Project 1

CS 7733 Network Security, Dr. Papa

Ian Riley, Graduate Student

02-28-18

Sniffer

The first deliverable of project one is to design and implement a packet sniffer. The sniffer is required to read packets from both a file and a network adapter. It must also be capable of reading Ethernet, ARP, IPv4, ICMP, TCP, and UDP packets. To implement such a sniffer, I use a few third-party libraries for the command line interface and the network adapter and developed my own set of classes to represent the many packet types. The set of classes that I implemented collectively represent the structure of a packet, the many supported packet header types, a bit buffer, and a hex buffer.

First, to implement a command line interface, I downloaded and imported the Apache Commons CLI. The Apache CLI can be found freely available online with supporting documentation. The CLI makes it much easier to state and document each command line argument. It also provides functionality for command flags which require multiple arguments as well as command flags which are required for the tool to perform its function. Most importantly, it provides the necessary API to check if a command flag has been provided as well as to fetch the corresponding input argument. It also provides neat formatting for the help documentation so that it is easy to read the documentation on how to use the command line interface.

However, it does not provide the necessary APIs to convert numeric input arguments to the correct type. The API for Apache Commons CLI only supports each arg as the String type. Rather than implementing my own check to ensure that provided numeric arguments only contain numeric characters, I downloaded the Apache Commons Lang which provides a StringUtils class which can assert certain qualities of a string. The Apache Commons Lang is also freely available online with supporting documentation. By using the StringUtils class, I was able to assert that each numeric input argument contained only numbers before converting the argument to an appropriate numeric primitive type. This avoids throwing ParseExceptions during the conversion process from a String type to a numeric type.

Next, to integrate with the network adapter, I installed the JNetPcap library rather than using the SimplePacketDriver library that was provided in class. The JNetPcap is a freely available library with supporting documentation. Unfortunately, their support forum is indefinitely down, but StackOverflow still contains a great deal of documentation. JNetPcap is a common library to use in Java for a network adapter. I chose to use this library since I was unable to support the SimplePacketDriver using the Eclipse IDE and since using JNetPcap taught me how to build a packet sniffer using a fully-formed network adapter interface. The JNetPcap does provide its own APIs to parse packet headers, but I did not feel that using this section of the API was within the spirit of the project. As such, I only used JNetPcap APIs to check live network adapters, listen to a select network adapter for outgoing and incoming packets, and send packets out via the selected adapter. When a packet is received or read from file, the packet is dumped to a byte buffer and then mapped to a set of classes which I wrote specifically for this project using my own parser. If the count flag is provided, then only count number of packets are read before the adapter is closed. The available adapters can be seen using the '- show' command flag and an adapter can be selected using the ‘-dev’ flag. I found this very useful since I have multiple network adapters available on my machine, but only one supports traffic and it is not often the adapter that is selected by default.

When a packet is received over the network, the packet is dumped into a byte buffer and then it is converted to a hex buffer. The byte buffer is the ByteBuffer class that is provided through Java’s core libraries, but the hex buffer is my own. I designed two string-like classes for this project. The first is a BitString class which contains only binary characters. The second is a HexString class which only contains the hexadecimal characters. The classes are immutable once they have been constructed and provide a number of string operations such as concat and substring which produce new objects with respect to the operation. These classes were designed for two reasons. The first is that it is much easier to work with Strings in Java than it is to work with any other data type. By using a string, the entire packet can be dumped into a string buffer and then parsed in subsections via the substring operation. In addition, by using a string, there is no need to differentiate between one’s and two’s complement. Also, Java has a funny way of converting primitive types to strings of bits and back. In particular, Java does not produce bit strings with leading zeroes. As such, leading zeroes must be manually inserted when writing out the packet data to the network or to a file. Not having leading zeroes is also very misleading since it changes the arrangement of the bits as they correspond to the fields of each header. As such, while it is technically computationally more expensive to work with strings, it is substantially easier to implement. The second reason is that the packets had to be structured in a particular format which uses hex notation. It is far easier to read packets and write packets using a HexString which encodes the packets as hex. These two classes, BitString and HexString, also handle the conversion of bytes to hex and hex to bytes.

A packet class was implemented to represent the general structure of a packet. Packets typically contain a series of headers followed by data, also known as the payload. Each header corresponds to a specific protocol and there are often two or three protocols implemented on top of each other. I designed a Packet class which is essentially just a linked list. The head Packet, or the head of the linked list, is the bottom most protocol, and then, as we move across the list, we move to higher level protocols. For this implementation, the head Packet is Ethernet followed by IP or ARP to IP, followed by ICMP, TCP, or UDP. Each packet contains a header which is the protocol header and points to the next packet. The last packet is always either a packet with a header but no data, or a packet with data but no header. If the last packet has data but no header, then it is a DataPacket. DataPackets are simply a packet wrapper around a HexString which terminate the linked list. Each packet can be converted to a HexString which will contain the bytes for the packet’s header, all next headers, and the payload. A more concise way to explain this Packet implementation is that it is essentially a linked list of protocol headers which terminate with a payload, empty or otherwise.

Unless the Packet is a DataPacket, it must have a header. There are six classes of headers, one for each supported protocol: Ethernet, ARP, IPv4, ICMP, TCP, and UDP. Each class extends the abstract class Header. To be a Header, each class must provide a string representation and a hex representation so that the header can be output to a packet file in hex notation. Each header class must also provide a parse function and a number of fields which describe the minimum, maximum, and actual size of the header. Unfortunately, Java does not provide the ability to express abstract static fields so the Header class cannot enforce all of the above requirements on its children.

The Sniffer functions as follows. The Sniffer first instantializes a command line interface which parses and validates the provided input arguments. Then, if '-help' is used, documentation on the CLI is provided and the Sniffer terminates. Otherwise, if '-show' is used, then a list of open network adapters is provided and the Sniffer terminates. Otherwise, the Sniffer uses the CLI to determine if it should sniff a file or the network. If an output argument has been used, then the output file is overwritten. If an input argument has been provided, then the input file, which is in hex notation, is read. Each packet from the input file is dumped into a HexString. Each HexString is converted to an Ethernet packet which contains an EthernetHeader followed by either an IPHeader or an ARPHeader. If it is an ARPHeader, then the ARPHeader is followed by an IPHeader. The IPHeader is followed by either a ICMPHeader, TCPHeader, or UDPHeader and then data, represented as a HexString. Once the packet has been restructured as a linked list, the packet is either written to a file in the same hex notation, or, if no output file has been provided, is written to std.out in the specified human readable formats. If an output file is used, then the output file should look identical to the input file. If a count is specified, then only so many packets will be read and written out. Additionally, if other args are used, then packets are filtered according to source address, destination address, source port, destination port, and packet type. If a packet type is specified, then only packets which contain the header of the type are written out. If the '-h' flag is used, then only header information will be written out.

If an input file is not specified, then the Sniffer selects a network adapter to sniff. If a device name is provided, then the Sniffer selects the adapter, if it is open. If it is not open, then the Sniffer terminates. If a device name is not specified, then the first adapter is opened. Once the adapter has been opened, the Sniffer listens for any traffic on the adapter. Each packet that passes through the adapter is dumped to a byte buffer, converted to a HexString, and then is converted to a linked list of headers using the Packet class. The packet is then written out to file in hex notation, if an output file is specified, or is written to std.out in human readable format, otherwise. Only so many packets are read which is set according to count, and only those packets which satisfy the conditions set by the other args. If a count is set and other conditions are set, then the number of packets read is limited by count, not the number of packets which satisfy all of the conditions. If not count is specified, then the sniffer will monitor the network adapter indefinitely and must be terminated through an interrupt signal.

Generator

The packet generator uses the exact same set of classes and libraries as the Sniffer. The Generator does use a separate CLI which requires that an input argument is specified. The Generator reads the hex data from an input file. The hex data is converted to byte buffers, and then is sent out via a network adapter. The network adapter is selected according to the specified device name, or is the first open network adapter, if no device name is specified. This is done so that the Generator can be forced to use the same network adapter that is being monitored by the Sniffer.

This project was written in Java using the Eclipse IDE. All source code and supporting libraries have been provided and have been made publicly available through Github under the repository of CS7473 of ttowncompiled.

Results

The following results were collected using the Sniffer and Generator, designed as specified above, using the provided sample file and the provided lab network. The network traffic was sniffed using a Surface Pro connected to the network using an ethernet adapter. The sniffer was used twice. The first instance collected 10,000 packets of all types while the second instance collected only 30 ARP packets. The second instance was used to sniff network traffic while the generator was placing packets on the network. The run configuration for the collection of 10,000 packets is

```
-dev \Device\NPF_{E84728CE-492C-4F17-BF2D-8C693B0F1E14} -c 10000 -o C:\\Users\\ianri\\Workspace\\CS7473\\data\\all.dat -hex
```

This command sniffs 10,000 packets and then places them in one file in hex notation. The second run configuration collected 30 ARP packets from the network and dumped them to one file in hex notation. The run configuration is as follows:

```
-dev \Device\NPF_{E84728CE-492C-4F17-BF2D-8C693B0F1E14} -c 30 -t arp -o C:\\Users\\ianri\\Workspace\\CS7473\\data\\allarp.dat -hex
```

The data generated by the previous command was placed into one file. That file was then parsed multiple times by the sniffer to the produce the required deliverables. The deliverables are as follows:

1. *Obtain the Client IP, username and password of the telnet session to 192.168.1.22.* The client IP is **192.168.1.66**, the username is **group15**, and the password is **192.168.1.66**. The data for this deliverable was parsed using the following run configuration:

   ```
   -r C:\\Users\\ianri\\Workspace\\CS7473\\data\\all.dat -c 20 -dst 192.168.1.22 -dport 23 23 -o
   C:\\Users\\ianri\\Workspace\\CS7473\\data\\telnet.dat -hex
   ```

2. *Obtain the Client IP of the failed logon session to 192.168.1.66.* The client IP is **192.168.1.62**. The data for this deliverable was parsed using the following run configuration:

   ```
   -r C:\\Users\\ianri\\Workspace\\CS7473\\data\\all.dat -c 10 -dst 192.168.1.66 -dport 21 21 -o
   C:\\Users\\ianri\\Workspace\\CS7473\\data\\ftpfail.dat -hex
   ```

3. *Obtain the Client IP, username, password, and the name of the file transferred to 192.168.1.42.* The client IP is **192.168.1.62**, the username is **group14**, the password is **192.168.1.62**, and the name of the file is **GROUP14.NFO**. The data for this deliverable was parsed using the following run configuration:

   ```
   -r C:\\Users\\ianri\\Workspace\\CS7473\\data\\all.dat -c 15 -dst 192.168.1.42 -dport 20 21 -o
   C:\\Users\\ianri\\Workspace\\CS7473\\data\\ftpsucc.dat -hex
   ```

4. *Obtain the Client IP, name and content of the html file transferred to 192.168.1.10.* The client IP is **192.168.1.22**, the name of the file is **cs7493/HTTP/1.0**, the content of the file is **UUUUUU**. The data for this deliverable was parsed using the following run configuration:

   ```
   -r C:\\Users\\ianri\\Workspace\\CS7473\\data\\all.dat -c 10 -dst 192.168.1.10 -dport 80 80 -o
   C:\\Users\\ianri\\Workspace\\CS7473\\data\\http.dat -hex
   ```

5. *Obtain the IP address of hosts iodine and hydrogen as returned by the DNS servers at 192.168.1.14 and 192.168.1.46.* The client IP for 192.168.1.14 returned by the DNS server is **192.168.1.66** and the client IP for 192.168.1.46 returned by the DNS server

is **192.168.1.22**. The data for this deliverable was parsed using the following run configurations:

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\data\\all.dat -c 5 -sord 192.168.1.14 192.168.1.14 -o
C:\\Users\\ianri\\Workspace\\CS7473\\data\\dns14.dat -hex
```

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\data\\all.dat -c 5 -sord 192.168.1.46 192.168.1.46 -o
C:\\Users\\ianri\\Workspace\\CS7473\\data\\dns46.dat -hex
```

6. *Send an ARP request to 192.168.1.200, capture the ARP reply and obtain the MAC address.* An ARP request was put on the network with the generator using the following run configuration:

```
-dev \Device\NPF_{E84728CE-492C-4F17-BF2D-8C693B0F1E14} -r
C:\\Users\\ianri\\Workspace\\CS7473\\resources\\fakearp.dat -c 10
```

   A packet was created from an ARP packet that was sent to 192.168.1.200 that was previously sniffed from the network. The MAC address that was obtained is **00:00:00:22:15:61:e3:f4**. The data for this deliverable was generated using the following run configuration:

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\data\\allarp.dat -c 10 -o
C:\\Users\\ianri\\Workspace\\CS7473\\data\\arp.dat -hex
```

Copies of the results are also included in the project report as well as with the source code. The source code is included as an archive file produced by the Eclipse IDE.