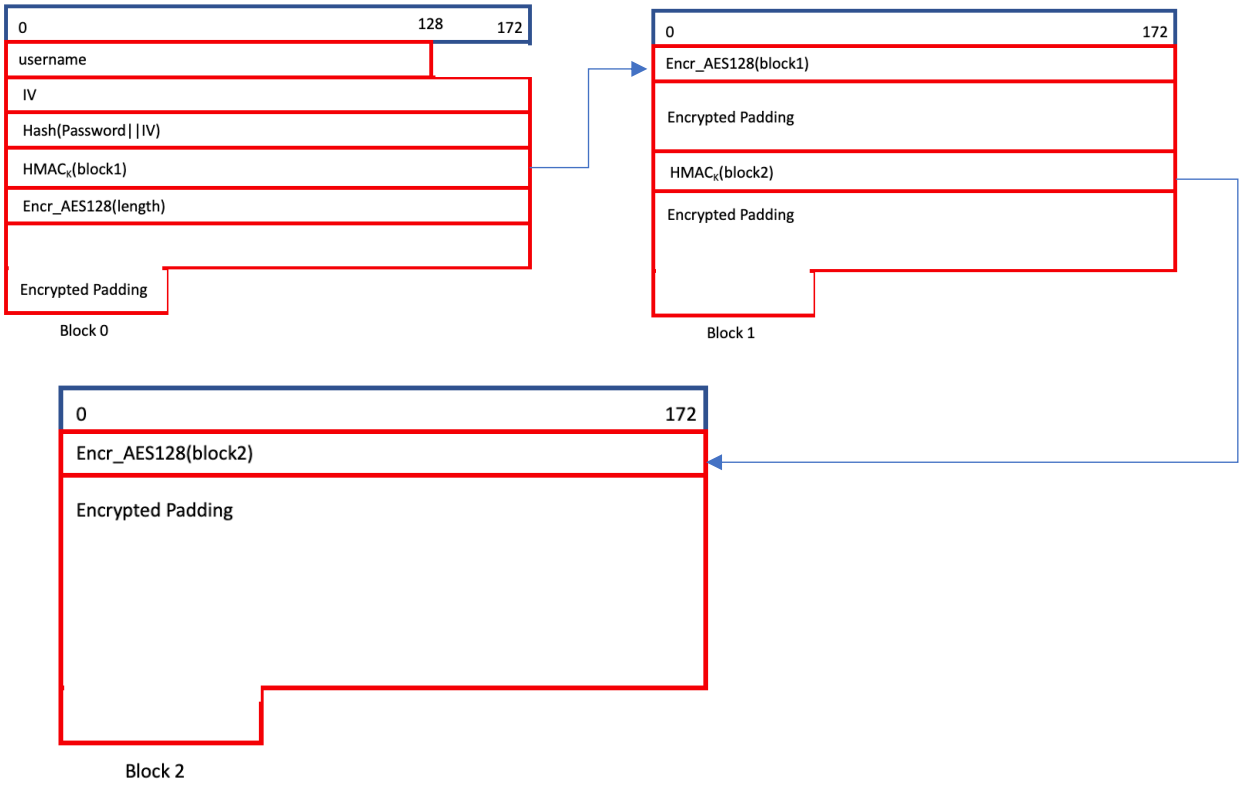
Kunal Mukherjee
CS 6324
Dr. Chung Hwan Kim
03/22/21

3.1 Design explanation

- Meta-data design



  - Block 0 – e.g. "/awesome.txt/0"- contains the meta data pertaining to the entire message in "awesome.txt"
    - 0-127 bytes – username
    - 128-299 bytes – IV
    - 300-471 bytes – hash of salted password
    - 472-643 bytes – Encrypted HMAC for 128-byte message in block 1
    - 644-815 bytes – Encrypted length for whole message in "awesome.txt"
    - 816-1023 – Encrypted padding

  - The entire message is partitioned into 128-byte. The message is padded incase the message length is not a multiple of 128. For example, 130-byte message will be padded to 256 bytes.
  - Let each 128-byte message be m_i, and it is stored in block i – e.g. first 128 bytes of "awesome.txt" is stored in "/awesome.txt/1"
  - i = [1: x], where x = total number of blocks = (message.length() / 128) + 1.
    - *start_index*=(i-1) *128.

- Each block will contains m_i = message[*start_index: start_index* + 128], and the HMAC for the message m_i that is stored in block i+1, (if block i+1 exists)
  - "/awesome.txt/1" will contain encrypted m_i, and HMAC for m_{i+1} that is in block 2
  - 0-171 bytes – encoded encrypted padded m_i
  - 172-471 bytes – encrypted padding
  - 472-643 bytes – encoded encrypted padded HMAC for m_{i+1} stored in block i+1, if block i+1 exists
  - 644-1023 bytes – encrypted padding

- User authentication
  - Password is stored using the salted method. So, [salt, hash(password||salt)] is stored. Salt is the IV in our case.
  - To store the password
    1. Get the password string
    2. Pad the password string to 128 bytes, if password is less than 128 bytes
    3. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
    4. Create the 256 bytes of salted_password = password || IV
    5. Get the 256-bit hash of the 256 bytes salted password
       - Use the function Hash_SHA256(salted_password)
    6. Pad the 256 bits output of the hash of the salted password to 128 bytes
    7. Store the 172-bytes encoded padded hash in block0's 300-471 bytes
       1. Use Base64.encode() to encode the padded hash
       2. We need to encode as we do not want to leak information about our encryption language or the language the password is put in
  - To authenticate the password
    1. Get the hash of the salted password stored in block 0
       1. Get the encoded padded hash of the salted password from bytes in 300-471 in block 0
       2. Use Base64.decode() to decode the 172 bytes to the 128 byte of padded hash of the stored salted password
       3. Remove the padding to get the 256-bit hash of stored salted password
    2. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
    3. Get the current password user entered, and rerun the steps of "To store the password" but stop in step 5, to get the 256-bit hash of the current salted password
    4. Match the 256-bit hash of the current salted password with the 256-bit hash of salted password in block0
       - If they are the same, correctly authenticate
       - If not, the password mismatched

- Encryption design

    1. The entire message is partitioned into 128-byte. The message is padded incase the message length is not a multiple of 128.
    2. Let each 128-byte message be m_i, and it is stored in block i – e.g. first 128 bytes of "awesome.txt" is stored in "/awesome.txt/1"
    3. current block, i = [1: x], where x = total number of blocks = (message.length() / 128) + 1.
    4. *start_index*=(i-1) *128.
    5. Each block will contains, m_i = message[*start_index: start_index* + 128]
    6. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
    7. Each 128-byte message m_i is encrypted using AES_128
        1. Each m_i is partitioned into 8 128-bit messages, m_{i_128}
            - Each m_{i_128} is encrypted using AES_128 in CTR mode
            - The last byte of IV is incremented every time, as we are using AES-128 in CTR mode
        2. 128-byte encrypted m_i, e_m_i, is encoded using Base64.encode() to get 172-byte encoded e_m_i
        3. Finally, store the 172-byte e_m_i into the block i's 0-171 bytes
    - This design ensures security because even though the adversary can read stored version in the disk, they will never see the message in plaintext. Since, we are using AES in CTR mode, we are making sure each block is encrypted in a new way, every time it is encrypted, so there is no determinism, and we have randomized encryption.

- Length hiding
    - As mentioned in encryption design, since each message is broken down into 128 bytes blocks, so two files of length 1160 bytes and 1260 bytes both would need 9 (1160 bytes /128 bytes) physical files, then an adversary should not be able to tell which is the case. Thus, without looking at the actual message it is hard to estimate the length. The design does not leak any additional length information other than that the length can be anywhere in the range, (x*128) +/- 128 bytes, where x is the total number of blocks.
    - For example, if the attacker can see that 2 blocks are created, they would know the length of the file is between 128-254 bytes. So, they will have a 1/128 probability of guessing the length correctly.

- Message authentication

    - MAC for each 128-byte block of message, m_i
        - 472-643 bytes in block i, where i = 0 to x-1, contain the encrypted $HMAC_K$ for the m_{i+1} in block i+1. (x is the total number of blocks)
        - The HMAC is calculated:
            1. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV

2. Get the password from user (need to verify the password and proceed only if verified)
3. each 128-byte message be m_i, and it is stored in block i
4. current clock, i = [1: x], where x = total number of blocks = (message.length() / 128) + 1.
5. *start_index*=(i-1) *128.
6. Each block will contains, m_i = message[*start_index*: *start_index* + 128]
7. Create the $HMAC_K[m_i]$
    - $HMAC[m_i]$ is 128 byte padded output of Hash((K⊕opad) || Hash((K ⊕ ipad) || m_i))
    - ipad = the byte 0x36 repeated (256-bit/8) 32 times
    - opad = the byte 0x5C repeated (256-bit/8) 32 times.
    - K=padded key = (password || hash)
    - 256-bit hash output is padded to get 128-byte padded hash
    - 128-byte padded hash is partitioned into 8 128-bit blocks
    - Each 128-bit block is encrypted used AES-128
8. 128-byte encrypted $HMAC_K[m_i]$ is encoded using Base64.encode()
9. Store the 172-byte encoded encrypted $HMAC_K[m_i]$ into 816-987 bytes in block i-1

- To verify
    1. For each 128-byte message block, match the HMAC of the i^{th} block with HMAC stored 472-643 bytes in block i-1
    2. If both verification pass, only then verify the document

- Efficiency

    - My design does not the best storage efficiency, by a margin of 127 bytes in worst case, but has good speed efficiency. As I am encrypting messages in blocks of 128 bytes, so if a message is 129 bytes, then for my implementation, it would need an extra 1024 bytes block for just for just 1 extra byte of message.
      Since, I have broken the messages into 128 byte in each block, updating or writing new message means, I only need to update the blocks that contain the part of the message that needs updating. So, if I have to update a sub-string starting from let's say 515-600, I would only need to update block 3 and 4, and no need to touch blocks 0-2. Also, this works the same for reading also. So, to be able to read I can use the *starting_position* to just decrypt the needed block(s), as we don't need to decrypt any extra blocks to get the precise substring.
    - Max Storage Efficiency
        1. The implementation will model a type of stream cipher. We will generate message.length() bytes of random IV for the entire message and encrypt the entire message with a key. Since, we are not using any extra space for padding, we will have the best storage efficiency. Or in other words, we will use the entire block space available to us to store the message. We can store the HMAC and length in block0. Another modification we can make

is that we can use part of the block0 to store the encrypted message. So, that we are using all the blocks.

- Max Speed Efficiency
    1. Put one byte of message in one block. So, updating a message means updating only those blocks. But that would mean, for 10-byte message we need 10 blocks. But for updating, writing and reading x bytes, we can do the operation in O(1) as we can find the byte number in constant time and just access that block without any overhead.

3.2 Pseudo-code

A. create

- Create a block 0 using the filename of 1024 bytes and have its buffer filled with random values
- Store the username to block 0 in 0-127 bytes
- Create the IV and store it in bytes 128-299 bytes
    1. Get 128 bytes of random values using the secureRandomNumber(128)
    2. Encode the 128-bytes IV using Base64.encode()
    3. store it in bytes 128-299 bytes of block 0
- Create the password and store it in block0's 300-471 bytes
    - Password is stored using the salted method. So, [salt, hash(password||salt)] is stored. Salt is the IV in our case.
    - To store the password
    1. Get the password string
    2. Pad the password string to 128 bytes
    3. Get the IV from 128-299 bytes from block 0
    4. Create the 256 bytes of salted_password = password || IV
    5. Get the 256-bit hash of the 256 bytes salted password
    6. Pad the 256 bits output of the hash of the salted password to 128 bytes
    7. Store the 172-bytes encoded padded hash in block0's 300-471 bytes
- Create the encrypted padded length for the entire message and store it in 644-815 bytes in block 0
    1. Get the entire length of the message
    2. Add padding to the length to make it 128-byte
    3. Encrypt the 128-byte padded length to get 128-byte encrypted padded length
    4. Use Base64.encode() to encode the 128-byte encrypted padded length to the 172 bytes
    5. Store the 172 bytes in 644-815 bytes in block 0

B. length

1. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
2. Get the password from user
3. check to see if the user given password matches with the password in block 0, proceed only if it's true (call matchPassword())

4. Get the encoded padded length from 644-815 bytes in block 0
5. Use Base64.decode() to decode the 172 bytes to the 128-byte encrypted padded length
6. Decrypt the 128-byte encrypted padded length to get 128-byte padded length
7. Remove the padding to get the length

C. read

- Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
- Get the password from user
- check to see if the user given password matches with the password in block 0, proceed only if it's true (call matchPassword())
    1. from the starting_postion and len, get the blocks that you want to read
    2. starting_blocks = starting_postion/128, ending_block = starting_postion+len/128
    3. for the chosen blocks, c = [starting_blocks … ending_blocks],
        1. get 0-171 bytes of encoded encrypted padded message
        2. use Base64.decode() to decode the 172 bytes to the 128-byte encrypted padded message
        3. Decrypt the 128-byte encrypted padded message to get 128-byte padded message
        4. Remove the padding to get the message
        5. Concatenate the message for all the chosen blocks
    4. Only return the selected message = str[starting_position % 128, starting_position % 128 + len]

D. write

1. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
2. Get the password from user (need to verify the password)
3. check to see if the user given password matches with the password in block 0, proceed only if it's true (call matchPassword())
4. get the content
5. call read() to get the current_message
6. use starting_position and the current_message to create the new_message to be written
    - choose if the content needs to be appended from starting position of the current stored message or,
    - choose if the content will replace some substring of the current stored message or,
    - if the current stored message is empty, then we will just need to write the content
7. once the new message to be written is decided, call the function writeMessageBlock(new_message, starting_position), to write the message to the appropriate blocks

E. check_integrity
- Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
- Check to see if the user given password matches with the password in block 0, proceed only if it's true (call matchPassword())
- call helper function Function1() to check the integrity of the message
    1. For each message, m_i, is stored in the i^{th} block compute the HMAC and compare it with the HMAC stored in the $(i-1)^{th}$ block
    2. And do it for all the blocks
    3. Return true, only if all of them pass
- Detail of Function1()
    1. Use read() to get the entire message, M
    2. Partition the M into 128 bytes m_i = message[(i-1)*128 : 128*i]
        - i = 1 to x, where x = total no. of blocks = (message.length()/128) + 1
    3. Get the 172-byte encoded encrypted $HMAC_K$ block from 472-643 bytes in block i-1
    4. Decoded using Base64.decode() to get 128-byte encrypted padded $HMAC_K$
    5. Decrypt the 128-byte encrypted padded $HMAC_K$ to get 128-byte padded $HMAC_K$ of the stored message in block i
    6. Remove padding to get 256-bit $HMAC_K$ of the stored message in block i
    7. generate the $HMAC_K$ for m_i using the process listed in "MAC for each 128-byte block of message"
    8. If the generated $HMAC_K$ matches with the stored $HMAC_K$ for m_i, we know the message has not been altered
    9. Do this for all m_i, i = 1 to x

F. cut
    1. Call read to get the message, M
    2. Truncate the message M, to the desired length length_str
    3. Call the function writeMessageBlock(message, starting_position), to write the message to the appropriate blocks

G. matchPassword()
    1. Get the hash of the salted password stored in block 0
        - Get the encoded padded hash of the salted password from bytes in 300-471 in block 0
        - Use Base64.decode() to decode the 172 bytes to 128 bytes of padded hash of the stored salted password
        - Remove the padding to get the 256-bit hash of stored salted password
    2. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV

3. Get the current password user entered, and rerun the steps of "To store the password" but stop in step 5, to get the 256-bit hash of the current salted password
4. Match the hash of the current salted password with the hash of salted password in block0
    - If they are the same, correctly authenticate
    - If not, the password mismatched

H. Function writeMessageBlock(message, starting_position)
    1. get the length of the message
    2. convert length -> padded 128-byte length
    3. encrypted the padded length
    4. update the block0 with the encrypted padded length
    5. using starting_position, calculate the block that needs to be updated
    6. for each block that needs to be updated
        1. created the needed blocks if the blocks are not already created
        2. get the HMAC[m_i] for the message m_i (HMACcreatorFunc(m_i)) that is going to be written to the block i, and write it to 816-987 bytes in block i-1
        3. Get the encoded IV from 128-299 bytes from block 0 and decode it to get 128 bytes IV
        4. Get the password from user
        5. encrypt the part of the message, m_i, using the method described in "Encryption Design" and stored in block i's 0-171 bytes
            a. Each 128-byte message mi is encrypted using AES_128
                i. Each mi is partitioned into 8 128-bit messages, mi_128
                ii. Each mi_128 is encrypted using AES_128 in CTR mode
                iii. The last byte of IV is incremented every time, as we are using AES-128 in CTR mode
            b. 128-byte encrypted m_i, e_m_i, is encoded using Base64.encode() to get 172-byte encoded e_m_i
            c. Finally, store the 172-byte e_m_i into the block i's 0-171 bytes

I. Function HMACcreatorFunc(m_i)
    - Create the $HMAC_K[m_i]$
    1. HMAC[m_i] is 128 byte padded output of Hash((K⊕opad) || Hash((K ⊕ ipad) || m_i))
    2. ipad = the byte 0x36 repeated (256-bit/8) 32 times
    3. opad = the byte 0x5C repeated (256-bit/8) 32 times.
    4. K=padded key = (password || hash)
    5. 256-bit hash output is padded to get 128-byte padded hash
    6. 128-byte padded hash is partitioned into 8 128-bit blocks
    7. Each 128-bit block is encrypted used AES-128

8. 128-byte encrypted HMACK[mi] is encoded using Base64.encode() and returned

3.3 Design variations

1. Suppose that the only write operation that could occur is to append at the end of the file. How would you change your design to achieve the best efficiency (storage and speed) without affecting security?

Answer: If write operation can only append, then I will just need to change the writeMessageBlock() function. The change will that I will just create a new extra block for each write request and update the length in block 0. Since, we are appending, I know that any of the earlier blocks have not changed, so I do not need to even access them. I will just need to make and add the extra block(s) as well as update it with the encrypted message. I can use a global variable that will store the current block number and use it to create new blocks without calling the length() function. Since, I am not calling length() or read(), I am not decrying unnecessary blocks so I have having the best speed efficiency. I cannot change the storage efficiency without compromising the security, based on my design.

2. Suppose that we are concerned only with adversaries that can steal the disks. That is, the adversary can read only one version of the same file. How would you change your design to achieve the best efficiency?

Answer: I will get rid of the 128-byte message partition part as well as HMAC per block. Since, the attacker can only see one version, I will generate a HMAC for the entire message and put it in block 0. Then I can basically encrypt the entire message in one long stream and store it in 1024 bytes block. And, to verify, I can just decrypt the messages and check the HMAC with block 0. This design will have the max storage and speed efficiency as we are not introducing buffers nor breaking the messages into blocks. Since, we are not breaking into blocks, the number of blocks will eb less also, so to write or update we need to access a smaller number of blocks compared to my current design.

3. Can you use the CBC mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If no, why?

Answer: Yes, I will have to add the xor with previous cipher block to my algorithm. Currently, I have $C_i = E_k (M_i)$, I will just have to update it with $C_i = E_k (M_i \oplus C_{i-1})$, with C_0=IV. The storage space needed in CBC and my current implementation will be same, but the speed efficiency will change. The time to xor with the previous cipher block will be added for encryption and decryption so CBC mode will take more time.

4. Can you use the ECB mode?

Answer: No, we cannot unless we want to create an insecure EFS. Because using ECB would mean that the same message with the same password would lead to the same encryption block. I am assuming in ECB; we are not using the CBC in CTR mode or IV. It would be easier for the

attacker to understand if something is changed or not, or if a message is repeated, it would be easily detectable.

3.4 Paper Reading

Problem 1:

The paper "*Why Cryptosystems Fail*", written Anderson showed how in the last 1980s, the attack model misrepresented the security issue as they primarily focus on "what might happen?", rather than what was actually happening. There were multiple issues that affects the security and cryptosystem development. But one primary issue was the lack of feedback procedure and the code of secrecy by government and large organizations. Since, there was not much feedback procedure in place, not many people were aware of the flaws that were being discovered somewhere else. The malicious actors who were aware of the flaw got to misused it rampantly in some other premise or organization. Since, not many government organizations were sharing cryptosystem flaws with other organizations, who were still vulnerable to the exploit or backdoors, this created a sense of mistrust between organizations and government, that delayed the development of security patches as well as propagation of the patches to all the vulnerable devices.

The next big problem was that not the design of the cryptosystem but rather the implementation of the cryptosystem. The cryptosystem was implemented in a flawed manner as well as the cryptographic primitives were mismanaged by the implementers. The mistakes that were being made during the implementation propagated exploits as well as backdoors that were being exploited by malicious actors. Since the government were not sharing information the same mistakes that were made before were being made again and this cycle continued. Therefore, one of the solutions the author mentioned is that there needs to be adequate training and adequate resources available for security researchers and security workers. The security workers need to understand the security landscape as well as all the available vulnerabilities that are out there.

Anderson mentioned the last thing that was concerning to the community that more often than not the people who implemented the cryptosystem misused it. Since the people who implemented the cryptosystem knows the in and out, they can easily insert a back door or keep the private key with them; which that they can use later on. Organizations need to make sure of two things, first that the implementation is being done in a right way and that there are oversight on the security researcher who is implementing it. The organization needs to make sure when a private key is such as PIN is created for a designated user, it is only shared with the customer and no one else. The organization need to put checks in place, so that even if the key falls in the hands of an employee or someone who can misuse it, they should not be able to do it. So the main point the author was trying to make is that we need to protect the cryptosystem from its implementers. The author mentioned the main problem is we need to understand how to protect the private keys and transfer the private keys to only the designated people even if the private key is created by someone else. Finally the author mentioned that we need a new security paradigm, we need to instead of focusing on what might go wrong we need to systematically understand what is likely to go wrong.

Problem 2:

The paper "*Intercepting Mobile Communications: The Insecurity of 802.11*", written by Borisov et. Al. showed an interesting occurrence of how incorrect usage of cryptographic primitive can render a supposedly secure protocol defenseless against a resourceful and smart attacker. The 802.11 standard introduced WEP protocol so that wireless transmission becomes as secure as wired transmission. The authors mentioned regarding the attack practicality that even though it is necessary for the attacker to have specialized equipments as well as powerful processors to execute these kinds of attacks, the discussion regarding these attacks are necessary. The discussion is necessary because once the message becomes too important such as corporate espionage significantly powerful attackers come into the picture and would try to break the protocol.

The main of issue in WEP protocol is because of the reuse of the initialization vector and the keystream. The WEP protocol mentions that initialization vector as well as the keystream should not be reused. Since, IV are public, the attacker can easily figure out is an duplicate IV is used. Since, re-use of IV almost always cause reuse of RC4 keystream, the attacker can easily regenerate the victim's keystream. Thus, once the key stream is generated the attacker just xor it with the CT to get the PT.

But the authors show that due to inefficient and incorrect implementation of the WEP protocol as well as incorrect usage of the cryptographic primitive, stream cipher, attackers can easily eavesdrop into confidential conversation, find private keys, modify message and inject them without any consequence. But, before explaining the attacks, lets mention the two conditions necessary for the attacks so succeed: 1> availability of ciphertexts where some portion of the keystream is used more than once and 2> Partial knowledge of some of the plaintexts. The author also mentioned that it is not hard to brute force the 24-bit IV for WEP and even with a random IV. The attacker can create dictionaries of every known word and use the dictionary to just query to find if a known plaintext has been sent. The attacker also gets the power that using this dictionary to decrypt subsequent ciphertext with very little work as well as be able to predict if the underlying key has changed in subsequent messages.

The authors focus on message injection as well as integrity protection of WEP protocol. The attacker can use the checksum to do controlled modification of a ciphertext as well as inject malicious frame into the network, because WEP is not a keyed function of the plaintext message. The attacker once it gets hold of old keystream it can generate the old initialization vector. And once that attacker gets the IV it can use it indefinitely and as many times as it wants to generator new WEP message and injected those messages into the stream without raising any kind of alarm. This is the reason, why we need to use keyed message authentication code (MAC) such as SHA1-HMAC, such that the result output is a keyed function of the message.

Another way that occur can cause malice 'softer it has intercepted an authentication sequence using a particular key the attacker can authenticate itself with that key indefinitely. The access point will have no way to distinguish between a valid network point and this attack agent. The attackers can also use the access point as an Oracle. That occurs can see how the Oracle or the access point reacts to a single bit flip. That occur can see the reaction an estimate Argos what the next educated bitflip should be to extract more information. So, the attacker can easily exploit the access point's willingness to decrypt arbitrary ciphertext and leak bits of information one at a time. Eventually, the attacker will be able to regenerate the entire plain text message without even raising an alarm.

The authors also mentioned that this could be easily avoided is the cryptographic research community was invited during the protocol creation. The protocol was created to be easy, fast, stateless and liberal which were a disaster from a security perspective. The reuse of keystream became an issue due to the use of stream cipher, and stream cipher was used because of speed and implementation easability. The industry standard of not inviting researchers to collaborate has caused this issue and this should change in the future, so the researchers have more understanding as well as infrastructure to work with industry application designers to create a better application.