# Assignment 5

Abbas Khan , Mariia Rybalka , Linara Adilova

June 21, 2016

## Task 5.1 (theoretical): Block Cipher Based MACs

### Part (a)

- ECB gives encrypted with a shared secret blocks of a plain text. So basically this works as MAC for showing that the sender is the one, who knows shared secret and it can prove integrity because receiver can just decrypt the message and compare with received plain text. But by definition MAC has to be a short tag, and here it will be at best as long as the message itself. Also it is easily breakable because it saves the structure of the plaintext.

- CBC is usually used for generating MAC, by sending the last block. It will contain information about all the message and will be changed if the message is changed and also it will be encrypted by the shared secret, so it supports authenticity. The problem is with lots of known attacks, such as

- CTR is better than CBC in means of possibility to parallelise calculations, but like with ECB we need the whole encryption to support integrity, because here again all the blocks are different. It supports authenticity because of using shared secret for encryption.

### Part (b)

- In ECB mode padding can make it even easier to find out the shared secret - if the last block will be a constant padding attacker can guess the structure of the padding and the key.

- In CBC mode padding adds ability to attack it also - because it is easy to get the new message, that will have the same MAC. One just have to know two pairs of plain text and MAC and then he can use it to construct new plain text with same MAC.

- In CTR mode padding does not really change the situation, as we XOR last padded block with random counter and nonce.

## Task 5.2 (theoretical): RADIUS

Used material - [1].
From the traffic dump, taken during running 'authme' command, can be concluded that the hosts that take part in the authentication process as RADIUS server 10.0.0.10 (orange) and as RADIUS client 10.0.0.5 (white). I was able to found packets with protocol RADIUS, one is from 10.0.0.5 to 10.0.0.10 and one packet back. In the information about the packets I could find following:

- UDP packet, information concerned with RADIUS protocol: code of the packet is Access-Request, identifier of the packet and Authenticator code - random 128 bit string, used as salt for securing the password of the user, passed in the packet. Attribute Value Pairs in the packet are following: User-Name that has requested for authentication (plain text), User-Password in encrypted form, NAS-IP-Address (Network Access Server = NAS), NAS-Port, Message-Authenticator field.
- Also a UDP packet, with code Access-Accept and the same identifier as in the request packet. Also the Authenticator code is contained in it.

If try to send not existing username or wrong password, the code of the answer packet will be Access-Reject.
Before RADIUS interaction 10.0.0.1 (hellgate) establishes TLS connection with Authenticator (RADIUS client), in our case 10.0.0.5 (white). This also can be seen from the traffic dump. Every time when 'authme' is run this connection will be established.

## Task 5.3 (practical): RADIUS

Data needed for brute-force attack on the shared secret is one Access-Request packet. As we know the password sent and we can get request Authenticator code from the packet, we can use User-Password attribute for trying to find shared secret. User-Password attributed created as following (in case of password shorter than 16 octets):

$$zero\_padded\_password\ XOR\ MD5(secret \mid\mid authenticator)$$

We used just '0000000000000000' (16 zeros) as a password, in order to get needed length of the password - no need to pad it with zeros and split it to 16 octets (if password is longer than 16 octets process of securing is changed, password will be broken to chunks and encrypted in chain). So we take the dictionary (RFC 7511 as noted in hint) and try every word from it.
Source code is in **break_radius.py**. Captured traffic is in **traffic_task2.pcap**.
Found shared secret is .

## Task 5.4 (practical): One-Time Pad

Linux kernel maintains an entropy pool, which it fills with environmental noise which it collects via device drivers or other sources.The problem is that once that entropy pool is used up /dev/random blocks until it can get more noise to fill up its entropy pool. One alternative can be /dev/urandom which uses CSPRNG which is seeded from the entropy pool. It does not suffer from blocks but is considered less secure compared to /dev/random.

Following are the functions which perform the required tasks
In server.java

- run()
- ChatActionPerformed
- getRandom

And for client

- run()
- sendActionPerformed

Please run server.java and than client.java.Type in the message in the lower textArea ,it will be encrypted and sent to the other , who in turn will decrypt it and the result will appear on the top textArea.The code uses library which can be found in the libraries folder.

## Task 5.5 (practical): HMAC

We heard on the lecture that strength of HMAC depends on used hash function. We can use SHA-1 or SHA-256, and the key length will be 512 bits (64 bytes) (block size). If we use SHA-384 or SHA-512, we will need key of length 1024 bits (128 bytes), that is definitely overhead. But if SHA-1 and SHA-256 need the key of the same length, no need to peek weaker one. So our choice is SHA-256. By the way, even SHA-1 is stronger than MD5, so MD5 was not even a candidate.
Our output is
**d93292b2969aad6821c0f2dbaf7e4704672f27f45d39da9d763fbd956de3da6a**.
Source code is in file **hmac.py**.

## References

[1] Remote Authentication Dial In User Service (RADIUS) *https://freeradius.org/rfc/rfc2865.html*