Advanced Network Systems and Security

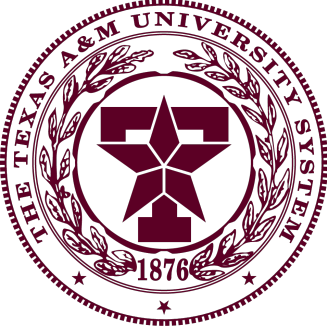
Lab Report 3

UDP Sockets and Encrypted Chat

Krish Sareen - Fernando Velez

330000230 - 527008214

February 8, 2023



**1 Introduction**

In this lab, we created a chat application over our TAMU wifi network, and added a layer of security to prevent the messages from being read over wireshark. It was a great exercise of our skills in python programming and problem solving, as well as a good use of code we wrote in previous labs, such as the encryption code and the code to write a continuous chat feature between server and host. This application ran over UDP, a protocol that differs from TCP but still runs on the transport layer. A benefit of this protocol (and in some cases a detriment) is that UDP does not have acknowledgement packets. It focusses on a constant stream of data rather than checking to resend packets because in many cases, individual packets are not as important. In this lab, we created an server-client application that worked very similarly to our first. The main differences were the aforementioned layer 4 protocol and of course the connection: Wifi.

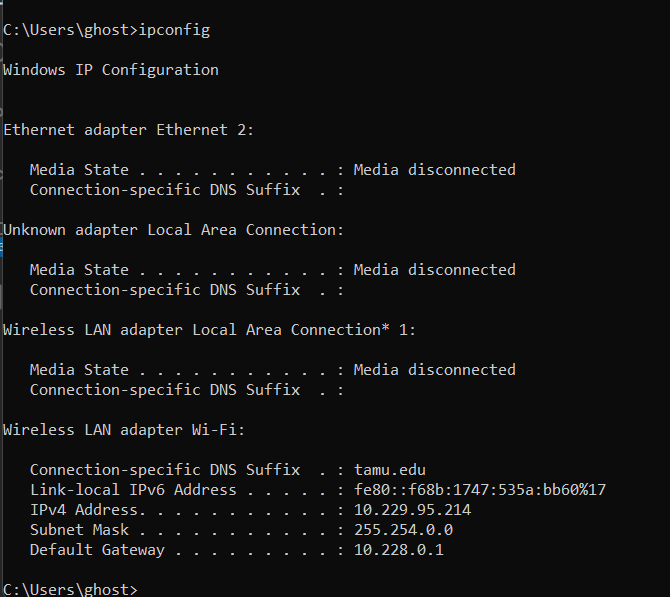
Connecting over wifi for the first time, we were now able to speak to each other without having a single cable connected or router configured. Being hosted on the TAMU network, however, required that we protect our information. To see the risk, we captured the packets that were sent by this application on wireshark, and saw that each message was completely readable in the data payload section of our packets. To prevent a breach of data, we exersized our knowledge of encryption by encrypting it in 2 ways. The first was using a caesarean cipher, which shifts characters along the ascii “wheel” by a predetermined number (key) by adding directly to their binary values. The second was a xor cipher, that toggled the bits of every character byte with a predetermined key character, such as ‘k’ or ‘%’. Both of these ciphers are symmetric, meaning one key is all one needs to decrypt the code. In a caesarean cipher, the code can be decrypted by subtracting the shift value from every byte, returning them to their original value. A xor cipher, however, is a rarity because it is truly symmetrical; the same operation can be done on the ciphertext to make it toggle back to the original string / byte. This is because of the characteristics of the XOR operation. This meant that we only wrote 3 functions to encrypt/decrypt: 2 for caesar, and 1 for xor. In this report, we detail our findings after using these encryption styles.

**2 Results**

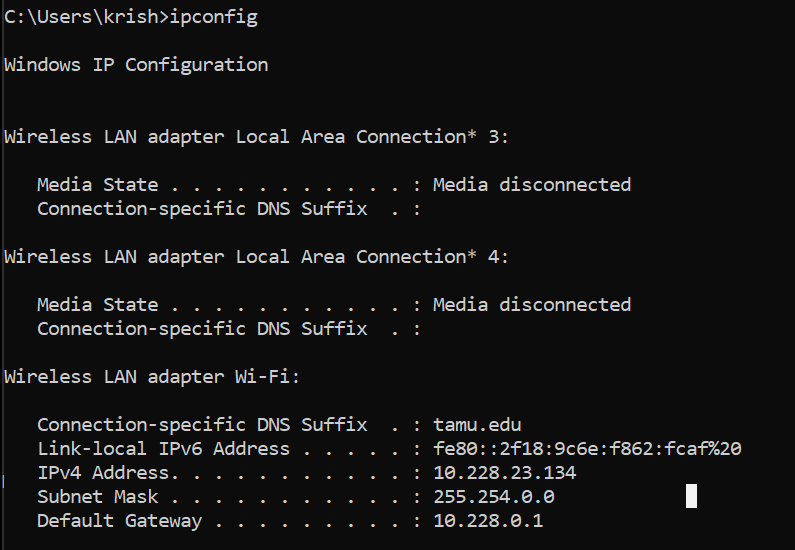
For the first task, we first found our IP addresses by typing “ipconfig” in the command prompt of our laptops. We had the following IP addresses seen below and in Figures 1-2.

**IPv4 Address for Client:** 10.229.95.214

**IPv4 Address for Server:** 10.228.23.134

****

**Figure 1:** IP address of the client.

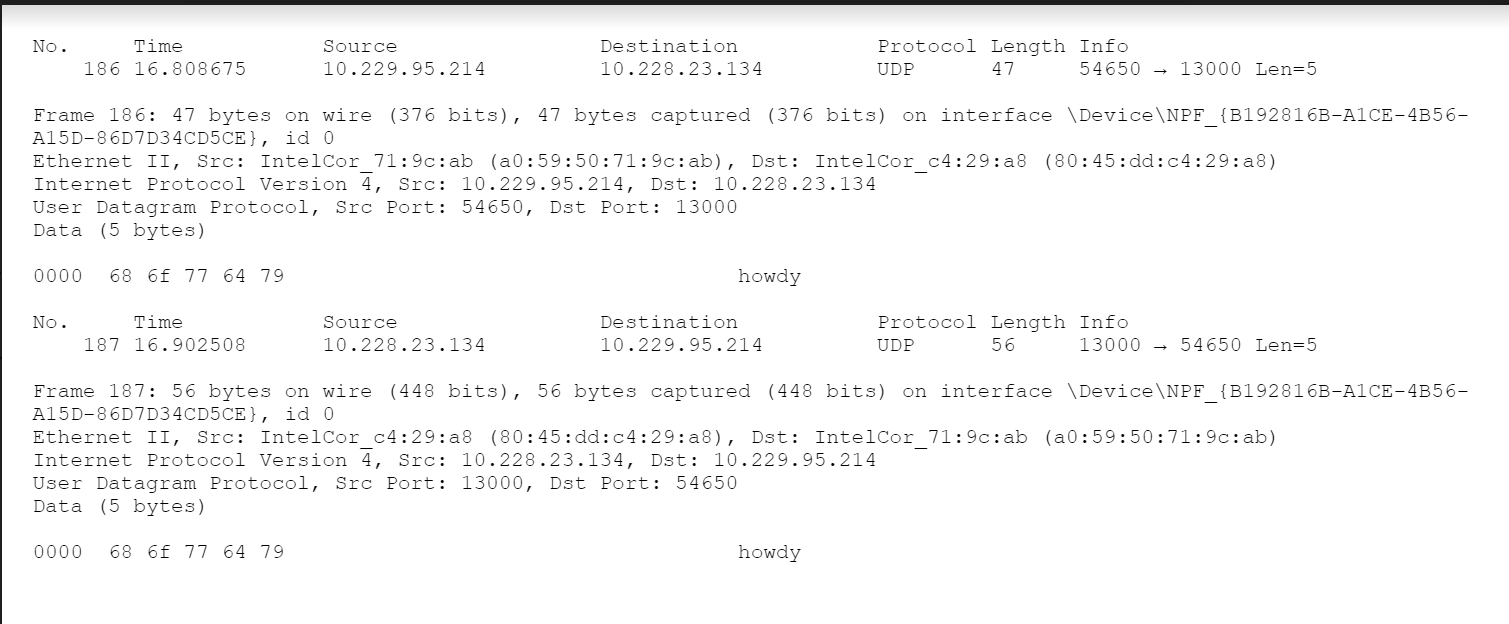


**Figure 2:** IP address of the server.

These IP addresses were then added into the provided Python code for both the client and the server. After that, we began running them and saw that the server and client could communicate with each other. At this point, they would each send one message to each other.

**2.1 Question 1** - First Wireshark Packet

When we looked at the Wireshark capture, it was observed that there was no 3 way handshake. In fact, there were only 2 packets that were captured related to the interaction. Both of these packets were just the data, in this case the messages, sent from the server to the client or vice versa. The first packet captured can be seen below in Figure 3, where the client sent the message “howdy” to the server. Unlike TCP, there were no acknowledgement packets, and the packets that *were* sent were quite smaller, because UDP is a much simpler protocol that focusses more on quick transmission of information.

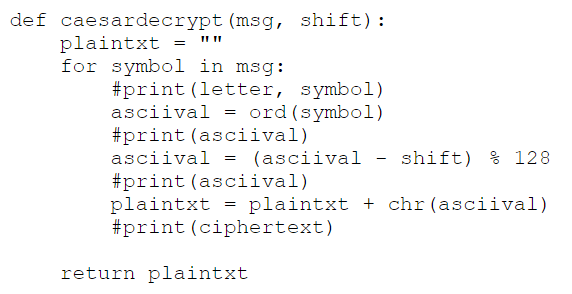
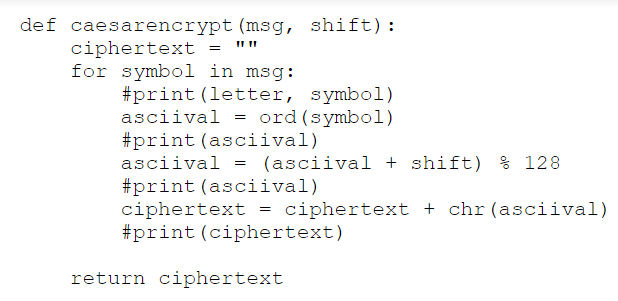


**Figure 3:** UDP Handshake interaction (Echo of Howdy)

**Table 1:** UDP packet analysis

| **SRC:** 54650 | **DST:** 13000 |
| --- | --- |
| **LEN:** 5 | **CHECKSUM:** N/A |
| **DATA:** 0000 68 6f 77 64 79 (“howdy”) | |

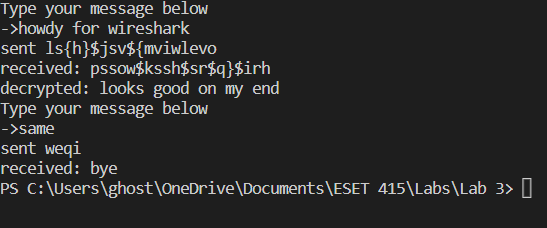
**2.2 Question 2 -** Data transfer



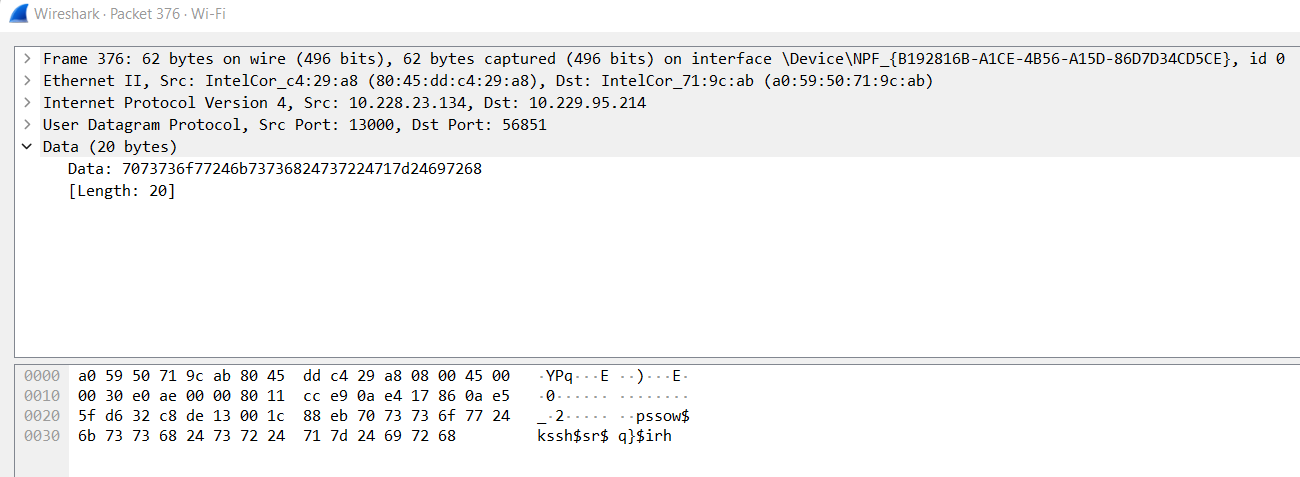
**Figure 4:** Caesar cipher encryption function **Figure 5:** Caesar cipher decryption function

In order to encrypt and decrypt, we made two different functions in Python that could perform each task. Our function to encrypt a message had two parameters, “msg” was the plaintext and “shift” was the number of times each character would be shifted. Inside the function, the ASCII value of each character in the plaintext would then be shifted to the right by the value of “shift”. This was done by adding the value of “shift” to the ASCII value. Making sure to apply modulus 128 to it as well. These new characters are put together to make the ciphertext, then the ciphertext is returned. For our decryption function, we also had two parameters. The first was “msg”, this time being the ciphertext to be decrypted, and the second was “shift”, this is still the number of times the character would be shifted. In the decryption function, the ASCII value of each character in the plaintext would then be shifted to the left by the value of “shift”. This was done by subtracting the value of “shift” to the ASCII value. Making sure to apply modulus 128 to it as well. These new characters are put together to make the original plaintext, then the plaintext is returned.

When we ran our Caesar cipher code, we decided to shift the values by 4. By the end of the connection four messages were exchanged. Therefore, four packets were sent in total. The second packet can be seen in Figure 7. This packet contained the encrypted message “looks good on my end” sent from the server. What the client received was “pssow$kssh$sr$q}$irh” as seen in both Figure 6, which contains the actual conservation, and Figure 5 which is the actual packet itself. The client then replied with “same”, which appeared as “weqi” on the server end of the connection. The packet containing this message can be seen in Figure 6.

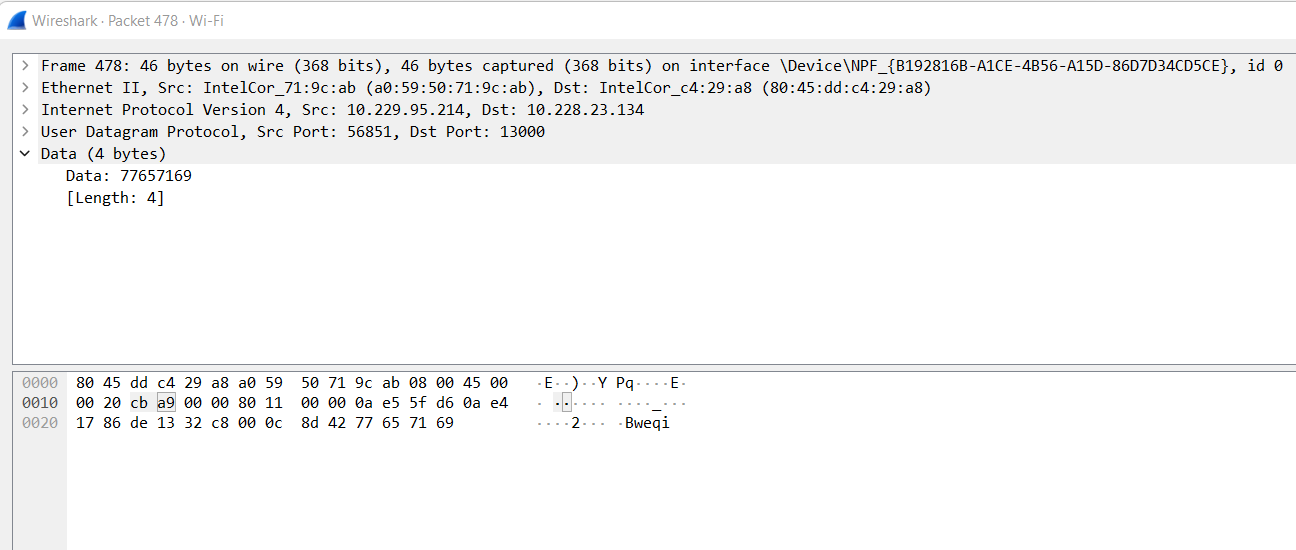


**Figure 6:** Conversation with Caesar Cipher from the point of view of the client.



**Figure 7:** Second packet sent

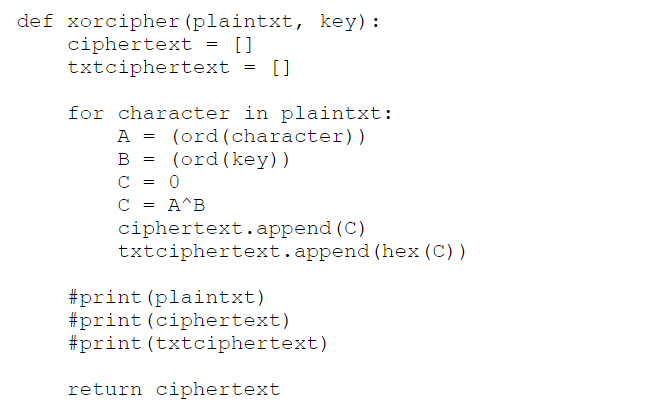
This packet on wireshark is the message “looks good on my end” from the server. We can tell this because we know what the encrypted ciphertext looks like for that message, and it matches the end of the payload in the UDP packet.



**Figure 8:** Third packet sent

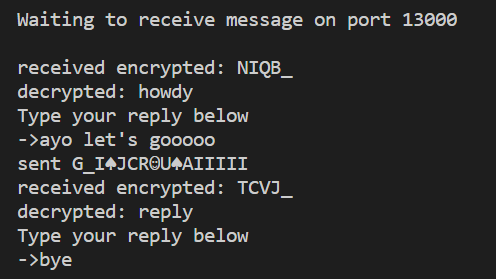
This packet is similar, as it contains the message “same” from the client to the server.

**2.3 Question 3 -** XOR Cipher



**Figure 9:** XOR cipher function

To encrypt our messages with an XOR cipher, we created an XOR encryption function. This function had 2 parameters similar to the previous Caesar cipher functions. The first parameter was “plaintext” which would be the string that would go through the XOR cipher, and “key” which was the character that would be used as the key for the encryption. Then inside the function, the ASCII value of each character in the plaintext would be XOR’d with the ASCII value of the key we used. This new ASCII value is then converted back to a character and used to create the ciphertext. Once all the characters have been encrypted, the completed ciphertext is returned. When we want to decrypt the ciphertext, we simply use the same encryption function on the ciphertext this time. As that would undo the previous encryption. The results of this encryption on our conversation between server and client can be seen below in Figure 10.



**Figure 10:** Conversation using the XOR cipher

**3 Conclusions**

In this lab, there were many protocols and skills that were used for the first time; for one, we used UDP to have fast packet transfer without acknowledgements. For another, we connected over wireless network. However, we also used many skills and concepts from our previous work to help us complete the tasks in this lab. We encrypted messages and created a messaging application between a server and a client using python. We established sockets the same was as before, except this time the layer 4 protocol was UDP instead of TCP. We connected to each other by opening the server port and making the client establish the connection, just as before. We found that the handshake to establish contact was different: UDP sent only 2 packets, whereas the previous TCP system sent 3. The difference in protocol was not tangible on the user end, but through the use of wireshark, we saw the difference.

Understanding the key differences between these protocols has given us a clearer picture of how the 4th layer of the OSI structure works, and why one protocol might be used instead of another. Another very interesting part of this lab was the security we created by encrypting our messages. Just before a message is sent to be encoded into bits and sent to the other host, we encrypt our data in 2 ways: either a caesar cipher or a xor cipher. Using these techniques, we are able to prevent others from looking at our messages (which we found to be very possible on wireshark before encrypting). We programmed our interfaces to show us what our messages looked like as we encrypted / decrypted them, and so were able to see the encrypted messages on wireshark, proving that our technique indeed worked. On the user end, it is possible to make the application so seamless that the user doesn’t even know their data is being encrypted. This is what modern applications do: on the user end, nothing changes, but on the transmission end, software is working very hard to ensure user data is not harmed or stolen. This is one of the first exercises or labs where we can see the application or our cybersecurity content, and going forward, this is a great lesson and learning experience to take away.

**A Lab Journal**

A.1 Python

List builtin functions that were used and describe them. This is an opportunity for you to keep track of things, so you can look back at them later for reference.

* chr(int number) - returns the character corresponding to the value in the ASCII table.
* ord(char) - returns the integer corresponding to the character’s position on the ASCII table
* hex(int number) - returns the hex value of the integer
* encryptcaesar(string plaintxt, int shift) - returns ciphertext shifted by caesar cipher
* decryptcaesar(string plaintxt, int shift) - returns decrypted ciphertext shifted back
* xorcipher(string plaintxt, char xorkey) - returns xor-toggled ciphertext, and if ciphertext is passed in, returns original text

A.2 Putty (configurations below are done using int e0 as an example)

* enable (en) - enter privileged EXEC mode
* config terminal (config t) - enter configuration mode
* For Cisco 2514: interface e0 (int e0) - specify the interface (in this case Ethernet0) for the Interface level
* For Cisco 4221: interface g0/0/0 (int g0/0/0) - specify the interface (in this case Ethernet0) for the Interface level
* ip address 10.10.10.1 255.255.255.0 - assign current interface (in this case Ethernet0) with an IP address and subnet mask
* no shutdown - enable the interface to send and receive packets
* end - get out of Privileged EXEC mode
* show running-config (sh run) - check the current running configuration, this can be used to check if your configurations were saved and accepted by the router.