Chapter 5

Transport Layer, UDP Protocols and Attacks

Contents

5.1 The Transport Layer

The job of the IP protocol is to get a packet from the sender machine to the destination machine. Once the packet arrives at its destination, the IP's job is done. However, the operating system still needs to give the data in the packet to the target application. That is the job of the transport layer.

Let us see what functionalities are missing from the IP layer if we want to deliver data from a sender application to the destination application. This will help us understand the functionalities of the transport layer.

- How do we know which application should get the data? When the receiver replies, how does it know what application on the sender side should get the reply?
- What should we do if a packet gets lost, duplicated, or delayed? How do we even know whether a packet has been delivered or not?
- What if packets arrive out of order? For example, when the sender sends packets A, B, and C in the stated order, but when they arrive, C arrives first, what should the receiver do? What if A or B never arrived, should C be given to the application?
- What if the sender sends data too fast for the server, faster than what the receiver can handle, how do we ask the sender to slow down?

The IP protocol handles none of the issues above, because by design, that is not its job. The IP layer implements the best-of-effort delivery mechanism, i.e., it will try its best to deliver packets, but there is no guarantee of the delivery order, and there is no mechanism to handle errors, such as packet loss.

The UDP protocol only handles the first item on the list, i.e., it connects packets to its sender and receiver applications, which is something that the IP does not do. That is the only functionality that UDP adds above the IP layer. Therefore, the UDP protocol is very light-weighted. The other issues listed above are handled by another transport-layer protocol, the TCP protocol. We will discuss TCP and its security problems in another chapter. For this chapter, we only focus on UDP.

I often joke with my students, telling them that if I am hiring people, I would only hire people of the TCP type, not the IP or UDP type. The people of the IP type will try their best to finish their jobs, but when something goes wrong, they will not bother to fix the problem, because they consider fixing the problem is other people's job. They are not reliable. The UDP type is similar. The people of the TCP type are reliable. Not only do they provide a better quality in their jobs, but also when something goes wrong, they try to fix it. They will keep trying, until they either fix the problem or conclude that there is no point continuing the effort.

5.1.1 UDP-Based Applications

Without the need to achieve reliability, UDP is much light weighted than TCP, and can thus achieve better performance than TCP. Because of this advantage, many real-time applications use UDP. For example, real-time video streaming and voice-over-IP applications use UDP. For these applications, errors such as packet losses have minor impact and usually go unnoticed, but long latency is quite unpleasant.

It should be noted that movie streaming is different from real-time video streaming. For example, Netflix, YouTube, and Hulu do use TCP, because these streaming services are not real-time, and delay is not crucial. TCP transfers can be easily accomplished over HTTP and web browsers. That is why these services use TCP. When making a decision on whether to use TCP or UDP, performance, difficulties in implementation, security, and many other factors should be considered.

Some UDP-based applications do need to ensure some degree of reliability, such as maintaining the order of the data, detecting packet loss, etc. They can still do that; they just have to implement these inside the application. For example, they can give each payload a sequence number. Using TCP, applications do not need to worry about this, because these functionalities are already provided by the underlying transport layer protocol.

Another application of UDP is multicast and anycast. Multicast allows a sender to send packets to a group of destination computers simultaneously, while IP anycast allows the sender to send packets to one of the receivers in a group. In both cases, it is quite hard to use the connection-based TCP protocol, but the connectionless UDP protocol is a good candidate.

5.1.2 Port Numbers

Using the mailing address as an analogy, the IP address is like the address of the apartment building. When the mail carrier delivers a letter to the front desk of an apartment building, the post office's job (analogically to the IP protocol) is done. The people at the front desk place the letter to the recipient's mailbox based on the apartment number written on the envelope. The apartment number and the port number serve the same goal.

A port number is a 16-bit unsigned integer, ranging from 0 to 65535. If an application needs to get data from a remote computer, it registers for a port number. The sender will place this number in the transport layer header of the packet, so when the packet arrives at its destination, the operating system can deliver the payload to an application based on the port number.

Theoretically, an application can register for any port number, as long it is not used by another application. In practice, there are conventions:

- 0 – 1023: the well-known port numbers. These port numbers are reserved for well-known applications, such as ssh (22), telnet (23), DNS (53), and WWW (80 and 443). The use of these ports are controlled by the operating system, and only the applications with the root privilege can use the ports in this range.

- 1024 – 49151: the registered port numbers. They are used by vendors for their own server applications. Vendors can register their application's ports with ICANN. Other vendors are supposed to respect these registered values to avoid duplication. However, this assignment is not controlled by the operating system.

- 49152 – 65535: the dynamic or private ports. This range is used for private or customized services, for temporary purposes, and for automatic allocation of ephemeral ports. When a client application sends out packets, it also needs to register for a port number, so the receiver can send the response back. The value of this port number does not matter much, as it is used temporarily. Such a short-lived port is called ephemeral port. Most client applications let the operating system randomly pick one for them. The OS picks the port number from this range.

5.2 The UDP Protocol

5.2.1 UDP Header

UDP is a very simple protocol, adding only one main functionality above the IP protocol, i.e., connecting the incoming packet to its final destination program. This is done through the port numbers, which are the main fields in the UDP header (see Figure 5.1).

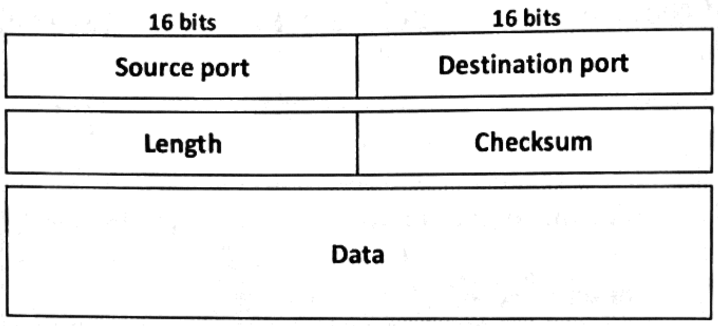

Figure 5.1: UDP header format

The length field specifies the length of the UDP header and UDP data. The minimum length is 8 bytes, which is the length of the header. The checksum field is used for error-checking of the header and data. For UDP in IPv4, the checksum field is optional. If we do not intend to set the checksum field, we need to set this field to zero. If this field is not zero, the recipient will verify the checksum field, and discard the packet if the verification fails.

5.2.2 UDP Client Program

To learn how UDP works, we will implement a UDP client and a UDP server. Python has a `socket` module, which according to its manual, *"is a straightforward transliteration of the Unix system call and library interface for sockets"*. Compared to C, it makes using socket easier, but unlike many other wrapper libraries, it still exposes sufficient lower-level system details, which are essential for learning how things work. Therefore, we choose to use Python to write the UDP client and server programs.

To write a client program, we first create a UDP socket: the `socket.SOCK_DGRAM` aregument indicates that this is the UDP type (TCP uses a different value). Then, we can use this socket to send out UDP packets to any UDP server. In the code, we sent two UDP packets to two different servers with different port numbers. To test the program, we need to start a UDP server on each server. We can use the `nc` command. For example, we can run "`nc -luk 9090`" to start a UDP server listening to port `9090` .

```
#!/bin/env python3
import socket

udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

data = b"Hello, Server 1\n"
udp.sendto(data, ("10.9.0.5", 9090))

data = b"Hello, Server 2\n"
```

```
udp.sendto(data, ("10.9.0.6", 9091))
```

A UDP packet includes a source port and a desintation port, but in the code, we have only specified the desintation port. When a socket is not bound to a port number, the operating system will randomly generate a port number (from a specific range), and assign it to the socket when the socket is first used. In the code above, the first `sendto()` will tigger the port number generation. When the second `sendto()` is invoked, the socket is already bound to a source number.

Although it is not common for client programs, if a client program does want to choose a specific source port number, it can use `bind()` to bind the socket to a source port number. It needs to be done before the OS does that for the socket, i.e., before the first `sendto()`. See the code in the following.

```
udp.bind(("0.0.0.0", 9090))
```

5.2.3 UDP Server Program

To write a UDP server program, we first create a UDP socket. Unlike client programs, for servers, we cannot let the operating system pick a random port, because client programs need to know this number. Therefore, we need to pick a port number, and bind the socket to this number, indicating that the server is listening to this port.

The first argument in `bind()` indicates which interface this socket is bound to, i.e., from which interface will this server get data from. The argument `0.0.0.0` means binding to all the interfaces. If we pick a specific IP address for this argument, the server will only get the traffic from a particular interface. For example, if we use `127.0.0.1`, the server will only listen to the UDP packets from the loopback interface.

```
#!/bin/env python3
import socket

udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp.bind(("0.0.0.0", 9090))

while True:
    data, addr = udp.recvfrom(1024)
    print ("From {}: {}".format(addr, data))
```

Once the server is set up, we can use `recvfrom()` to get data. If no UDP packets are available at the socket, the call waits for a UDP packet to arrive, unless the socket is non-blocking. The call returns both the data and the client's address (including the IP address and port number). The argument in the call specifies the maximal number of bytes that should be retrieved from a UDP packet. If this number is less than the size of the data in a UDP packet, the excess data will be discarded. This is quite different from TCP, which keeps all the unread data in its buffer for the next read operation.

To test this server, we can use our UDP client program to send UDP packets to it, or use `"nc -u <ip> <port>"` to send UDP packets. The `nc` command is a very useful tool; it can serve as TCP and UDP client and server programs. We use it quite a lot in this book.

5.3 Attacks Using UDP

For a connection-based protocol, before two computers communicate with each other, they need to go through a process to establish a connection. This process typically takes at least one round of interaction between the sender and receiver, so if the sender uses a spoofed IP address, it will be quite difficult for them to establish a connection. Without the connection, the receiver will not accept the data from the sender.

UDP is not connection based. To send data to another computer, the sender can simply put the data in a UDP packet, and send it to the destination. The sender can use a spoofed source IP address, and the receiver will accept the data regardlessly. This has made it possible for a computer M to send data to a computer B on behalf of another computer A. When B replies, the reply will go to A. If the resource consumption from M to B is significantly less than the resource consumption from B to A, B can be used to magnify an attacker's power in a denial of service attack against A.

A number of UDP attacks follow the pattern described above. We use an analogy to describe three common approaches used by attackers to magnify their attacking power. For each approach, we will provide an example of a real attack. The analogy is depicted in Figure 5.2.

- Figure 5.2(a): turning one grenade into many grenades
- Figure 5.2(b): creating a regenerable grenade
- Figure 5.2(c): turning a grenade into a missile

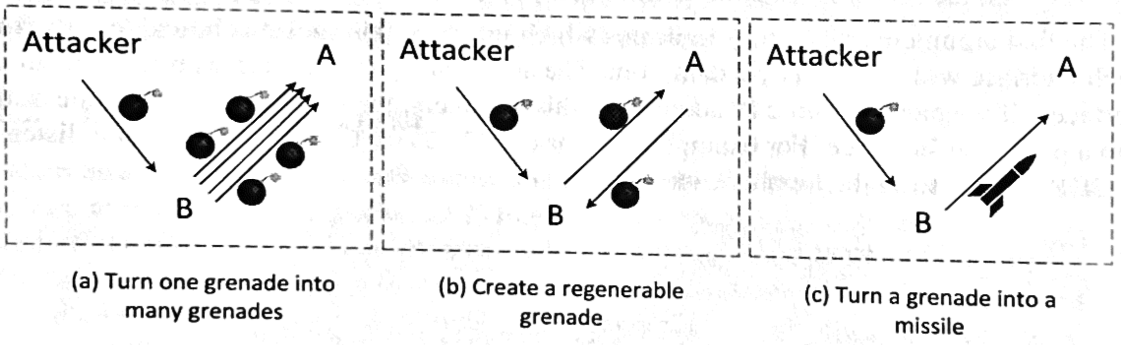
Attacker
A
B
(a) Turn one grenade into many grenades
Attacker
A
B
(b) Create a regenerable grenade
Attacker
A
B
(c) Turn a grenade into a missile

Figure 5.2: An analogy illustrating how attackers magnify their power

5.3.1 Fraggle Attack: Turning One Grenade Into Many

In this category of attacks, the attacker throws one grenade towards B, and B turns it into N grenades and send them towards the victim A. This way, the attacker's power is magnified by N times. The Fraggle attack is an example of such an attack.

The Fraggle attack is a variation of the ICMP-based smurf attack. In this attack, the attackers send a UDP packet to the UDP echo service (port 7). The packet uses a directed broadcast address as its destination IP address, while using the victim A's address as the source IP address. In the old days, routers will broadcast this packet to all the hosts on the destination network, and all the hosts will reply to A. Therefore, if there are N hosts on the network, A will receive N replies. The attacker's power is magnified.

5.3.2 UDP Ping Pong: Creating Regenerable Grenade

In this category of attacks, the attacker creates one grenade, and throw it towards the victim. Every time a grenade explodes, a new grenade is generated, and this process will repeat for many times or infinitely. The UDP Ping Pong attack is an example of such an attack.

Some UDP server programs send back a reply after receiving a UDP packet, without inspecting what is in the packet. This behavior is dangerous, as the servers can become a victim of the UDP ping pong attack. We use an example to illustrate how the attack works. The following is a UDP server program. After receiving a UDP packet from a client, it prints out the data and the client information, before sending a "*Thank you*" message back to the client.

```
#!/bin/env python3
import socket

udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp.bind(("0.0.0.0", 9090))

while True:
    data, (ip, port) = udp.recvfrom(1024)
    print ("From {}:{}: {}".format(ip, port, str(data, 'utf-8')))

    # Send back a "thank you" note
    udp.sendto(b'Thank you!\n', (ip, port))
```

The program seems quite normal, but if this server program is running on two different computers (say A and B), we can let these two computers "play" ping pong with each other infinitely. All we need to do is to send a spoofed packet to one of the servers to trigger the ping pong. The attack program is in the following:

```
#!/bin/env python3
from scapy.all import *

ip   = IP(src="10.9.0.5", dst="10.9.0.6")
udp  = UDP(sport=9090, dport=9090)
data = "Let the Ping Pong game start!\n"
pkt  = ip/udp/data
send(pkt, verbose=0)
```

The attack program sends a spoofed UDP packet to the server running on B (`10.9.0.6`). It puts A's address (`10.9.0.5`) in the source IP address field, while using 9090 as the source port number. When the server on B receives this UDP packet, it sends back a reply to A's port 9090, which gets into the server program running on A, and triggers A to send back a reply to B. This reply will then trigger B to send back a reply back to A. This process will go on between A and B forever, until we stop one of the servers.

If we use `tcpdump` to monitor the interface on one of the machines, we will see the ping pong "ball" going back and forth between the two victims. Here the ping pong ball is the regenerated grenade. We only list a very small portion in the following, but from the timestamp at the beginning of each record, we can see that the rate is quite fast. This is a denial-of-service attack.

```
02:44:58.837942 IP 10.9.0.5.9090 > 10.9.0.6.9090: UDP, ...
02:44:58.837994 IP 10.9.0.6.9090 > 10.9.0.5.9090: UDP, ...
```

```
02:44:58.838218 IP 10.9.0.5.9090 > 10.9.0.6.9090: UDP, ...
02:44:58.838298 IP 10.9.0.6.9090 > 10.9.0.5.9090: UDP, ...
02:44:58.840450 IP 10.9.0.5.9090 > 10.9.0.6.9090: UDP, ...
02:44:58.840560 IP 10.9.0.6.9090 > 10.9.0.5.9090: UDP, ...
```

5.3.3 UDP Amplification Attack: Turning Grenade into Missile

In this category of attacks, the attacker creates one grenade, and throw it towards B, when B replies, the payload in the reply is significantly larger than the payload in the "grenade" packet. The attacker uses A's IP address as the source IP of the packet, so B's reply goes to A. Therefore, by sending a small packet to B, the attacker successfully causes a large packet being sent to A. The attacker's power is magnified, and a grenade is turned into a missile. This attack is called UDP amplification attack.

Many UDP services unintentionally provide such an undesirable magnification effect: their response packets are significantly larger than the request packets. The size ratio between the response and the request packets is called bandwidth amplification factor. An investigation of the amplification factor for various UDP services was conducted by Rossow [2014]. The study shows "that a number of protocols are susceptible to bandwidth amplification and multiply the traffic up to a factor 4670. In the worst case, attackers thus need only 0.02% of the bandwidth that they want their victim(s) to receive." The study also shows that millions of public hosts can be abused as amplifiers. The amplification factors for some of the UDP services are listed below:

- CharGen: the average is 358.8
- DNS: ranging from 28.7–64.1. The amplification factor is larger for DNSSEC.
- NTP: ranging from 556.9 to 4670.0
- Quake 3 (a game): the average is 63.9

5.4 Summary

UDP is a quite simple protocol. Its main job is to deliver the data between two applications. In this chapter, we have discussed the port number, the UDP header format, and how the UDP client and server work. Due to its connection-less nature, UDP is often used in denial-of-service attacks. For this type of attacks, power is essential. We have shown several ways attackers can magnify their attacking powers using UDP services.

❐ Problems and Resources

The homework problems, slides, and source code for this chapter can be downloaded from the book's website: `https://www.handsonsecurity.net/`.