CRAM-MD5, HMAC, Symmetric Key encryption

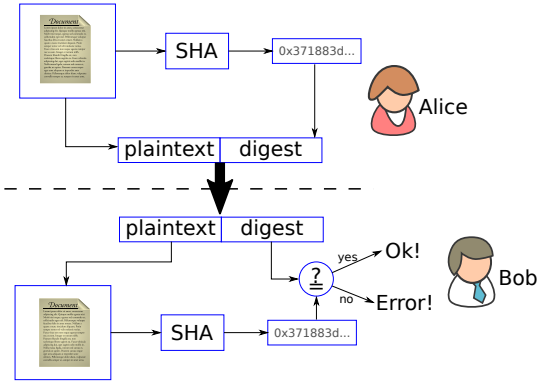
COMPUTER NETWORKS A.A. 24/25

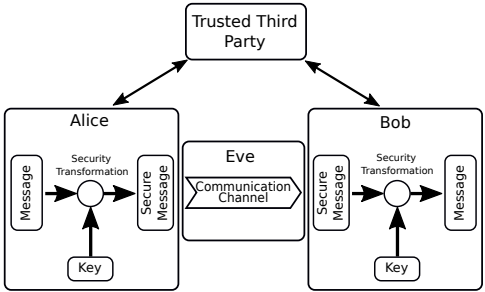
Leonardo Maccari, DAIS: Ca' Foscari University of Venice,
leonardo.maccari@unive.it

Venice, fall 2024

Sect. 1 HMAC and CRAM-MD5

A (dummy) hash-based security protocol





However, Eve controls the channel and she wants to modify M without Bob noticing. What could she do?

Hash limitations

- Eve intercepts the communication that contains the plaintext message and the digest (M, D)
- She modifies M in M', and recomputes $H(M') = D'$
- She forges a new message (M', D')
- Bob receives this message, verifies that $H(M') = D'$, and concludes that everything is OK
- Eve was able to break the integrity of the communication.

Hashes Take Away

Need for two separate channels

- Using a hash to ensure integrity makes sense if the digest is sent on a channel different than channel used for the message.
- Hashes are effective is the attacker can not intercept both channels.
- The example of Ubuntu using the website (to deliver the hash) and Torrent (to deliver the image) is explanatory: if the website is secure, the integrity is preserved.
- But if the attacker can modify both the message and the digest (the Ubuntu image and the website), it makes little sense.

Counter Example

Using hashes as attachments to secure email messages is pointless, if Eve can intercept the email, she can also intercept and change the attachment.

HMAC: Hashed Message Authentication Code

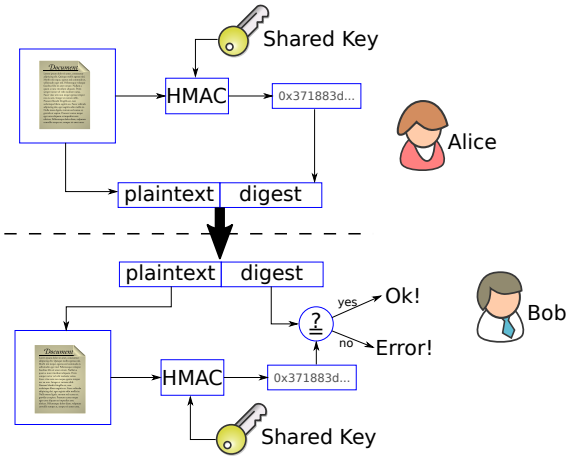
- An HMAC is used to solve this problem
- In an HMAC, Alice and Bob first agree on a secret shared key K (like a password), and on a secure hash function $H()$.
- They can then use the password as a further input to the hash function[1]:

$$D = HMAC(M, K) = H(\,(K \oplus \texttt{opad}) \,|\, H(\,(K \oplus \texttt{ipad}\,) \,|\, M\,)\,)$$

- `ipad` and `opad` are just padding sequences needed to align K with the length of the output of $H()$. If K is longer than the digest, replace K with $H(K)$.
- For instance, if the hash function is `SHA-256()` then `ipad` is `0x36` repeated 32 times. If K is longer than 32, replace K with `SHA-256(K)`.

[1] This is the construction of an HMAC as in RFC 2104
`https://datatracker.ietf.org/doc/html/rfc2104`

HMAC


Shared Key
Document
HMAC
0x371883d...
Alice
plaintext
digest
plaintext
digest
yes
Ok!
?
no
Error!
Bob
Document
HMAC
0x371883d...
Shared Key

HMAC

- If Eve intercepts both M and D, in order to break the integrity of the message she should:
  - modify M into M',
  - recalculate $D' = HMAC(M', K)$
  - but Eve does not know K, so she cannot calculate the HMAC.

HMAC Vs Hash

- Pro: with HMAC both the message and the digest can be sent in the same communication channel. For example, HMAC can be used to provide integrity for emails.
- Pro: HMAC provide authentication. Bob is sure the message comes from Alice because she is the only one that owns the shared secret and could generate a correct HMAC.
- Cons: Alice and Bob need a different, secure communication channel to agree on the shared key. In practice, if Alice and Bob use email to agree on the shared secret, Eve can intercept it and break the integrity in the next messages.

NOTE: HMAC provides integrity and authentication, it does not provide secrecy. Eve can still read the message. To achieve secrecy, you need encryption.

On Secure Channels

- Often we will see that a certain protocol needs a secure channel to be used
- This means that Alice and Bob will use two channels to communicate:
  - one that is somehow secure (guarantees secrecy or authentication, or both). This channel is generally used only once to exchange a key.
  - one that is not secure. This channel is used all the time to transfer information, this second channel is made secure using cryptography.

Example: Alice and bob meet in person to agree on a key (the secure channel), then they use the key to secure the content of emails (the insecure channel).

Generating keys

- A key should not be guessable by an attacker, it should be be a random binary string.
- People are not good at remembering random strings.
- One way to generate a key is to use an hash function:
  - Alice and Bob agree on a strong password:
    $P =$"nelmezzodelcammindinostavita-1321"
  - They communicate the password once, over a secure channel
  - K is generated with an hash function: `K = H(P)`
- Due to the pseudo-random behaviour of a hash function, knowledge of the hash doesn't give information about P

A Strong Enough Password

Of course Eve can brute force the hash, so the features of a password are:

- It must be long. Passwords with less than 10 chars, can be generated by a computer and even found on-line. Target your password to at least 15-20 chars
- It must be made of letters, numbers, symbols. Increasing the alphabet makes it longer to generate all the passwords.
- It must not be present in a dictionary of existing words.

HMAC and CRAM-MD5

↳1.1 CRAM-MD5

A Simple Challenge/Response Authentication

- So we now have the cryptographic functions to implement an HMAC.
- we can use it in a protocol that achieves some security service.
- CRAM-MD5 is a simple protocol that achieves user authentication, based on the knowledge of a password and the agreement of a hash function.
- It was defined in RFC 2195[2]

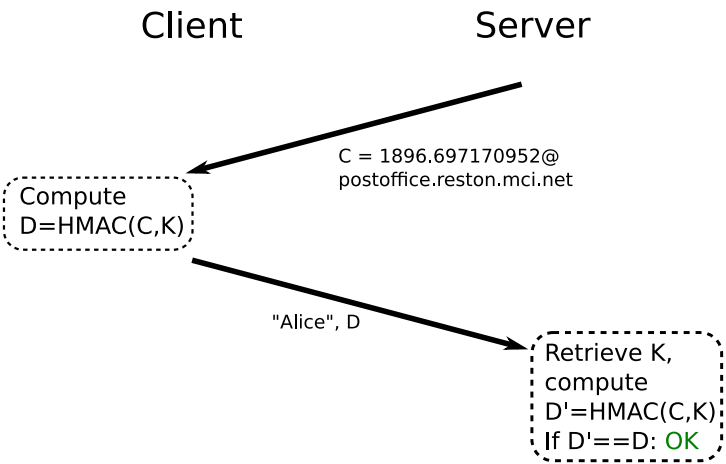
[2] `https://datatracker.ietf.org/doc/html/rfc2195`

CRAM-MD5

- Assume Alice (the client application) wants to authenticate to Bob (the server application). The usename is "Alice" and the shared key is K
- Bob composes a challenge. This could be a random string, but customary it is of the form: `<timestamp@domain>`. For instance:

$$C = < 1896.697170952@\texttt{postoffice.reston.mci.net} >$$

- Alice knows K and can compute $D = HMAC(C, K)$, then send back to Bob the message $(\text{"Alice"}, D)$
- Bob check in the users database if "Alice" is present, retrieves K, he can recompute $D' = HMAC(C, K)$ and then check if $D' == D$



It's important that the challenge is always different, so that Eve can not make a reply attack, reusing D she intercepted in a previous exchange.

Security of CRAM-MD5

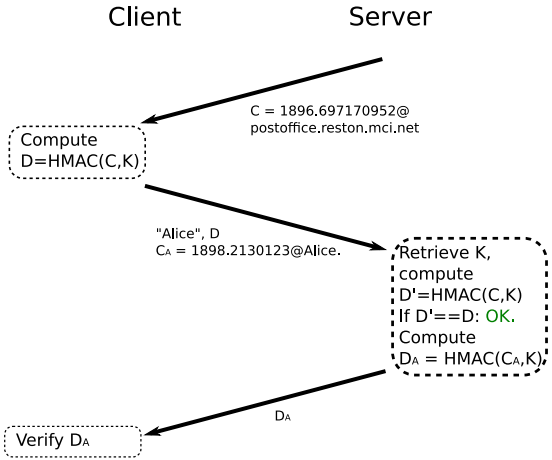
- The protocol was an improvement over plaintext transmission of the secret key
- However it is now deprecated[3]. We divide the reasons in three categories:
- 1) Insecure Crypto and lack of extendibility
  - MD5 is not considered secure anymore
  - In general, it is not wise to rely on one single crypto function.
  - But if the protocol relies on more, there is a need to exchange some initial messages to agree on which one to use
  - This introduces complexity, slows down the protocol and can be tricked by downgrade attacks in which Eve changes the messages and forces Alice and Bob to agree on the weakest function.

[3] `https://datatracker.ietf.org/doc/html/draft-zeilenga-luis140219-crammd5-to-historic-00`

Security of CRAM-MD5

- The protocol was an improvement over plaintext transmission of the secret key
- However it is now deprecated[4]. We divide the reasons in three categories:
- 2) Authentication is non-mutual
  - Alice does not know if the server is really Bob
  - We could repeat the protocol twice, reversing the role of the Server and the Client
  - So in the end Alice knows that Bob also knows the password
  - This is similar to what WPA2 does in your wi-fi network, but CRAM-MD5 does not support it

[4] https://datatracker.ietf.org/doc/html/draft-zeilenga-luis140219-crammd5-to-historic-00


Client
Server
C = 1896.697170952@
postoffice.reston.mci.net
Compute
D=HMAC(C,K)
"Alice", D
CA = 1898.2130123@Alice.
Retrieve K,
compute
D'=HMAC(C,K)
If D'==D: OK.
Compute
DA = HMAC(CA,K)
DA
Verify DA

Security of CRAM-MD5

- The protocol was an improvement over plaintext transmission of the secret key
- However it is now deprecated[5]. We divide the reasons in three categories:
- 3) Off-line brute force attack
  - Eve can observe C and D in plaintext, which means that she can produce an off-line brute-force attack. Given a list of passwords L

```
1    for p in L:
2    if HMAC(C,p) == C:
3    break  # success!
```

- This is much worse than an on-line brute force attack, that can be stopped as soon as detected.
- To solve this problem we need more crypto

[5] https://datatracker.ietf.org/doc/html/draft-zeilenga-luis140219-crammd5-to-historic-00

Sect. 2 Symmetric Key Encryption

Encryption Fundamentals

- Elements of a cryptographic system:
  - cipher (algorithm)
  - key (information)
- Kerchoffs principle:
  - the cipher is supposed to be known to all, ciphers are public
  - the secret resides in the key
- Knowledge of the key
  - allows to encrypt/decrypt documents
  - in some cases it can be a proof of identity
- The encryption algorithms implement secrecy and sometimes authentication

Snake Oil

- There is nothing like a secure secret cipher.
- By definition, a cipher is as secure as more people tried to break into it and failed.
- A secret cipher is **Snake Oil**.

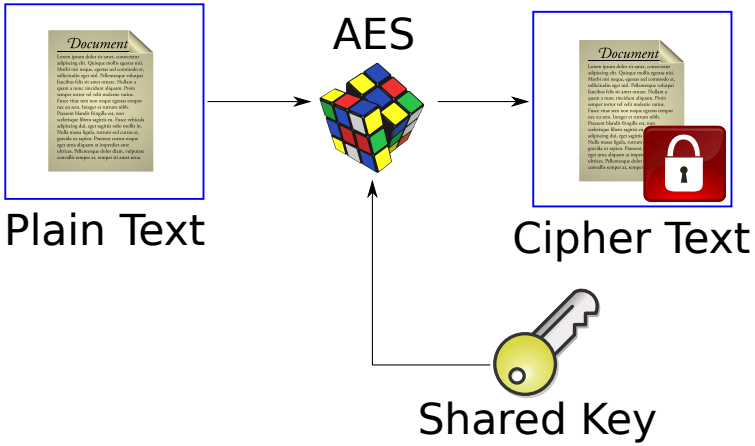
Wikipedia says:

> *In cryptography, snake oil is any cryptographic method or product considered to be bogus or fraudulent. The name derives from snake oil, one type of patent medicine widely available in 19th century United States.*

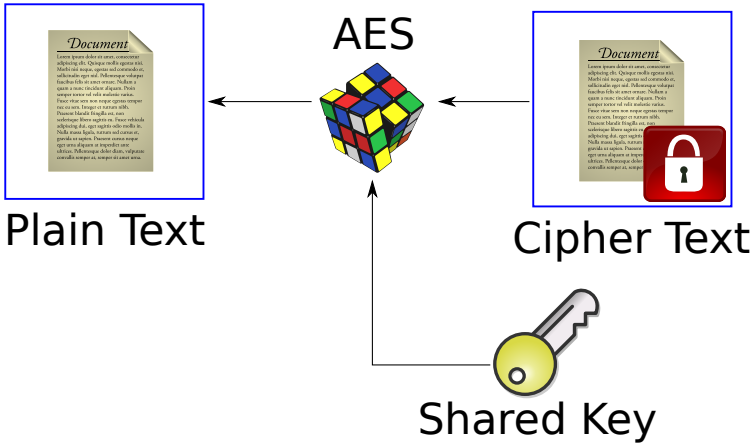
Symmetric Key Encryption

- as for a HMAC Alice and Bob use a secure channel to agree on a shared key K, and a cipher to use. For instance, today the AES encryption function is considered secure.
- when Alice wants to send a message M to Bob she uses the AES algorithm to create an encrypted message C.
- The input to AES are the key K and the message M: `C = AES(K,M)`
- When Bob receives the encrypted message C, it uses the same function to recover M: `M = AES(K,C)`
- generally the algorithm is the same both to encrypt and to decrypt

Encryption



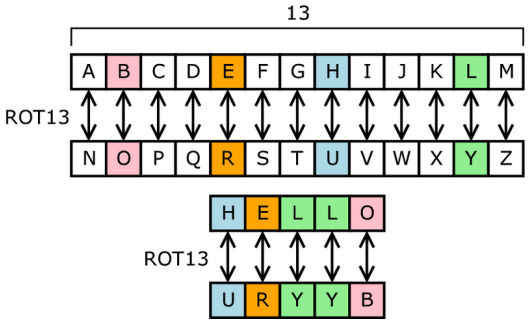
Dencryption



Base Encryption Operations

- As hash functions are based on XOR and shift, encryption is also based on two operations, substitution and transposition.
- The substitution algorithms replaces one symbol with another
- The transposition algorithms (or permutation) exchange the order of the symbols
- To understand what they are, let's look to ancient algorithms.

Classical Algorithms: Caesar algorithm (ROT-13)



Caesar Algorithm

- Pick a number K from 1 to 25. K is the key. To encrypt a text do:
  1. pick each letter X one after the other.
  2. Transform X into a number equal to its position in the alphabet: $X \to N_X$
  3. Add K to it: $N_X + K \mod 25$
  4. replace X with the letter that in the alphabet has position $N_X + K$
- To decrypt the letter, do the same operation subtracting K instead of adding it.
- Note that if $K = 13$ then encryption and decryption are the same function (summing or subtracting 13 produces the same letter).

Simple Substitution

- The Caesar algorithm is a simple substitution algorithm, it replaces a symbol with another symbol in a consistent way (e.g. A is always mapped to N).
- Plus, it only allows 25 keys (you only have 25 shifts), so it allows simple brute force attacks.
- You can make this scheme more complicated, for instance, instead of using a shift in the alphabet, you could have a static map from one letter to another.
- The number of keys then passes from 25 to the number of permutations of 26 letters, and make the brute force attack harder.

Simple Substitution

Regardless of how you design the mapping between cleartext and ciphertext symbol, this kind of encryption can be easily defeated with simple statistical analysis[6]. A sketch of an attack is (let's assume the text is in the Italian language):

- Eve picks a corpus of italian texts, she count the number of times a certain letter is used.
- Once she has a frequency diagram, she does the same with the encrypted text.
- Then she tries to match the most frequent letters in Italian, with the most frequent symbols in the encrypted text and check if the outcome is something reasonable. If not, tries with slightly different matches...

[6] Wkipedia has a nice page on this
`https://en.wikipedia.org/wiki/Frequency_analysis`

Transposition Algorithm

Another classical crypto algorithm is a transposition one

SEGRETO DI GRANDISSIMA IMPORTANZA

5	3	1	6	2	4
S	E	G	R	E	T
O	D	I	G	R	A
N	D	I	S	S	I
M	A	I	M	P	O
R	T	A	N	Z	A

1	2	3	4	5	6
G	E	E	T	S	R
I	R	D	A	O	G
I	S	D	I	N	S
I	P	A	O	M	M
A	Z	T	A	R	N

GEETSR IRDAOG ISDINS IPAOMM AZTARN

Simple Transposition

- The order in which columns are swapped is the key K.
- Transposition still preserves the frequency of letters, so it is easy to guess the original language
- yet, a letter X is not always mapped to the same letter when encrypted.
- Then one may try to anagram pieces of text to find sequences that look like known words and recover the transposition pattern.

Algorithms Take Away

- A cipher transforms a plain text message M in a ciphertext C using a key K.
- A cipher works if there is no statistical correlation between C and M.
- If there is some correlation (for instance, the frequency of the letters is preserved), then Eve may be able to recover M from C without knowing K.
- Simple transposition and substitution leave enough correlation to be exploited by Eve to break the cipher.

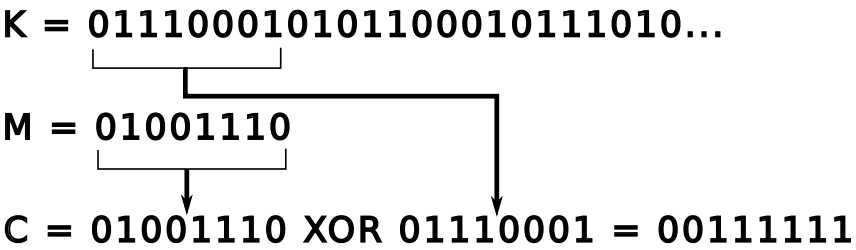
A Perfect Cipher

- Is there a perfect, theoretically unbreakable cipher?
- Yes, there is one, and it is called One-Time-Pad (OTP).
- One-time-pad works like this:
  1. Alice generates a random string of n bit. This string is transmitted to Bob over a secure channel (a channel that Eve can not intercept)
  2. To encrypt a message M of $m < n$ bit Alice takes a portion K' of K of length m and calculates $C = M \oplus K'$
  3. The portion K' of the original K key must not be re-used.
  4. It can be easily shown that there is no correlation between C and M: statistical analysis can not be performed. OTP is theoretically impossible to break.

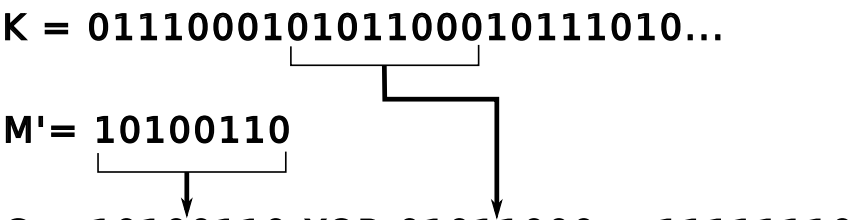
One-Time-Pad

- So imagine $K = 1100100111010110$ and $M = 00101101$
- $C = 11001001 \oplus 00101101 = 11100101$
- M can not be recovered from C without knowing K
- The next message M' will need to be encrypted with the following part of K

OTP: first message M



OTP: second message M'



Why is OTP "perfect"?

- Imagine a toy example in which the cipher text Eve has is 101.
- This means that the key must be three bits long, and Eve can easily brute-force it
- She will compute a table as this one, every candidate k corresponds to a candidate plaintext

k	c	$m' = k \oplus c$
000	101	101
001	101	100
010	101	111
011	101	110
100	101	001
101	101	000
110	101	011
111	101	010

Why is OTP “perfect”?

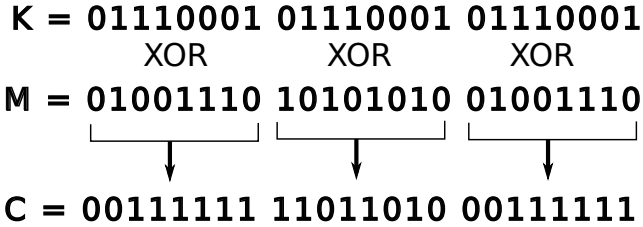
- However, every possible candidate appears only once in the table
- That is, every candidate key maps to one and only one candidate plaintext
- This means that there is no candidate plaintext that has a higher probability than another.
- Brute forcing does not give any information.

One-Time-Pad Limitations

- The problem with One-Time-Pad is that Alice to send a message of lenght N needs a secure channel to send a key of lenght N.
- If she onws that secure channel, why don't she just use it to send M?
- One solution would be, re-use the same key?

Reducing the Key length

- Imagine Eve has the encrypted message C, that is several bytes long, and that she knows this message is a text message in Italian.
- Let us assume that the key is 8 bit long, and that it is used for all chars of the message.



- Note that the first and third bytes of the message are the same ones. And so they are in the ciphertext.

Reducing the Key length (2)

- This is the exact situation of the Caesar substitution algorithm: every char is mapped to another char, preserving the frequency of the symbols.
- Eve can now exploit this property as with the Caesar algorithm, and attack the encryption

If you reduce the size of the key of an OTP you reintroduce the correlation between M and C.

One-Time-Pad Limitations

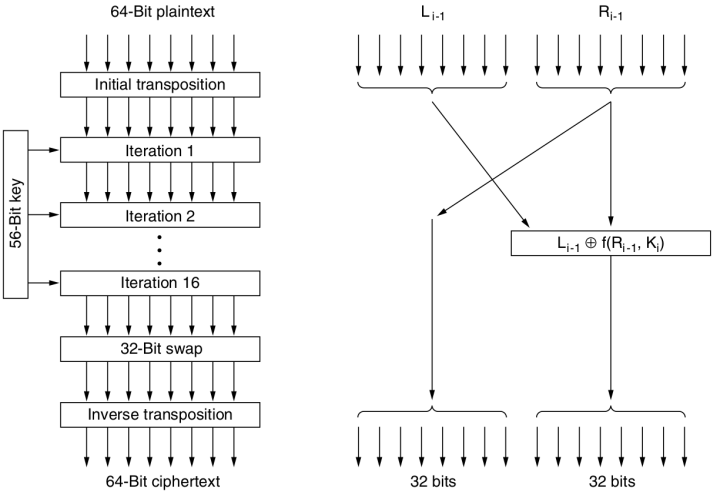
- The use case of OTP is limited. Crypto folklore tells that spies used it: the spy was sent abroad with a long-enough key for future communications and could use it for the lenght of the whole mission.
- But in most use cases it is not suitable. Modern algorithms need to use a different approach.

Modern Algorithms

- Modern algorithms try to mix the two primitives of transposition and substitution to achieve an algorithm with the following features:
  - The key lenght is fixed, it does not need to be as long as the plaintext.
  - The statistical correlation between M and C is so little that it is computationally impossible to recover M from C.
- To achieve this, modern ciphers apply a high number of rounds of substitutions and transposition to M.
- Every round will remove some correlation, and hopefully at the end so little correlation remains that is not noticeable.

Feistel Scheme

Modern block cipher use a Feistel scheme: divide input into blocks, on each block apply a sequence of transpositions and substitutions. This is the scheme of DES.



Some more Remarks on Obscurity

- Two things emerged:
  - The only theoretically secure cipher is not practical. Popular ciphers try to mimic a perfect one, but they are not, i.e. they introduce some correlation between M and C. **We assume it is too small to be exploited by attackers**.
  - Keys are generated by passwords via hashing and hash functions allow collisions. An attacker could find another password that has the same hash. **We assume this event is very rare**.
- The mathematical design is the only thing that can justify the common assumption that, albeit theoretically possible, it is "computationally impossible" to break a cryptographic function.
- Can mathematicians be wrong?

Yes, They Can.

- Example: The RC4 algorithm is an encryption algorithm that was used in the first generation of Wi-Fi networks.
- Based on a shared key K and a network packet M it generated an encrypted packet C to be sent by the Access Point to the client (or the other way around).
- RC4 was revealed to be broken. An adversary that was able to collect a small amount of packets was able to make a mathematical correlation between the packets and the key.
- With enough packets, any attacker could obtain the shared secret K.
- This was possible by simply listening to a wireless network traffic.

Wardriving[7]

Early 2000s were very interesting years to be a Wi-Fi hacker...

[7] Image from `https://pygmalion.nitri.org/wardriving-using-an-ubuntu-10-04-notebook-and-a-garmin-etrex-40.html`

How was this Possible?

- It was possible because at the moment of its adoption, RC4 was not public. It was widely used but its specifics were not public.
- When more people were able to look into it, they found out some extremely serious weaknesses.

Obscurity does not work

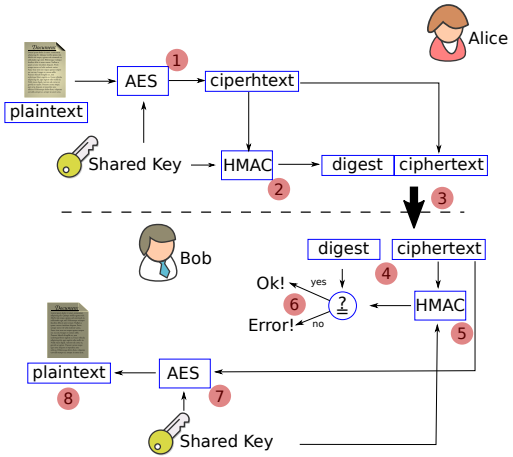
This is the fundamental reason why a cryptography function must be public and well known. Because the math is hard, and the more people contribute to its analysis the lower are the chances it may be broken.

Symmetric Key Encryption

↳2.1 Encryption and Authentication

Achieving Integrity, Secrecy and Authentication

Once Alice and Bob have a shared key K, they can use the same key for both HMAC and Encryption.



Example Protocol: Step-by-step analysis

On the sender side:

1. Alice uses a symmetric encryption scheme to transform a plaintext into a ciphertext, using a shared key. In this example we use AES. This operation needs a symmetric key, that must be shared with Bob.
2. The ciphertext is then passed through an HMAC function, that uses again the shared key. The result is a digest of the ciphertext
3. The digest and the ciphertext are concatenated and sent over the communication channel

Example Protocol: Step-by-step analysis

On the receiver side:

4. Bob receives the message and splits the digest from the ciphertext
5. Bob uses the ciphertext together with the shared key to produce a candidate digest.
6. If the candidate digest corresponds to the digest that has been received, then the message was not tampered during the transmission. Otherwise the message is not valid and the process stops. This step guarantees integrity and also data origin authentication, since only Alice knows the secret key.
7. Bob uses AES with the shared key to decrypt the ciphertext
8. Bob now owns the plaintext, and he is sure the transmission was also secret, as nobody besides him and Alice know the shared key.

Example Protocol: Step-by-step analysis

This scheme provides a secure communication as long as:

- The encryption and HMAC functions are robust. If Alice and Bob use standard functions like AES and known hash functions, then this is guaranteed to the best of current knowledge.
- They somehow shared a secret key before the beginning of the communication. This key must be long enough. Nowadays 256 bit keys are considered secure. Note that in real life, they will use two different keys, one for encryption and one for MAC, but in this example we use only one for simplicity.

Off-line brute force

- Note that the off-line brute force problem problem still remains
- Using two separate keys for encryption and HMAC would make it harder
- However, if the two keys are generated from the same password (for instance with 2 different hash functions), it doesn't change anything

Example Protocol

An Encrypt-then-MAC scheme uses symmetric key encryption and Message Authentication Codes to first encrypt and then compute the MAC on the message. If the shared key(s) used are known only to Alice and Bob, then this scheme provides integrity, data origin authentication, and secrecy.

- We use variants of these protocols daily, for instance, Wi-Fi uses them
- What remains unsolved is how Alice and Bob will agree on a shared key. This problem is addressed by the use of public key encryption.

Sect. 3 Password Files

Hash Functions and Password Databases

- To perform access control, you need to have a user and password database to store credentials.
- There are two important constraints for a password database:
  - A password must be stored in a format that allows you to compare it with user input.
  - Passwords can not just be saved in clear. A password must be stored in some *modified* way that, if an attacker gains access to it, he/she must not be able to recover the password in clear.
- A hash function is a good candidate for storing passwords:
  - Hashes are good to test if two input values are the same
  - Hashes can not be inverted

Naive Password Datbase

- When a Alices logs-on in a system, a new user is created, the system asks for a username and a password, Alice enters a password P.
- The system computes D $=$ `MD5`(P)
- The password database is a list of entries, each entry has two fields: the username and D
- When Alice logs-in she enters the username and the password P
- The system computes D' $=$ `MD5`(P)
- The system looks up the username in the password database and extracts D
- If $D = D'$ then Alice is allowed to enter.

Password leakage

- If Eve is able to break into the system and steal the password database, she only obtains D.
- To log-in she needs P and since hash functions are one-way, she can not recover P
- This way, the system has a log-in function that works, and does not store passwords.

Pre-hashed Passwords

- Now consider the following scenario:
  - Eve breaks into system X and steals the password database, password are hashed.
  - Eve takes a huge dictionary of m words, and for each word p, Eve computes D $=$ `MD5`(p) and saves it. This is a brute force attack.
  - At the end she has a big table of m lines, each line of the kind $p : D$.
  - Eve has limited computing power, so hashing m words may take months.
  - In the end, only n users of system X used a weak password that was in the dictionary, so she is able to obtain just $n << m$ passwords.

Pre-hashed Passwords (2)

- Later on, Eve breaks into system Y and steals the password database. Password are hashed with md5.
- Eve can re-use the same m hashed words that she generated in the first attack to revert passwords from database of service Y.
- **She does not need to wait months to do the brute force attack again**.

Let me introduce you to Raimbow Tables

- Raimbow Tables are space-efficient data structures to store databases of pre-calcolated hashes.
- You can find them on-line: `http://project-rainbowcrack.com/table.htm`
- Example: the whole space of alphanumeric words made of 1 to 9 chars is roughly 2^{17} words. The Raimbow Table that matches every password to its MD5 hash is a 690 GB file.

Salted Hashes

- To defeat Raimbow Tables password files need to be made in a way that the same password, if saved in two different databases, produces a different hash.
- The solution is to use a 'salt', that is a random number that is prepended to the input.
- Example: instead of storing `D = MD5("password")`, a salt $S = 1234$ is chosen. The password is saved as `D = MD5(S|"password")` where ('|' means concatenate).
- Service X will use a different salt than service Y, and thus, Alice can not re-use the table she generated when trying to break service X.
- The use of salts makes raimbow tables useless.

Slow Hashes

- If raimbow tables are defeated, Eve must repeat the brute force from scratch for every database she wants to break.
- We know hashes are fast. This is a design choice, but it helps Eve to check many passwords per second.
- Hash functions use easy math, they can be implemented in hardware. A 700$ GPU can produce 40-50Mh/s[8]
- How fast is that? Let's do the math:
  - Consider all the letters (26) the numbers (10) and 4 most common signs (.,-=). A total of 40 symbols.
  - If you have password made of x chars, this means 40^x possibilities.

[8] See `https://en.bitcoin.it/wiki/Non-specialized_hardware_comparison`

Password Cracking Time

Password length	Time to crack them all (50Mh/s)
4	< 1 s
5	2 s
6	81 s
7	3200 s (< 1 hr)
9	60 days

Too slow?

Mining power

Whatsminer D1 (720days) Miners Leasing

Power	2.4 kw/h x 1 units
Electricity Rate	0.077 USD/kwh
Hash Power	48000 GH/s x 1 units
Est. Daily Earnings	0.41040000 DCR/units
Reviews	☆☆☆☆☆
Amount	- 1 +
Inventory	**0** units
Actually paid	$12,464.00

Sold out

Let's update our table

Password Cracking Time

Password length	Time to crack them all (158Th/s)
8	< 1
9	< 1
10	< 1
11	< 1
12	<2
13	50
20	22 Billion years

Your adversary may not have those resources, but a long passwords is, anyway, for free. Now you know why I suggested 15-20 chars for a password.

Slow Hashes

- Is there a way to make hashes slow? Yes, repeating the hash hundreds of times:

$$D^1 = MD5(salt|P);\ D^2 = MD5(D^1);\ \ldots\ D^{256} = MD5(D^{255})$$

- If you chain 256 hash execution and store in the database the output of the last execution (D^{256}), the hash is still fast enough to be usable for checking passwords but the attacker will take a time 256 larger for a brute force attack.

In Practice: Linux Password File

- "Crypt" is a password hashing algorithm that uses the following format: `$<id>[$<param>=<value>(,<param>=<value>)*][$<salt>[$<hash>]]` where:
  - id: identifier of the hash function: MD5, SHA-256...
  - $< param >=< value >$ some optional parameters, for example number of iterations
  - salt: Base64-like[9] encoded salt
  - hash: Base64-like encoded password hash and salt

[9] Base64 is a standard way of encoding binary numbers in text files.

In Practice: Linux Password File

- Example (from Wikipedia):
  `$md5,rounds=5000$GUBv0xjJ$$mSwgIswdjlTY0YxV7HBVm0$$`
  which stands for
  - use MD5 to hash passwords
  - 5000 hashing rounds
  - `GUBv0xjJ`: salt
  - `mSwgIswdjlTY0YxV7HBVm0`: hash
- If you check the contents of the `shadow` file you will find something similar.
- Practical example: enter the virtual machine and type `passwd`. We know what this command does and also that before changing the password it asks for the current one. Type something, push enter and note: *it takes some seconds to verify the password*. It's the time required to compute the hashes.