Lab9 Report

2021 04 21 16:25

Alex W
1003474

task4

```
┌──(dev㉿Kali)-[~/lab/lab9/Wireless Security part 1]
└─$ aircrack-ng -w word_list.txt wpa.full.cap
```

```
                               Aircrack-ng 1.6

      [00:00:13] 64481/14344391 keys tested (4984.90 k/s)

      Time left: 47 minutes, 45 seconds                          0.45%

                           KEY FOUND! [ 44445555 ]


      Master Key     : 17 4F E9 A8 9F 52 85 FF 0B 7F A3 05 03 DB 38 93
                       75 15 D2 0B CE 17 D8 E2 EE 36 90 F0 47 B4 C5 0E

      Transient Key  : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

      EAPOL HMAC     : AE 83 8A AD 75 5C 16 1D 08 87 CD 2C F3 8C AE 60
```

a. What is the difference between Monitor Mode and Promiscuous Mode
monitor mode allows user to use their wireless network interface card ot sniff packets that are transmitting through the air, without associating with any access point. whereas promiscuous mode requires the user to associate with an access point first.

b. If the WiFi traffic is on-going, how to crack the WiFi password?
for ongoing wifi traffic, attacker still need to capture the 4way handshake, that only happens when a new client connects

to AP. in this case, a deauthenticationn attack can be used to forcefully dessociate the client from AP, foring the client to do the 4way handshake again. in the process, attacker can use the opportunity to capture the 4way handshake packets. the captured pcap can be used with wordlist in aircrack-ng for bruteforcing the wifi password.
if there is no clients around, Reaver can be used. it bruteforces WPS PIN (wifi protected setup), a 8 digit number that can also be used in place of normal wifi password. it's easy to bruteforce a 8 digit pin in typically 4-10 hours (https://tools.kali.org/wireless-attacks/reaver)

task5

```
┌──(dev㉿Kali)-[~/lab/lab9/Wireless Security part 2]
└─$ aircrack-ng WEP.cap
```

```
                              Aircrack-ng 1.6

                 [00:00:00] Tested 1514 keys (got 30566 IVs)

KB    depth   byte(vote)
 0    0/  9   1F(39680) 4E(38400) 14(37376) 5C(37376) 9D(37376) 00(37120) C3(37120) 36(36864) 3F(36864) 73(36352)
 1    7/  9   64(36608) 3E(36352) 34(36096) 46(36096) BA(36096) 20(35584) B5(35584) 3A(35328) D3(35328) 5E(35072)
 2    0/  1   1F(46592) 6E(38400) 81(37376) 79(36864) AD(36864) 38(36608) 2A(36352) 42(36352) A9(36352) EC(36352)
 3    0/  3   1F(40960) 15(38656) 7B(38400) BB(37888) 5C(37632) 4F(36608) 66(35840) 1B(35584) DE(35584) 10(35328)
 4    0/  7   1F(39168) 23(38144) 97(37120) 59(36608) 13(36352) 83(36352) F6(36352) 2E(36096) FD(36096) D7(35840)

              KEY FOUND! [ 1F:1F:1F:1F:1F ]
     Decrypted correctly: 100%
```

as shown, the decryption key is found:
1F1F1F1F1F

task6
rc4 implementation

```python
import binascii
import struct


def KSA(key):
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
```

```python
        S[i], S[j] = S[j], S[i]

    return S


def PRGA(S):
    K = 0
    i = 0
    j = 0
    while True:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        K = S[(S[i] + S[j]) % 256]
        yield K


def RC4(key):
    S = KSA(key)
    return PRGA(S)
```

to verify the correctness of the implementation, the test cases provided in skeleton.py is converted to python unit tests and run accordingly:

```python
Run | Debug | Show Log | Show in Test Explorer
def test_1():
    key = '1A2B3C'
    cipertext = '00112233'
    plaintext = '0F6D13BC'

    assert plaintext == task6_rc4.crack_rc4(binascii.unhexlify(key),
                                            binascii.unhexlify(cipertext))


Run | Debug | Show Log | Show in Test Explorer
def test_2():
    key = '000000'
    cipertext = '00112233'
    plaintext = 'DE09AB72'

    assert plaintext == task6_rc4.crack_rc4(binascii.unhexlify(key),
```

```python
    assert plaintext == task6_rc4.crack_rc4(binascii.unhexlify(key),
                                            binascii.unhexlify(cipertext))


Run | Debug | Show Log | Show in Test Explorer
def test_3():
    key = '012345'
    cipertext = '00112233'
    plaintext = '6F914F8F'

    assert plaintext == task6_rc4.crack_rc4(binascii.unhexlify(key),
                                            binascii.unhexlify(cipertext))
```

running pytest in the terminal automatically runs all the test cases

```
pytest
============================================ test session starts =============================================
platform darwin -- Python 3.9.4, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/ALEX/Documents/Term 7/50.020 Network Security/lab/lab9
collected 3 items

Wireless Security part 2/task6_rc4_test.py ...                                                          [100%]

============================================= 3 passed in 0.01s ==============================================
```

here it shows that all three test cases are passed, verifying that the implementation of rc4 is correct

picking SN=2000

wireshark print out

```
    0111 1101 0000 .... = Sequence number: 2000
▼ WEP parameters
        Initialization Vector: 0x46bcf4
        Key Index: 0
        WEP ICV: 0x8ba2536e (not verified)
Data (54 bytes)
    Data: 98999de0ce2db11eb2169a5d442143cdd0470a8832f6712745fb4ffacdcc9ff99681c1da...
    [Length: 54]
```

copying the data into python code and run the decryption process using IV, and key that's decrypted from task5, to get the overall key by concatonating

the cipherdata is also a concatonation of data + encrypted ICV

to verify the correctness of decryption process, ICV is used first, the plaintext+icv is recovered
last part of icv is extracted out from plaintext
then, the same crc32 process is used to recalculate ICV
the recalculated ICV should match that of recovered ICV

```python
# using SN=2000
IV = "46bcf4"
key = "1F1F1F1F1F"
ICV_encrypted = "8ba2536e"
cipher_data = "98999de0ce2db11eb2169a5d442143cdd0470a8832f6712745fb4ffacdcc9ff99681c1da2f8c479ef446300eaa68aaca018b6a0a985c" + ICV_encrypted

plain_data_with_crc = crack_rc4(binascii.unhexlify(IV + key),
                                binascii.unhexlify(cipher_data))

print("plain_data_with_crc:", plain_data_with_crc)

CRC_recovered = plain_data_with_crc[-len(ICV_encrypted):]
print("CRC_recovered:", CRC_recovered)
plain_data = plain_data_with_crc[:-len(ICV_encrypted)]
print("plain_data:", plain_data)

CRC_calculated = struct.pack(
    '<L',
    binascii.crc32(bytes.fromhex(plain_data)) & 0xffffffff).hex()
print("CRC_calculated:", CRC_calculated)
```

```
 ALEX@MBP  ~/Documents/Term 7/50.020 Network Security/lab/lab9  master                                         2021-04-21 22:55:01
 /usr/local/opt/python@3.9/bin/python3.9 "/Users/ALEX/Documents/Term 7/50.020 Network Security/lab/lab9/Wireless Security part
2/task6_rc4.py"
plain_data_with_crc: AAAA0300000008060001080006040001000EA66BFB69AC100001000000000000AC1000F00000000000000000000000000000000000
006B8FE49D
CRC_recovered: 6B8FE49D
plain_data: AAAA0300000008060001080006040001000EA66BFB69AC100001000000000000AC1000F0000000000000000000000000000000000000
CRC_calculated: 6b8fe49d
```

from the screenshot above, it is evident that CRC recovered and CRC calculated match, hence, the data is decrypted correctly

the plaindata is also printed in the terminal