Institute of Telecommunications
Course: Electronics and Telecommunications

Year of study: III

Securing Data Transmission: Cryptology, Watermarking and Steganography

Topic: ITU-T X.1362 packet processing

Date : 16/06/2021

Student : Tomasz Lejkowski 296868

Supervisor : Piotr Chołda

Introduction

This project aims at implementation of ITU-T X.1362 packet processing in Python 3.8 programming language. The main idea is to create a process where we encrypt parts of the data on the basis of the associated mask. The mask is based on provided header of the data. Necessary padding for encryption is added, along with the information about it's size. Encrypted payload along with necessary information is put into IPv6/UDP type of packet and saved into a packet capture file.

Libraries used:

- **'pycryptodome'** for AES cipher
- **'hmac'** for message authentication
- **'hashlib'** for hashing
- **'secrets'** for IV and mask generation
- **'scapy'** for IPv6 packet forging and packet capture file magaement

Main concept

At first I declare the transmission unit, which is the size of the final payload included in IPv6 packet. We declare one password for encryption, second one for the message authentication code. For the sake of presentation I use some arbitrary IPv6 addresses and UDP source and destination ports. Next up we see a declaration of the file path to create a packet capture file containing our packets. If statement checks if the file exists, and deletes it. Both passwords are encrypted using sha256 algorithm.

```
80  TU=8000-82 # transmnission declaration unit in bytes
81  password_mac = "my mac key yes".encode()
82  password = "mypassword for en".encode()
83  ip_src = "2001:db8:3333:4444:5555:6666:7777:8888"
84  ip_dst = "2001:db8:3333:4444:CCCC:DDDD:EEEE:FFFF"
85  sport = 2150
86  dport = 2155
87  file_path = "E:\MyPcap2.pcap" #path of the packet capture file to be created
88  if (os.path.isfile(file_path)):
89      os.remove(file_path)
90      print("Old file removed")
91
92  key_mac = hashlib.sha256(password_mac).digest()
93  key_aes = hashlib.sha256(password).digest()
```

Figure 1 Main concept part 1.

Next up we declare an input file and we start to read chunks of the set value transmission unit. Each time we read a chunk of data, we call a packet forge of that chunk of data.

```
my_file_movie = open("E:\TBU_mov.mov","rb") #input file
my_content = my_file_movie.read(TU) # read chunk of data
x=1
while len(my_content) > 0:
    forge_packet(my_content, key_aes, key_mac, ip_src, ip_dst, file_path, sport, dport)
    print(x)
    x=x+1
    my_content = my_file_movie.read(TU)# read chunk of data

my_file_movie.close()
```

Figure 2 Input file and main loop for packet forging.

Packet forging

When we call the packet forge function we must set up our cipher. State mode I choose to be blockchain and initialization vector (IV) with length of 16 bytes. We also add padding to our message to fit our cipher block of 16 bytes. Together with padding we save information about padding length.

```
def pad_message(mess):  # passing in in bytes
    iterations = int(0)
    while len(mess) % 16 != 0:  # length of mess modulo 16 not equal to 0
        mess = mess + " ".encode()  # padding with bytes
        iterations = iterations + 1
    return mess, iterations  # we must inform about the length of padding

def forge_packet(message, key, key_mac, ip_src, ip_dst, file_path, source_port, dest_port):
    mode = AES.MODE_CBC  # block chaining mode
    IV = secrets.token_bytes(16)  # LENGTH 16 bytes
    cipher = AES.new(key, mode, IV)
    padded_message = pad_message(message)
    pad_length = padded_message[1]
    padded_message = padded_message[0]
```

Figure 3 Cipher setup and padding funciton.

Next up I generate mask for encryption and mask for MAC the same way I generate IV. I also encrypt it at the beginning and save the encrypted versions along side not encrypted.

```
    mask_encryption = secrets.token_bytes(16)
    enc_mask_encryption = cipher.encrypt(mask_encryption)

    mask_mac = secrets.token_bytes(16)
    enc_mask_mac = cipher.encrypt(mask_mac)
```

Figure 4 Masks and encryption of masks.

```
    i = int(len(padded_message) / 16)  # how many chunks of 16 bytes lenth do we have ?
    encryption_block = b""
    for x in range(i):
        extracted_block = padded_message[x * 16:(x + 1) * 16]  # extract data
        if mask_encryption[x % len(mask_encryption)] < 127: # save for later encryption
            encryption_block = encryption_block + extracted_block

    encrypted_block = cipher.encrypt(encryption_block) #encrypt chosen blocks
```

Figure 5 Encryption extraction loop.

On figure 5 we can see a loop in which we go through each chunk and check with mask if we should save it for encryption. At the bottom we can see a line where we encrypt extracted data. Next up on fig 6 we can see reassembly of the data with the encrypted parts put back according to the mask used.

```
    encrypted_packet = b""
    for x in range(i):  # we asseble our packet with encrypted parts
        if mask_encryption[x % len(mask_encryption)] < 127:
            # should we use an encrypted part of data ?
            encrypted_packet = encrypted_packet + encrypted_block[a * 16:(a + 1) * 16]
            a = a + 1
        else:
            #or original ?
            encrypted_packet = encrypted_packet + padded_message[x * 16:(x + 1) * 16]
```

Figure 6 Asselby of partially encrypted payload.

Next up comes assembly of our payload

```
b_pad_length = pad_length.to_bytes(1, 'big')  # pad lentgh conversion
b_next_header = bytes.fromhex("FD")  # 'experimental value for IPv6'
final_payload_trailer = IV + encrypted_packet + b_pad_length + b_next_header #assembly of the payload
```

Figure 7 Assembly of payload.

```
    extraction_for_mac = mask_encryption + mask_mac
    for x in range(i2):  # we start from 0
        if mask_mac[x % len(mask_mac)] < 127:
            extraction_for_mac = extraction_for_mac + padded_final_payload_trailer[x * 16:(x + 1) * 16]

    our_mac1 = hmac.new(key_mac, extraction_for_mac, hashlib.sha256)
    digested_mac1 = our_mac1.digest()

    final_comb = enc_mask_encryption + enc_mask_mac + padded_final_payload_trailer + digested_mac1
```

Figure 8 HMAC generation.

On figure 8 we see that mask for encryption and mask for mac are also included in the HMAC generation. We extract our data and apply hmac funtion in a simmilar way to encryption. On the bottom we see a final combination where encrypted masks are used as headers, payload is present and we add mac at the end of our data.

This type of data is ready to be sent and using scapy tool I wrap it into packet capture file with given options and presets for IPv6.

```
72      final_comb = enc_mask_encryption + enc_mask_mac + padded_final_payload_trailer + digested_mac1
73
74      pckt = Ether() / IPv6(src=ip_src, dst=ip_dst) / UDP(sport=source_port, dport=dest_port) / Raw(load=final_comb)
75      c = wrpcap(file_path, pckt, append=True)
```

Figure 9 Packet appending.

Decryption is done in a very similar manner, so explanation here will be omitted, the file is included.

Summary

We use a very secure cipher along with message authentication, so where we do not need the best privacy, we can apply mask to encrypting to lower the processing power. My implementation have timers included, so If I set the transmission unit to be low, the processing takes much longer, because of the additional tokens which must be generated.

Additionally I provide a quick video presenting the program in action: https://youtu.be/5fw2JLh5vSA