and now that we have a very simple client, we can modify the traffic by adding some code to our send loop. For example, we might simply change our username in all packets to something else—say from user1 to bobsmith—by replacing the inner code of the send loop (at ➏ in Listing 8-6) with the code shown in Listing 8-7.

To edit the username, we first convert the packet into a format we can work with easily. In this case, we convert it to a binary string using the ToDataString() method ➊, which results in a C# string where each byte is converted directly to the same character value. Because the strings in SuperFunkyChat are prefixed with their length, at ➋ we use the \uXXXX escape sequence to replace the byte 5 with 8 for the new length of the username. You can replace any nonprintable binary character in the same way, using the escape sequence for the byte values.

**Implementing a Simple Server**

We’ve implemented a simple client, but security issues can occur in both the client and server applications. So now we’ll implement a custom server similar to what we’ve done for the client.

First, we’ll implement a small class to act as our server code. This class will be created for every new connection. A Run() method in the class will get a Data Adapter object, essentially the same as the one we used for the client. Copy Listing 8-8 into a file called chat\_server.csx

using CANAPE.Nodes;

using CANAPE.DataAdapters;

using CANAPE.Net.Templates;

➊ class ChatServerConfig {

public LogPacketCollection Packets { get; private set; }

public ChatServerConfig() {

Packets = new Log

PacketCollection();z

}

}

➋ class ChatServer : BaseDataEndpoint {

public override void Run(IDataAdapter adapter, ChatServerConfig config) { Console.WriteLine("New Connection"); ➌ DataFrame frame = adapter.Read(); // Wait for the client to send us the first packet if (frame != null) {

// Write all packets to client

➍ foreach(var packet in config.Packets) {

adapter.Write(packet.Frame);

}

} frame = adapter.Read();

}

}

The code at ➊ is a configuration class that simply contains a log packet collection. We could have simplified the code by just specifying LogPacketCollection as the configuration type, but doing so with a distinct class demonstrates how you might add your own configuration more easily.

The code at ➋ defines the server class. It contains the Run() function, which takes a data adapter and the server configuration, and allows us to read and write to the data adapter after waiting for the client to send us a packet ➌. Once we’ve received a packet, we immediately send our entire packet list to the client ➍.

Note that we don’t filter the packets at ➍, and we don’t specify that we’re using any particular parser for the network traffic. In fact, this entire class is completely agnostic to the SuperFunkyChat protocol. We configure much of the behavior for the network server inside a template, as shown in Listing 8-9.

**Repurposing Existing Executable Code**

In this section, we’ll explore various ways to repurpose existing binary executable code to reduce the amount of work involved in implementing a protocol. Once you’ve determined a protocol’s details by reverse engineering the executable (perhaps using some tips from Chapter 6), you’ll quickly realize that if you can reuse the executable code, you’ll avoid having to implement the protocol.

Ideally, you’ll have the source code you’ll need to implement a particular protocol, either because it’s open source or the implementation is in a scripting language like Python. If you do have the source code, you should be able to recompile or directly reuse the code in your own application. However, when the code has been compiled into a binary executable, your options can be more limited. We’ll look at each scenario now.

Managed language platforms, such as .NET and Java, are by far the easiest in which to reuse existing executable code, because they have a well-defined metadata structure in compiled code that allows a new application to be compiled against internal classes and methods. In contrast, in many unmanaged platforms, such as C/C++, the compiler will make no guarantees that any component inside a binary executable can be easily called externally.

Well-defined metadata also supports reflection, which is the ability of an application to support late binding of executable code to inspect data at runtime and to execute arbitrary methods. Although you can easily decompile many managed languages, it may not always be convenient to do so, especially when dealing with obfuscated applications. This is because the obfuscation can prevent reliable decompilation to usable source code.

Of course, the parts of the executable code you’ll need to execute will depend on the application you’re analyzing. In the sections that follow, I’ll detail some coding patterns and techniques to use to call the appropriate parts of the code in .NET and Java applications, the platforms you’re most likely to encounter.