Project 2

CS7543 Network Security, Dr. Papa

Ian Riley, Graduate Student

03-28-18

Description

The goal for project 2 is to perform packet reassembly. There are five example cases that must be met. First, the packet assembler was handle ARP packets. It is assumed that ARP packets are not fragments. All other cases involved fragmented IP packets. The second case is a fragmented IP packet which contains no overlapping segments. An overlapping segment exists when two fragments contain data which pertains to the same bits of the original message. Unless the overlapping segment is the same in all of the packets that contain the segment, it is theoretically impossible to know what the original message was. Thirdly, the assembly must handle a fragmented IP packet which contains overlapping segments. The fourth case is a fragmented IP packet where the assembled packet would be larger than 64K. Lastly, the packet assembler must timeout if it takes too long to receive all of the fragments from the fragmented packet.

Work

This project is an extension of project 1. The sniffer, designed and implemented in project 1, was used to sniff the packets from the lab network and written to files provided through Harvey. The packet assembler was integrated into the sniffer from project 1. It was not designed as a standalone module. The packet assembler was rather concise to write, and by integrating it into the sniffer, it was simple to apply consistency between the assembler and other features that were already present in the sniffer. For example, the previous functionality of the sniffer was that a ‘count’ could be set so that only ‘count’ packets would be read. This functionality isn’t consistent when there are fragmented packets because the definition of a packet has changed. Is each fragment considered a packet or is only the assembled packet considered a packet, or are both considered packets? The new functionality is that only the assembled packet is considered a packet. By this new functionality, fragments won’t be counted against the global ‘count.’ Additionally, any packets that are either too large to be assembled or who time out are also not counted against the global ‘count.’

Originally, the sniffer was designed only to handle packets. For project 2, a new class was designed. The new class is called a Triple. It is simply a 3-tuple of a sid, a datagram, and an ordered list of fragments. The sid is the situation id of the triple. The datagram is the assembled packet, and the ordered list of fragments contains all of the fragments, ordered by offset. The sniffer now creates Triples rather than packets. Instances where there are no fragments are essentially a triple with no sid and no ordered set of fragments.

There are five kinds of triples, one for each case that must be handled by the packet assembler. The first kind of triple is for ARP packets. This kind of triple has a sid of 0, a datagram which is the ARP packet, and an order set of fragments which only contains one fragment, the original ARP packet. The second kind of triple is for fragmented IP packets with no overlap. This kind of triple has a sid of 1, a datagram which is the assembled packet, and an ordered set of fragments, ordered by offset. The third kind of triple is for fragmented IP packets with overlap. This kind of triple has a sid of 2, a datagram which is the assembled packet, and an ordered list of fragments, ordered by offset. Packets are assembled according to the order of the ordered list of fragments. The order is first determined by offset and then determined by time. Packets which have the same offset are ordered such that those packets that were received first appear later. By assembling the datagram this way, packets which are received later will overwrite packets that are received sooner, and packets which have a smaller offset will overwrite packets with a larger offset. The reasoning for this is to bias packets received later and bias packets which contain data prior to the overlap. The fourth kind of triple is for fragmented IP packets which are too large. This kind of triple has a sid of 3, a datagram which is the fragment with the smallest offset, and an ordered list of fragments, ordered by offset. The last kind of triple is for fragmented IP packets where the session timeouts before all of the fragments have been received. This kind of triple has a sid of 4, a datagram which is the fragment with the smallest offset, and an ordered list of fragments, ordered by offset.

There were three major challenges present in this project. First, it was necessary to learn how to handle fragmented headers. Second, it was necessary to learn when IP packets could be assembled. Lastly, it was necessary to determine when packets should timeout. At first, it was difficult to determine how to assemble packets which contained a fragmented header. However, features had already been designed into the sniffer to handle a hex string representation of a TCP, UDP, or ICMP packet. As such, I designed the assembly process such that it would construct a hex string representation of the assembled packet. This hex string representation is treated as a TCP, UDP, or ICMP packet depending on the protocol that was used in the IP fragments. Once the TCP, UDP, or ICMP packet is created from the hex string representation, the packet is wrapped into a new IP packet. The header of the new IP packet is nearly identical to the IP header of the first fragment, the fragment with the smallest offset, except that the DF flag was turned on, the MF flag was turned off, and the total length was modified to match the total length of the assembled packet.

Determining when to assemble packets was a much simpler matter. All IP packets contain a DF flag, a MF flag, and a fragmentation offset. Any packet whose MF flag is turned on or has a fragmentation offset greater than 0 is a fragment. Every IP packet also has a unique identifier. The unique identifier is used to determine which fragments go together. A global hashmap is used which maps unique identifiers to ordered lists of fragments. When a fragment is received, if its unique identifier has not been received before, a new list is created and it is added as the first fragment. If its unique identifier is registered, then it is added to the ordered list according to its fragmentation offset using insertion sort. After each packet is sorted into its respective list, its list is checked to determine if the fragments are ready for assembly. For a list of fragments to be ready for assembly, they must contain the first fragment of the segment, the last fragment of the segment, and all fragments in between. The first fragment of the segment should have its MF flag set to true and a fragmentation offset of 0. The last fragment of the segment should have its MF flag set to false and a fragmentation offset greater than 0. I then compare consecutive fragments to ensure that there are no gaps between consecutive fragments. A gap exists if the length of the packets data plus its fragmentation offset does not match or overlap the fragmentation offset of the following fragment.

Lastly, I needed to determine a timeout. The Java SQL native library provides a Timestamp class and the Java System class provides a function which returns the current system time in milliseconds.

Each time a fragment is received with a new unique identifier, that identifier is assigned a Timestamp which is set according to the millisecond system time when the packet was sniffed. Each instance a new packet is received, its corresponding timestamp, the timestamp that corresponds to its unique identifier, is compared to the current system time. The flag 'timeout' can be used to set the timeout in seconds. The default timeout is 1800 seconds which is 30 minutes. If the new fragment was received later than the timeout of the first fragment, then the fragments are timed out, and are not assembled.

Results

The results provided below are from the http.frag.1.dat and http.frag.2.dat files provided through Harvey and from ARP traffic collected from the lab. There are five separate files attached below. The first contains 10 packets that were assembled from http.frag.1.dat using the following run configurations. They are shown in human readable format.

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\resources\\http.frag.1.dat -c 10 -o
C:\\Users\\ianri\\Workspace\\CS7473\\output\\http.frag.txt -otype human
```

The second file contains a single assembled packet from http.frag.1.dat which shows that the packet assembler can assemble packets with overlap. They are shown in human readable format with the sid printed above, and the fragments printed in human readable format below. This format is called trip format. This file was generated with the following run configurations.

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\resources\\http.frag.1.dat -c 1 -o
C:\\Users\\ianri\\Workspace\\CS7473\\output\\http.frag.over.txt -otype trip
```

The third file contains a single assembled packet from http.frag.2.dat which shows that the packet assembler can assemble packets without overlap. The packet is shown in human readable format with the sid printed above, and the fragments print in human readable format below. This format is called trip format. This file was generated with the following run configurations.

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\resources\\http.frag.2.dat -c 1 -o
C:\\Users\\ianri\\Workspace\\CS7473\\output\\http.frag.no.txt -otype trip
```

The fourth file contains a number of packets from http.frag.1.dat which were timed out. These packets were timed out because the timeout was set to 0. This shows that the packet assembler can also timeout packets. The triples are shown in trip format. This file was generated with the following run configurations.

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\resources\\http.frag.1.dat -c 1 -o
C:\\Users\\ianri\\Workspace\\CS7473\\output\\http.frag.timeout.txt -otype trip -timeout 0
```

The fifth file contains a single ARP packet which was collected from the network. The packet is printed in trip format. This demonstrates that the packet assembler can also handle ARP packets. This file was generated with the following run configurations.

```
-r C:\\Users\\ianri\\Workspace\\CS7473\\data\\arp.dat -c 1 -o C:\\Users\\ianri\\Workspace\\CS7473\\output\\arp.frag.txt -otype
trip
```