Covert Channels

Jacob Foster

Tart Stepfather

[redacted]

[redacted]

July 19th, 2021

Executive Summary

The ACE staff tasks interns to design and develop a covert channel with the intention to exfil sensitive data from a target network. The ACE staff supplies interns with a packet capture from our target network as the bases for our covert channel. We assume there exists a default firewall on the target. We must thoroughly discuss and defend our decisions throughout the development process.

We made assumptions to constrain a solution to this problem. We assumed the data we wish to exfiltrate is text-based. We assumed we possess the ability to execute our client code on the target computer. We assumed that both the client and the server maintain their connection throughout the duration of the exfiltration process. We made these assumptions to replace unknown variables with educated guesses to constrain our solution.

Our team created a hybrid storage UDP covert channel. We chose this based on the convenience and simplicity of the UDP protocol. In addition, twenty percent of the network traffic within our target network consists of UDP traffic. Our team used metrics with respect to bandwidth, detection, permissiveness, prevention, and difficulty to implement. Our analysis suggested a UDP covert channel contains great potential. Our covert channel consists of a client and a server. We wrote our server in Python, and we wrote our client in both Python and C#. Our UDP covert channel utilizes a polyalphabetic cipher to encrypt our messages from the client to the server. We tested our covert channel and analyzed it with the previous five metrics. We created our scale from 1 to 5, with 1 as best and 5 as worst. Our covert channel received a 5 for bandwidth, 1 for detection, 1 for permissiveness, 1 for prevention, and 5 for difficulty.

1. Problem Statement

The ACE staff tasks interns to design and develop a covert channel with the intention to exfil sensitive data from a target network. The ACE staff supplies interns with a packet capture from our target network as the bases for our covert channel. We assume there exists a default firewall on the target. We must thoroughly discuss and defend our decisions throughout the development process.

2. Background

In the previous chapter, we outlined the problem. Chapter two contains broad prerequisite information necessary to understand the problem. This chapter discusses covert channels, UDP, OSI, IDS, IPS, and more.

2.1 Covert Channels

Covert channels communicate data in a non-traditional or secret manner. Often, these channels substitute the functionality of different communication protocols to disguise or obfuscate communication. [1]. This substitution may take place in many ways, and some include the addition of extra bits to outgoing packets or the replacement of certain fields within protocols to store data [2].

2.2 User Data Protocol

UDP, or user data protocol, provides a method of data transportation which prioritizes speed and simplicity. UDP possesses the ability to detect corruption in carried packets but provides no rectification methods for corrupt data transmitted [3]. Due to the low overhead and high speed, but lack of data integrity, users often communicate over UDP connections for time-sensitive applications such as voice chat or video streaming. **Figure 1** displays the configuration of a UDP packet header.

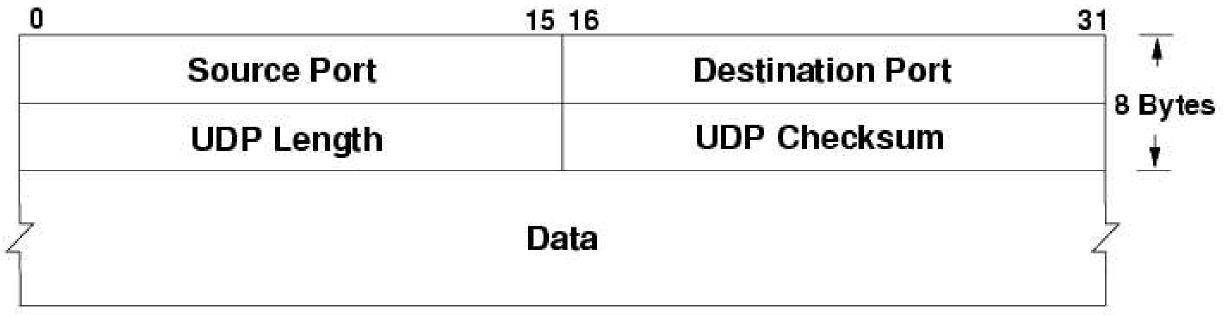


Figure 1: This figure shows the configuration of a UDP packet at the bit level.

2.3 Plaintext and Ciphertext

Plaintext represents unencrypted information. Plaintext encrypts to ciphertext when applied with a cipher. Ciphertext decrypts to plaintext [7].

2.4 UDP Sockets

Sockets provide endpoints for communication between a client and a server. A server binds a socket to a port number to listen for inbound client connections. Clients make connection attempts to the port number on the server bound to a socket [8].

In this chapter we covered the background information related to the problem at hand. In the next chapter we detail the assumptions we employ to constrain our solution.

3. Assumptions

In the last chapter we outlined background information necessary to comprehend the problem. This chapter communicates assumptions we laid as foundation to formulate our solution. We make these assumptions to replace unknown variables with educated guesses to constrain a single solution.

We assume the data we wish to exfiltrate is text-based. This assumption constrains our solution as we need not develop a means to encode and transmit files such as pictures. Rather, our solution shall focus on transmitting text.

We assume we possess the ability to execute our client code on the target computer. This assumption bounds our solution space as we need not provide a solution for the delivery of our client or the execution of our code.

We assume that both the client and the server maintain their connection throughout the duration of the exfiltration process. This constrains our solution because we need not develop a method for reconnection if the client or server loses connection with the other.

Chapter three outlines the assumptions used to bound a single solution to an unconstrained problem. The next chapter explains the tools and techniques we utilized to solve the problem.

4. Tools and Techniques

In the previous chapter we described the assumptions we made. This chapter outlines the tools and techniques we employ such as polyalphabetic ciphers, storage channels, and random value patterns.

4.1 Substitution Cipher

Substitution ciphers shift characters in a message with a key. The result of this shift produces ciphertext. For example, a key with the value of four replaces the plaintext character A with the fourth letter in the alphabet to its right. Therefore, the plaintext character A encrypts to E. To unencrypt the ciphertext, the recipient must shift the character E to the left by the value of our key. This reverse shift produces the original plaintext character, A.

4.2 Polyalphabetic Cipher

Polyalphabetic ciphers utilize multiple substitution ciphers to encrypt each character of a message with a unique key. This adds a layer of complexity makes the ciphertext much more difficult to decrypt. For instance, the plaintext message AAA may correspond to encryption keys 1,5, and 8. With these keys, the ciphertext renders as BFI.

4.3 Covert Channels: Storage Channels

Storage channels send data covertly via the alteration of content within the packet or datagram itself. One entity communicates through alteration of the data length or field, sequence numbers, headers, checksums, etc. Another entity receives and observes the changes made within the contents of the data by the sender. The changes represent obfuscated communication.

4.4 Covert Channels: Size Modulation Patterns

Size modulation patterns communicate data via the length of specific fields in a packet or datagram. For example, a server may receive information and translate the total packet length into an ASCII character. If the length of the packet equals 68 the server may interpret that as the ASCII character D.

4.5 Covert Channels: Position Pattern

Position patterns alter the location of a specific element within a packet or datagram. For example, a client may communicate an ASCII character in plaintext at the first byte of the data section in a packet. The next character it sends may be at the fifth byte, and the next character at the tenth byte. This adds a layer of stealth and obfuscation to a covert channel.

4.6 Covert Channels: Random Value Pattern

Random value patterns in covert channels introduce random values in fields to blend in with traffic that already exists. This makes it harder for a network defender to interpret our covert channel. For example, a covert channel may embed a plaintext character within the data section of a packet and surround it with random values. This way, the plaintext character blends in with the random values.

4.7 Wireshark (Version 3.4.6)

Wireshark intercepts and interprets packets from a packet capture file or from a network interface. Wireshark provides users with detailed information about packets captured or recorded.

4.8 Python (Version 3.9.6)

Python supplied developers with a scripted and object-oriented high level programming language.

4.9 .NET Framework (4.7.2)

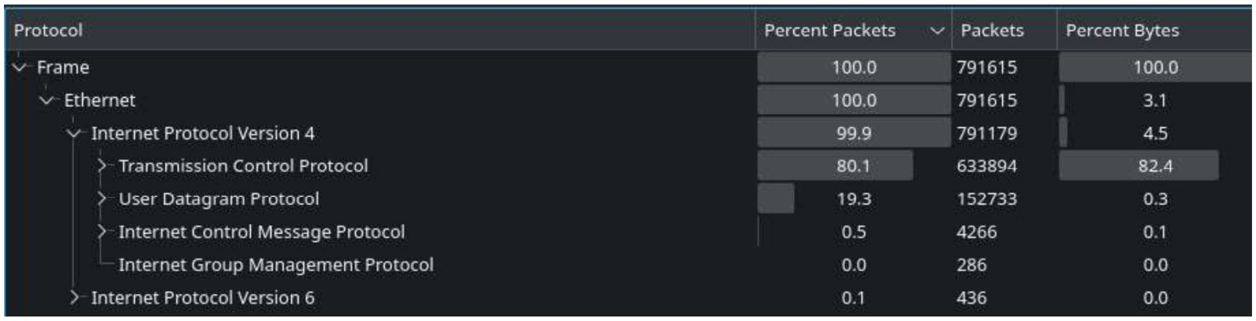
The .NET (Network Enabled Technology) Framework aids the execution of Windows execution environments. C# executes code as a high-level programming language and builds off the foundation of the .NET framework.

5. Problem Solution

In the previous chapter we discussed the tools and techniques used to confront the problem. This chapter will describe our solution to the problem and our methodology to arrive at our solution. In the previous chapter, we described the tools and techniques we utilized to solve the problem. In this chapter we describe our solution to the problem.

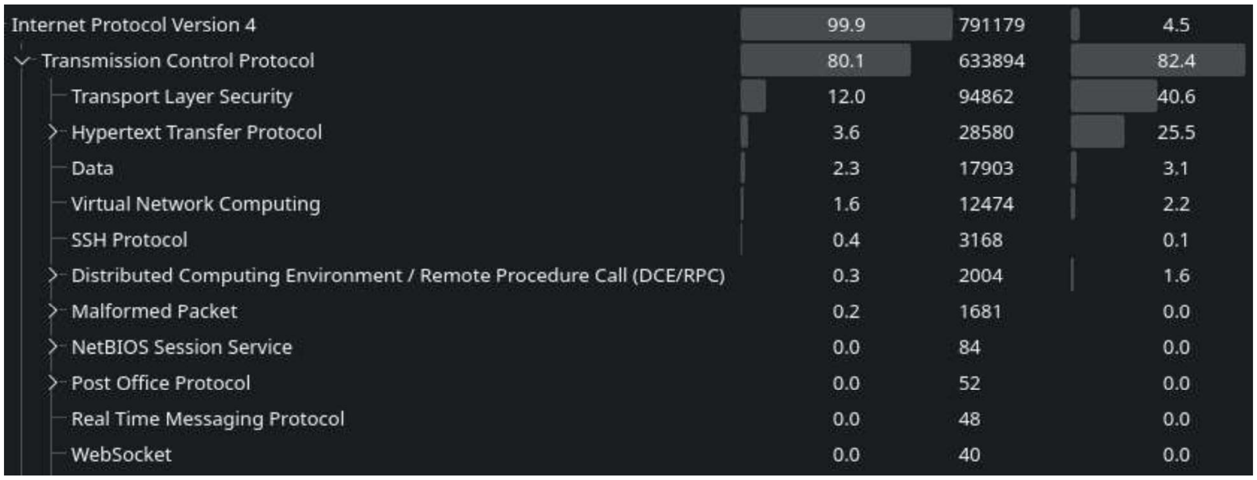
5.1 Target Network Analysis

Our team decided to analyze the network traffic provided to us. Our team opened the Exersize2.pcap file in Wireshark and utilized the statistics function to gather information. **Figure 2** displays the TCP subprotocol analysis in Wireshark.



Protocol	Percent Packets	Packets	Percent Bytes
Frame	100.0	791615	100.0
Ethernet	100.0	791615	3.1
Internet Protocol Version 4	99.9	791179	4.5
Transmission Control Protocol	80.1	633894	82.4
User Datagram Protocol	19.3	152733	0.3
Internet Control Message Protocol	0.5	4266	0.1
Internet Group Management Protocol	0.0	286	0.0
Internet Protocol Version 6	0.1	436	0.0

Figure 1: This figure shows the protocol statistics of the target network traffic.



Internet Protocol Version 4	99.9	791179	4.5
Transmission Control Protocol	80.1	633894	82.4
Transport Layer Security	12.0	94862	40.6
Hypertext Transfer Protocol	3.6	28580	25.5
Data	2.3	17903	3.1
Virtual Network Computing	1.6	12474	2.2
SSH Protocol	0.4	3168	0.1
Distributed Computing Environment / Remote Procedure Call (DCE/RPC)	0.3	2004	1.6
Malformed Packet	0.2	1681	0.0
NetBIOS Session Service	0.0	84	0.0
Post Office Protocol	0.0	52	0.0
Real Time Messaging Protocol	0.0	48	0.0
WebSocket	0.0	40	0.0

Figure 2: This figure shows the TCP subprotocol hierarchy of the target network traffic.

We listed the benefits and drawbacks to develop a covert channel with UDP, ICMP, HTTP, and TLS. We judged the protocols with five metrics. We assigned grades for each metric to each protocol based on our best estimates of the protocol and selected the protocol with the lowest sum score from the five metrics.

We constructed a decision matrix which included our four protocol options, with 1 as the best score and 5 as the worst score. We evaluated the benefits and drawbacks of each protocol before we scored each protocol. **Figure 3** displays our decision matrix and the scores we provided after discussion to each option. UDP received a score of 10, and it became the chosen protocol for our covert channel.

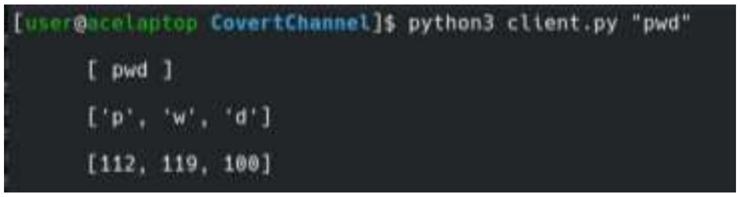
Protocol	Bandwidth	Detection	Permissiveness	Prevention	Difficulty	Total
UDP	1	2	3	1	3	10
ICMP	1	5	3	4	1	14
TLS	5	2	3	1	4	15
HTTP	3	4	3	1	3	14

Figure 3: This figure displays our decision matrix with scores.

5.2 Encrypted Covert Communications

Our team uses the length of the data section in our datagrams to communicate data. Without the obfuscation of this information, we introduce risk that a dedicated observer may detect our covert channel. Therefore, our team decided to encrypt our messages with a polyalphabetic cipher as they are sent across the network. This action increases the odds our covert channel will remain undetected.

To begin this process, we accept user supplied text-based input. This input is broken down into individual ASCII text characters. From here, the ASCII text is translated into their ASCII decimal values. **Figure J1** displays this process.



```
[user@acelaptop CovertChannel]$ python3 client.py "pwd"
          [ pwd ]
          ['p', 'w', 'd']
          [112, 119, 100]
```

Figure J1: This figure shows the initial steps taken to encrypt our messages.

Once we have translated our initial message into a list of ASCII decimal values, we generate a pseudorandom key value from between the ranges zero to fifteen. We generate a new key for each character we transmit from the client to the server.

After we generate a key, our team identifies the ASCII decimal of the first character in our message. **Figure J1** displays the first character as a *p* with the ASCII decimal value *112*. We take the value *112* and add our key to this value.

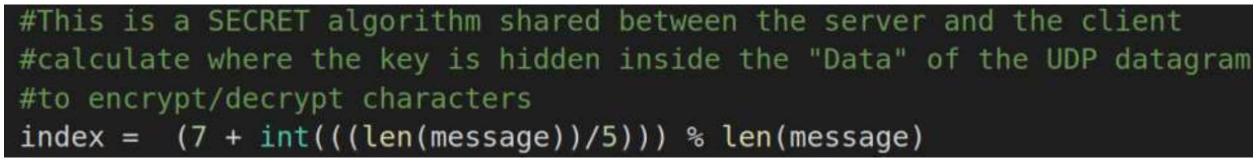
For example, let us say the generated key equates to the value *3*. We encrypt our plaintext data by adding our key with the ASCII decimal of our characters. In this instance we sum *112* with *3* which produces the result *115*.

Once we calculate the ASCII decimal ciphertext, we generate X random ASCII text characters where X is the value of our decimal ciphertext. This string of random ASCII text shall be sent as the data of our UDP datagram.

In order for the server to decrypt our ciphertext, the server must know the key our client generated to encrypt our message. The server will not know our original plaintext if the server knows not this key, the server will not know our original plaintext.

To address this problem, the client sends our key in plaintext to the server. The client embeds the key into the data section of our UDP datagrams with a position unique to the length of our encrypted ciphertext.

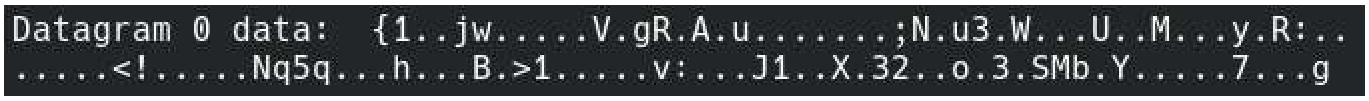
The index, or position, of where our key must be embedded is calculated with a function shown in **Figure J2.** In this scenario, our index would equal the value *(7 + int(((len(message))/5))) % len(message)*, where *message* equals the value *115*. Therefore, the index equals *30*. Our key will be embedded into index *30*.



```
#This is a SECRET algorithm shared between the server and the client
#calculate where the key is hidden inside the "Data" of the UDP datagram
#to encrypt/decrypt characters
index =  (7 + int(((len(message))/5))) % len(message)
```

Figure J2: This function calculates the position, or index, that the key must be embedded.

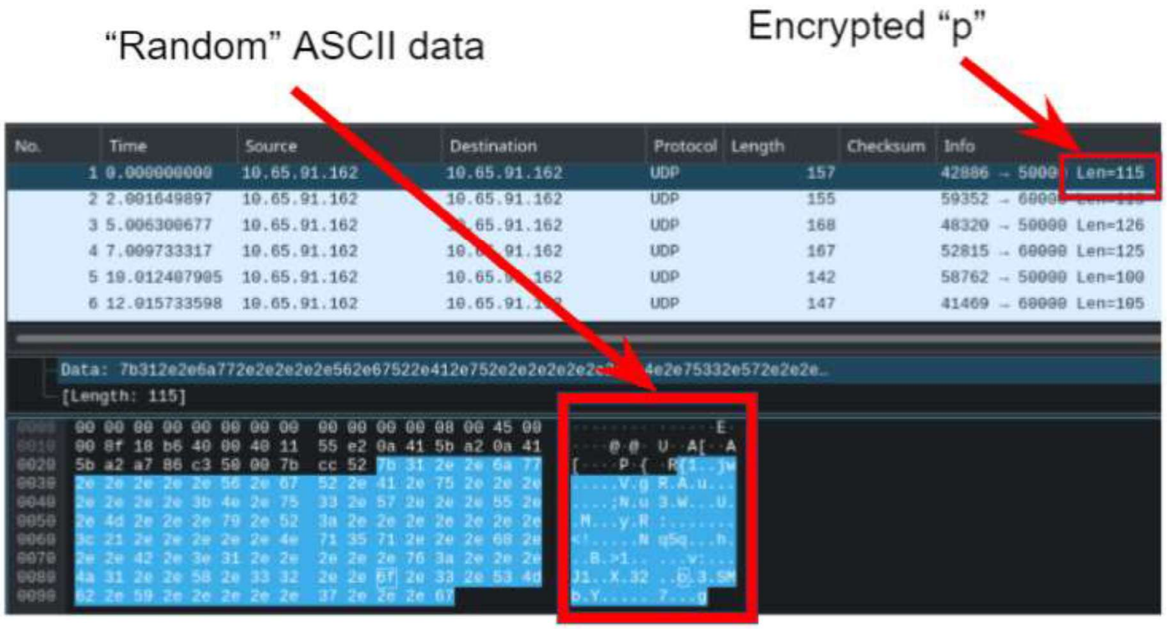
Once we have embedded the key inside our data section, we may begin to send our first datagram. **J3** reflects the contents of this datagram. This datagram possesses a length of *115* bytes. Recall, *115* is the length of our encrypted ASCII decimal value.



```
Datagram 0 data:  {1..jw.....V.gR.A.u.......;N.u3.W...U..M...y.R:..
.....<!.....Nq5q...h...B.>1.....v:...J1..X.32..o.3.SMb.Y.....7...g
```

Figure J3: This figure displays the data which shall be sent to the server within the datagram.

The client then sends this first datagram to the server. **Figure J4** displays a Wireshark capture of this traffic. Note that the blue-highlighted data in **Figure J4** holds the data as seen in **Figure J3**.



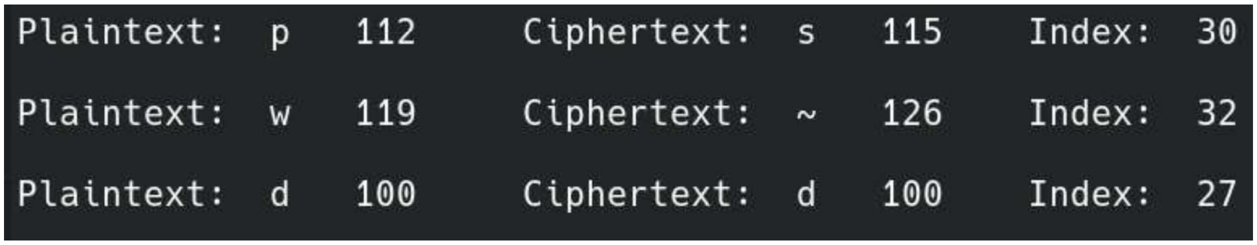
```
No.  Time          Source         Destination    Protocol Length Checksum Info
   1 0.000000000   10.65.91.162   10.65.91.162   UDP      157             42886 → 5000  Len=115
   2 2.001649897   10.65.91.162   10.65.91.162   UDP      155             59352 → 6000
   3 5.006300677   10.65.91.162   10.65.91.162   UDP      168             48320 → 50000 Len=126
   4 7.009733317   10.65.91.162   10.65.91.162   UDP      167             52815 → 60000 Len=125
   5 10.012407905  10.65.91.162   10.65.91.162   UDP      142             58762 → 50000 Len=100
   6 12.015733598  10.65.91.162   10.65.91.1     UDP      147             41469 → 60000 Len=105

Data: 7b312e2e6a772e2e2e2e2e562e67522e412e752e2e2e2e2e     4e2e75332e572e2e2e…
[Length: 115]

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ..............E.
0010  00 8f 18 b6 40 00 40 11  55 e2 0a 41 5b a2 0a 41   ....@.@.U..A[..A
0020  5b a2 a7 86 c3 50 00 7b  cc 52 7b 31 2e 2e 6a 77   [....P.{.R{1..jw
0030  2e 2e 2e 2e 2e 56 2e 67  52 2e 41 2e 75 2e 2e 2e   .....V.g R.A.u...
0040  2e 2e 2e 2e 3b 4e 2e 75  33 2e 57 2e 2e 2e 55 2e   ....;N.u 3.W...U.
0050  2e 4d 2e 2e 2e 79 2e 52  3a 2e 2e 2e 2e 2e 2e 2e   .M...y.R :.......
0060  3c 21 2e 2e 2e 2e 2e 4e  71 35 71 2e 2e 2e 68 2e   <!.....N q5q...h.
0070  2e 2e 42 2e 3e 31 2e 2e  2e 2e 2e 76 3a 2e 2e 2e   ..B.>1.. ...v:...
0080  4a 31 2e 2e 58 2e 33 32  2e 2e 6f 2e 33 2e 53 4d   J1..X.32 ..o.3.SM
0090  62 2e 59 2e 2e 2e 2e 2e  37 2e 2e 2e 67            b.Y..... 7...g
```

Figure J4: This figure shows the encrypted decimal ASCII sent over the network.

Once the server receives this datagram it makes note of the data field length. The server interprets the length is *115*. With the same algorithm as the client in **Figure J2**, the server calculates the index of the key to be at position *30*. The server parses the 30th byte of the data to retrieve the key. In this instance, the key equals *3*.

After the server parses the key, it can decrypt the ciphertext. The server calculates the original plaintext by subtracting the key from the length of the data section. For instance, *plaintext ASCII decimal = data length – key*.

The server calculates the plaintext ASCII decimal to be value *112*, which corresponds to the ASCII text character *p*. This process repeats between the client and the server until the entire message finishes transmitting. **Figure J5** displays the output of the server when it has received messages from the client.



```
Plaintext:  p   112     Ciphertext:  s   115    Index:  30
Plaintext:  w   119     Ciphertext:  ~   126    Index:  32
Plaintext:  d   100     Ciphertext:  d   100    Index:  27
```

Figure J5: This figure displays server output once it receives datagrams.

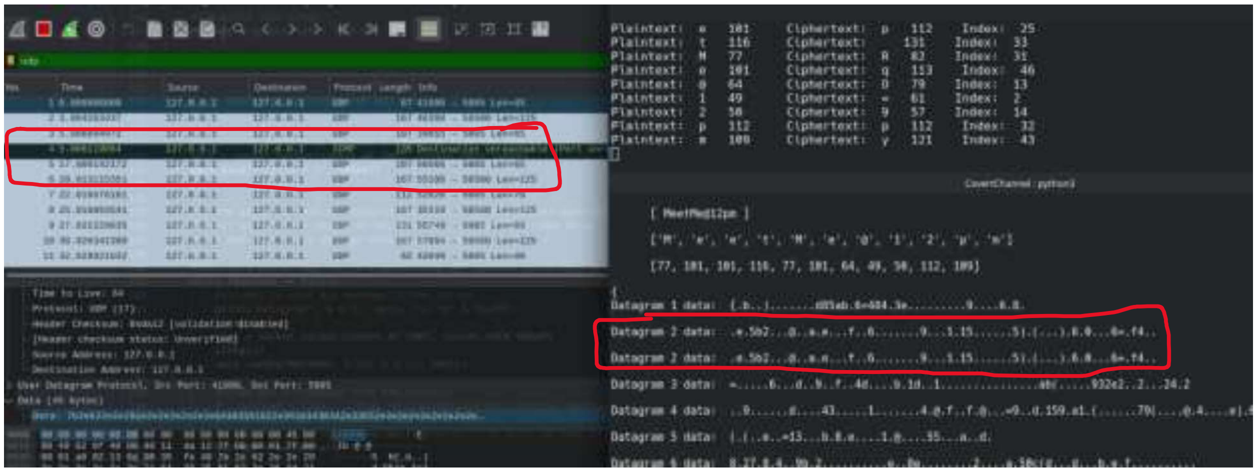
5.3 Reliable UDP

We decided to use UDP as the basis of our covert channel. UDP renders itself as a simple protocol in addition to the abundance of UDP traffic within our target network. However, we also note that UDP lacks the characteristics desired for covert channels. UDP lacks reliable data delivery and data integrity. In this section, we outline how we attempt to resolve reliable data delivery.

At this time, we possess no way to guarantee that each datagram the client sends to the server shall arrive at the server. In addition, possess no way to guarantee that the datagrams shall arrive in their original order. Due to these constrains, we decided to implement a method of data retransmission that addresses both of the above issues.

Our intention constitutes that the client shall send a message to the server. The client will listen for a server response from five to ten seconds. If the client receives no response, the client will resend its original message.

For this to work, the server must possess the ability to inform the client that it receives a datagram. We implemented this functionality within the client and the server. **Figure E1** shows this functionality. In this instance, a UDP datagram sent from the client was dropped or lost during transmission. Because the client received no response, the client resent the data to the server.



```
Plaintext:  e   101    Ciphertext:  p   112    Index:  25
Plaintext:  t   116    Ciphertext:      131    Index:  33
Plaintext:  M   77     Ciphertext:  R   82     Index:  31
Plaintext:  e   101    Ciphertext:  q   113    Index:  46
Plaintext:  @   64     Ciphertext:  O   79     Index:  13
Plaintext:  1   49     Ciphertext:  =   61     Index:  2
Plaintext:  2   50     Ciphertext:  9   57     Index:  14
Plaintext:  p   112    Ciphertext:  p   112    Index:  32
Plaintext:  m   109    Ciphertext:  y   121    Index:  43
```

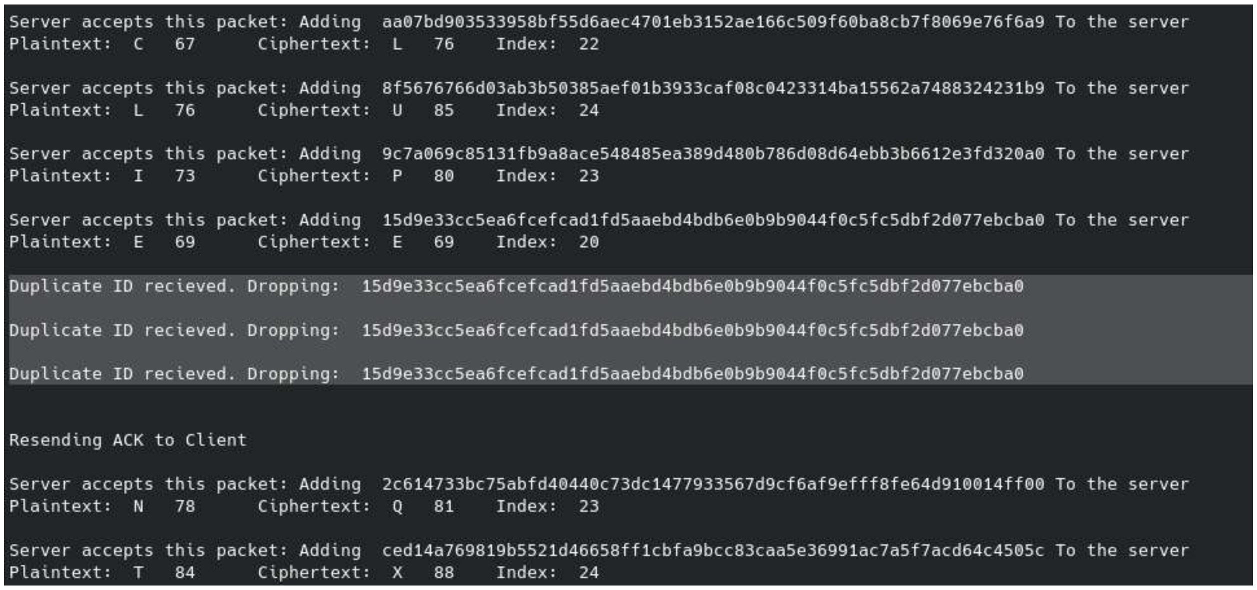
```
[ MeetMe@12pm ]

['M', 'e', 'e', 't', 'M', 'e', '@', '1', '2', 'p', 'm']

[77, 101, 101, 116, 77, 101, 64, 49, 50, 112, 109]

{
Datagram 1 data:  [illegible]
Datagram 2 data:  .e.5b2...@..a.a...f..6........9...1.15.......5].(...).6.0....6=.f4..
Datagram 2 data:  .e.5b2...@..a.a...f..6........9...1.15.......5].(...).6.0....6=.f4..
Datagram 3 data:  [illegible]
Datagram 4 data:  [illegible]
Datagram 5 data:  [illegible]
Datagram 6 data:  [illegible]
```

Figure E1: Shows successful retransmission of datagrams that failed to arrive at the server.

This solution introduced another problem. Our team pondered upon what would happen if the client sent multiple messages to the server. If the client sent three identical messages to the server and the first message arrived to the server after the second our server would accept the duplicate messages.

To fix this out-of-order delivery, we must possess the ability to identify datagrams we received and received not. Our team accomplished this through hashes. Once the server receives a message from the client the server hashes the data field of that datagram. This hash serves as a unique identifier for datagrams and is stored in a table.

The server dops the packet if it receives a datagram and the hash of that message equals another hash stored in our table of unique identifiers. **Figure E2** displays the server dropping duplicate packets.



```
Server accepts this packet: Adding  aa07bd903533958bf55d6aec4701eb3152ae166c509f60ba8cb7f8069e76f6a9 To the server
Plaintext:  C   67      Ciphertext:  L   76    Index:  22

Server accepts this packet: Adding  8f5676766d03ab3b50385aef01b3933caf08c0423314ba15562a7488324231b9 To the server
Plaintext:  L   76      Ciphertext:  U   85    Index:  24

Server accepts this packet: Adding  9c7a069c85131fb9a8ace548485ea389d480b786d08d64ebb3b6612e3fd320a0 To the server
Plaintext:  I   73      Ciphertext:  P   80    Index:  23

Server accepts this packet: Adding  15d9e33cc5ea6fcefcad1fd5aaebd4bdb6e0b9b9044f0c5fc5dbf2d077ebcba0 To the server
Plaintext:  E   69      Ciphertext:  E   69    Index:  20

Duplicate ID recieved. Dropping:  15d9e33cc5ea6fcefcad1fd5aaebd4bdb6e0b9b9044f0c5fc5dbf2d077ebcba0

Duplicate ID recieved. Dropping:  15d9e33cc5ea6fcefcad1fd5aaebd4bdb6e0b9b9044f0c5fc5dbf2d077ebcba0

Duplicate ID recieved. Dropping:  15d9e33cc5ea6fcefcad1fd5aaebd4bdb6e0b9b9044f0c5fc5dbf2d077ebcba0


Resending ACK to Client

Server accepts this packet: Adding  2c614733bc75abfd40440c73dc1477933567d9cf6af9efff8fe64d910014ff00 To the server
Plaintext:  N   78      Ciphertext:  Q   81    Index:  23

Server accepts this packet: Adding  ced14a769819b5521d46658ff1cbfa9bcc83caa5e36991ac7a5f7acd64c4505c To the server
Plaintext:  T   84      Ciphertext:  X   88    Index:  24
```

Figure E2: This figure displays the server that prevents out-of-order delivery.

Our team established retransmission our UDP covert channel in addition to the prevention of out-of-order delivery of datagrams.

Our solution requires us to generate more traffic. This solution increases the chance of detection from a network defender. However, we consider the benefits of this solution outweigh the risks associated with the decision. Our team considers the risk acceptable through the achievement of reliable data delivery.

5.4 Covert Channel Metrics

Our team realized our metrics for our covert channel after development. **Figure 14** displays the scores for each category of our covert channel.

Channel	Bandwidth	Detection	Permissiveness	Prevention	Difficulty	Total
UDP	5	1	1	1	5	13

Figure 14: This figure shows our scores for our covert channel.

We scored our channel with a 5 for bandwidth. We calculated our bandwidth as 3.2 bits-per-second. We scored our channel with a 1 for detection because our channel camouflages itself. We scored our channel with a 1 for permissiveness because it meets the criteria for valid datagrams on the target network. We scored our channel with a 1 for prevention because our cover channel may be difficult to block given that it utilizes UDP.

We scored our channel with a 5 for difficulty given the complexity of our solution. We implemented reliable data delivery as well the use of symmetric keys for encryption.

In this chapter, we explained what we did to solve the problem. We also elaborated on why we took our actions and hoe we went about executing them. The next chapter circles back to our assumptions and looks at the risk of damage to our solution from incorrect assumptions.

6. Risk Analysis

In the previous chapter, we presented the solution to our problem. This chapter analyzes the assumptions made to arrive at our solution and the impact incorrect assumptions weigh on the solution.

We assumed the data we wish to exfiltrate is text-based. If the data we wish to exfiltrate is not text-based we must implement a new exfil method that can encode and transmit the file itself, not just the contents of the file.

We assumed we possess the ability to execute our client code on the target computer. If we do not possess the ability to execute our code on the target computer we must implement a way to execute it via other means.

We assumed that both the client and the server maintain their connection throughout the duration of the exfiltration process. If the client and server do not maintain their connection throughout the duration of the exfiltration process we must implement a method to resync the client and server if either temporarily lose contact with each other.

In this chapter we discussed what happens if any of our assumptions fail, and how our solution must change.

7. References

[1] C. Cilli, “Understanding Covert Channels of Communication,” *ISACA*. [Online]. Available: https://www.isaca.org/resources/news-and-trends/isaca-now-blog/2017/understanding-covert-channels-of-communication. [Accessed: 12-Jul-2021].

[2] Fitzgerald, D, “Covert Channels: Hiding Data in Plain Sight”, in Ace, 2021.

[3] P. Fox, “User Datagram Protocol (UDP),” *Khan Academy*. [Online]. Available: https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/user-datagram-protocol-udp. [Accessed: 13-Jul-2021].

[7] “Introduction to cyber security: stay safe online,” *OpenLearn*. [Online]. Available: https://www.open.edu/openlearn/ocw/mod/oucontent/view.php?id=48322§ion=1.1. [Accessed: 14-Jul-2021].

[8] “What Is a Socket?,” *What Is a Socket? (The Java™ Tutorials > Custom Networking > All About Sockets)*. [Online]. Available: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html. [Accessed: 18-Jul-2021].