NetSec - Exercise 05

Che-Hao Kang, Aman Azim

June 21, 2016

Task 5.1 (theoretical): Block Cipher Based MACs

Part(a)

ECB

Is it suitable as a MAC?

- No.

Why not?

- Because each block is encrypted separately there is no relation and influence between the encrypted blocks.

CBC

Is it suitable as a MAC?

- Yes.
- Because the last block of cipher text contains the influence of all the previous blocks so, loss or change of any block will reflect on the last block of cipher text.

CTR

Is it suitable as a MAC?

- No.

Why not?

- Because each block of plaintext is encrypted independently there is no influence of one block to another.

Part (b)

ECB

Is it still suitable when dealing with messages of variable length?

- Yes.

  Because each block of message is encrypted independently. Block that cannot match the necessary size can be padded with zeros.

CBC

Is it still suitable when dealing with messages of variable length?

- No.

  Because, CBC uses chaining of XOR to produce cipher text hash.

  Thus, when the final cipher block of one message that erase the trace of its previous blocks is XORed with the first plaintext block of another message and then concatenate this result to the hash result of the first message; this will cancel out the effect of the first message because of XOR property.

  Let, m=<m1,m2,m3> a message and m_n be the final cipher block of m.

  Let ,n be another message with final cipher block n_n.

  Then, if we create a new message such that, O=<(m1^n_n),m2,m3>

  Use hash function H().

  The result will be , H(n||O)=H(m) . Means, the effect of n is totally cancel out.

CTR

Is it still suitable when dealing with messages of variable length?

- Yes.

  Because, each block of message is encrypted independently so even if the length changes it will be processed separately. Block that cannot match the necessary size can be padded with zeros.

References:

http://www.tutorialspoint.com/cryptography/block_cipher_modes_of_operation.htm
https://en.wikipedia.org/wiki/CBC-MAC
http://crypto.stackexchange.com/questions/18538/aes256-cbc-vs-aes256-ctr-in-ssh
http://stackoverflow.com/questions/1220751/how-to-choose-an-aes-encryption-mode-cbc-ecb-ctr-ocb-cfb
http://johnx.blogspot.de/2010/10/aes-cbc-or-aes-ctr-mode.html

Task 5.2 (theoretical): RADIUS

Which hosts of the SecLab are involved in the authentication process?

- User IP= 10.0.0.1
- RADIUS Client IP = 10.0.0.5 (White)
- RADIUS Server IP = 10.0.0.10

1	0.000000	10.0.0.1	10.0.0.10	DNS	65 Standard query 0xa290 A white
2	0.001022	10.0.0.10	10.0.0.1	DNS	81 Standard query response 0xa290 A white A 10.0.0.5
3	0.001173	10.0.0.1	10.0.0.5	TCP	74 44005 → 443 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=61368597 TSecr=0 WS=128
4	0.001743	10.0.0.5	10.0.0.1	TCP	74 443 → 44005 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=368489962 TSecr=61368597 WS=64
5	0.001773	10.0.0.1	10.0.0.5	TCP	66 44005 → 443 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=61368597 TSecr=368489962
6	0.001941	10.0.0.1	10.0.0.5	TLSv1	355 Client Hello
7	0.002306	10.0.0.5	10.0.0.1	TCP	66 443 → 44005 [ACK] Seq=1 Ack=290 Win=30080 Len=0 TSval=368489963 TSecr=61368597
8	0.003251	10.0.0.5	10.0.0.1	TLSv1	1040 Server Hello, Certificate, Server Hello Done
9	0.003334	10.0.0.1	10.0.0.5	TCP	66 44005 → 443 [ACK] Seq=290 Ack=975 Win=31232 Len=0 TSval=61368598 TSecr=368489963
10	0.004058	10.0.0.1	10.0.0.5	TLSv1	392 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
11	0.008966	10.0.0.5	10.0.0.1	TLSv1	300 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
12	0.009251	10.0.0.1	10.0.0.5	TLSv1	156 Application Data, Application Data
13	0.027274	10.0.0.5	10.0.0.10	RADIUS	116 Access-Request(1) (id=215, l=74)
14	0.034535	10.0.0.10	10.0.0.5	RADIUS	62 Access-Accept(2) (id=215, l=20)
15	0.036722	10.0.0.5	10.0.0.1	TLSv1	172 Application Data, Application Data
16	0.036922	10.0.0.5	10.0.0.1	TCP	66 443 → 44005 [FIN, ACK] Seq=1315 Ack=706 Win=31104 Len=0 TSval=368489972 TSecr=61368599
17	0.037491	10.0.0.1	10.0.0.5	TCP	66 44005 → 443 [FIN, ACK] Seq=706 Ack=1316 Win=33152 Len=0 TSval=61368606 TSecr=368489971
18	0.037900	10.0.0.5	10.0.0.1	TCP	66 443 → 44005 [ACK] Seq=1316 Ack=707 Win=31104 Len=0 TSval=368489972 TSecr=61368606

1. In first 2 steps the user queries the DNS server for the name of the RADIUS client and the DNS server returns the name of the NAS as “white” that have IP=10.0.0.5. (the DNS and RADIUS server have same IP because they are in the same machine)

2. In step 3, 4 and 5 the user completes the 3 way TCP hand shake with the RADIUS clients. We can see that in the first 3 steps of this packet.

3. In step 6 the user sends the “Client Hello” using a signature to verify it self to the NAS. This step uses TLSv1 protocol.

4. In step 7 the NAS sends the acknowledgement of that “Client Hello”.

5. Step 8 the NAS sends its certificate to the user to verify itself with “Server Hello” and finally the message “Server Hello Done”. This step uses TLSv1 protocol.

6. In step 9 the user sends verification acknowledgement of the certificate to the RADIUS client/NAS.

7. In step 10 the user exchange the shared secret key with the NAS. Sends “Change Cipher Spec” message to negotiate the usage of this same CipherSpec. Finally sends the first encrypted message using that specific algorithm and shared secret key. This step uses TLSv1 protocol.

8. In step 11 the NAS sends the user the “New session ticket”, aggress on the usage of the same CipherSpec and also sends it first encrypted message using that specific algorithm and shared secret key to the user. This step uses TLSv1 protocol.

9. In step 12 the client sends its identity credentials in encrypted form using the negotiated CipherSpec as “Application Data”. This step uses TLSv1 protocol.

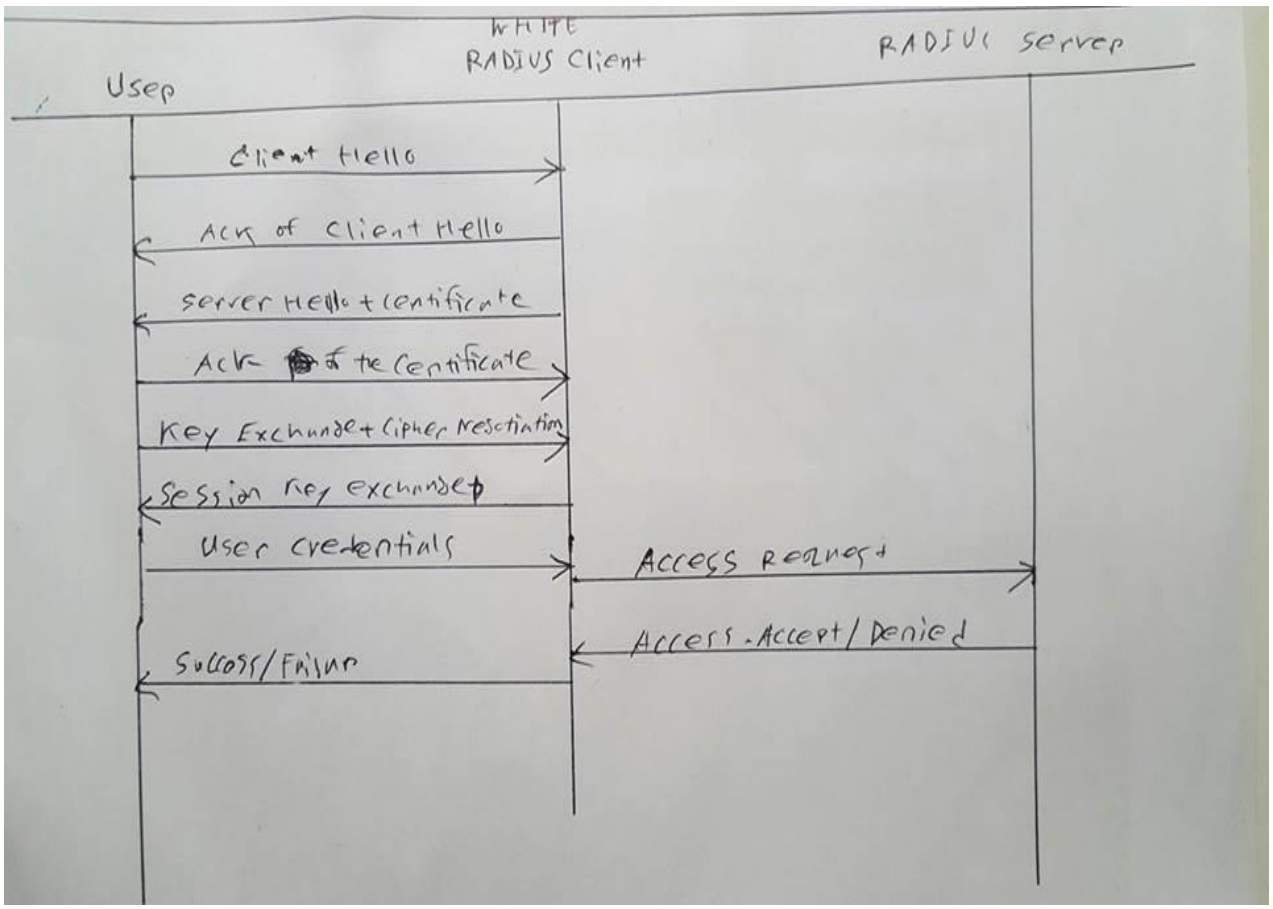
10. In step 13 the NAS sends the “Access-Request(1)” with all the necessary encrypted data and user credentials to the RADIUS server. In the attribute field of this packet we can see the user name “kang” and all other details.

This step uses RADIUS protocol.

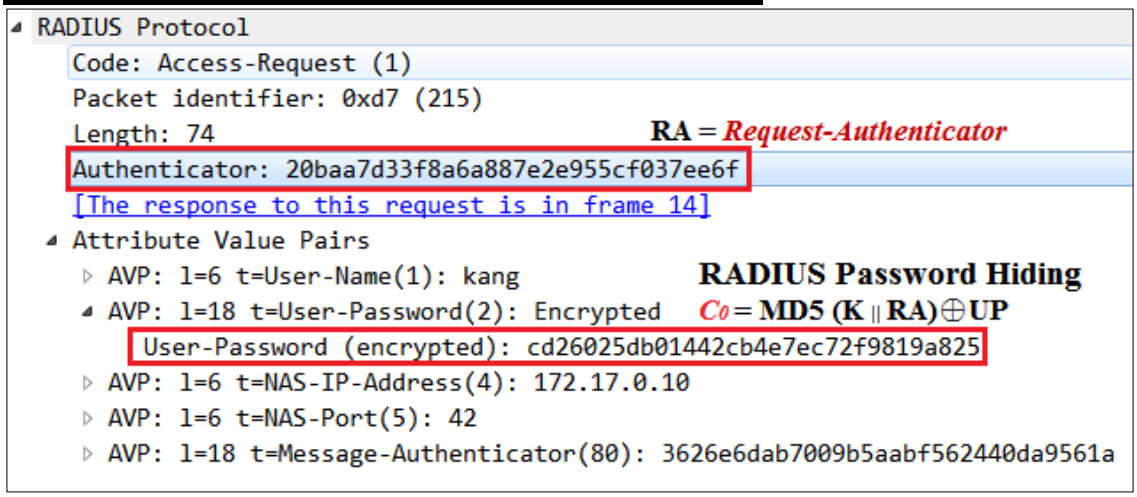
11. In step 14 the RADIUS server sends “Access-Accept” to the NAS after checking its submitted data. This step uses RADIUS protocol.
12. In step 15 the RADIUS client confirms the user about successful access by returning its credentials. This step uses TLSv1 protocol.

13. Now NAS and the user communicates normally using TCP.

    In the sketch I draw from step 6 to simplify and I used pen and paper because you asked for sketch if I am not wrong.


WLAN
RADIUS Client
RADIUS server
User
Client Hello
Ack of Client Hello
Server Hello + Certificate
Ack of the Certificate
Key Exchange + Cipher Negotiation
Session key exchanged
User credentials
Access Request
Access-Accept / Denied
Success / Failure

Task 5.3 (theoretical): RADIUS (again)



As we can see from the above picture, what we need are **Request-Authenticator** and **User-Password (encrypted)**. Besides, we use the **RFC-7511 dictionary** as keys to execute a brute-force attack with the formula $C_0 = MD5\ (K \,||\, RA) \oplus UP$.

The overall procedure is the following (All values are in **bytes** data type):

1. Pad the **plain-text password** with zero at the end **(UP)**
2. Take a word from **RFC-7511** **(K)**
3. Append this **RFC-7511** with **Request Authenticator** **(K || RA)**
4. Calculate the **MD5** by hashlib.md5 **(MD5(K || RA))**
5. XOR the **MD5** (from Step 5) with **password** (from Step 2) **(MD5(K || RA) ⊕ UP)**
6. Check if it is the same as **User-Password (encrypted)**

- The source code is in the folder and detailed comments are in the code.
- Didn't find any matched shared secret.

Task 5.4 (practical): One-Time Pad

In Linux manual (**man urandom**), it says that "the **/dev/random** device will only return random bytes within the estimated number of bits of noise in the entropy pool. **/dev/random** should be suitable for uses that need very high quality randomness such as **one-time pad** or key generation." [1] This means **/dev/random** will **block** after the entropy pool is empty. We need to wait until extra data is gathered to fill the entropy pool.

In contrast, "the **/dev/urandom** device will **not block** waiting for more entropy. As a result, if there is not sufficient entropy in the entropy pool, the returned values are theoretically vulnerable to a cryptographic attack on the algorithms used by the driver." [1] This indicates that the **/dev/urandom** will reuse the entropy pool to generate random numbers and **not block**. This is useful when randomness is not so required (e.g., generate random testing data).

- In our programming, **exercise5_4_server.py** is for server part; **exercise5_4_client.py** is for client part.

  On server part, we use **SocketServer.TCPServer** to listen for connections. Every time a client connects to the server, the request will be passed to **ServerTCPHandler**.

```
host, port = "localhost", 1234

SocketServer.TCPServer.allow_reuse_address = True
server = SocketServer.TCPServer((host, port), ServerTCPHandler)
server.serve_forever()
```

  The result is the following:

```
kang@hellgate:~$ python exercise5_4_server.py
Server encrypted "IAMYOURSERVER" and sent "WOJHGUTXVZPVN to the client
Server decrypted the client's message "CYFJEYQZIKFZAZC" to OKIAMYOURCLIENT
```

  On client part, we employ **socket.socket** to create the socket, **socket.connect** to connect to the server and **sock.sendall** to send data to the server.

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    # Connect to server and send data
    sock.connect((host, port))
    encryptedMessage = encryptMessage("OKIAMYOURCLIENT")
    print "Client encrypted \"OKIAMYOURCLIENT\" and sent \"" + encryptedMessage + "\" to the server"
    sock.sendall(encryptedMessage + "\n")
finally:
    sock.close()
```

  The outcome is the following:

```
kang@hellgate:~$ python exercise5_4_client.py
Client received an encrypted message: WOJHGUTXVZPVN
Client decrypted the message to: IAMYOURSERVER

Client encrypted "OKIAMYOURCLIENT" and sent "CYFJEYQZIKFZAZC" to the server
```

  Both use **generateOneTimePad** to produce a one-time pad if not exist yet, **encryptMessage** to encrypt a message and **decryptMessage** to decrypt a message (Detailed comments are in the code).

Reference:

1. Linux manual - **RANDOM(4) (man urandom)**

Task 5.5 (practical): HMAC

I choose **SHA3-512** because:
1. It is the newest version of SHA (released by NIST on August 5, 2015[1])
2. It uses a different approach called **Sponge Construction**[1] for hashing and its internal structure varies from the other SHAs.

Therefore, it is not easy for hackers to compromise the security of **SHA3-512**.

- We need to use **$HMAC_k(m) = h((k \oplus opad) || h((k \oplus ipad) || m))$** to generate the HMAC. Because the block size of **SHA3-512** is **72 bytes** and the **key** only has **10 bytes**, we have to pad the key with leading zeros. (Detailed comments are in the code)
- The HMAC for this PDF document is

```
b'p\xf5zz\xf5\x0b\xee\x04D\xecH\x1d\xa52\xda\xc9\x17I\x8d*\x08=p\x8bdax\xef\x86\x83\\\xa6-\x81\xd7\xc2\xd9\x82\xa1\xba
\xbb\x03l#\x1c\x9f\xaa\x894\xb6\xf7\x98\xb3\x15\x01\xb6t\x9a\xb8\xdf\xfc$\t\xdf'
```

Reference:

1. Wikipedia (SHA-3) - **https://en.wikipedia.org/wiki/SHA-3**