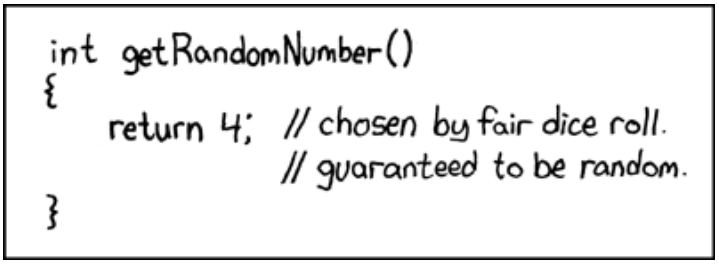
Cryptology (8)

Signatures, Message Authentication Codes (MACs), and Hashes

John York, Blue Ridge Community College
Weyers Cave, VA
http://www.brcc.edu

Obligatory XKCD Cartoon



"RFC 1149.5 specifies 4 as the standard IEEE-vetted random number."
https://imgs.xkcd.com/comics/random_number.png
JY note: 42 is the alternate random number

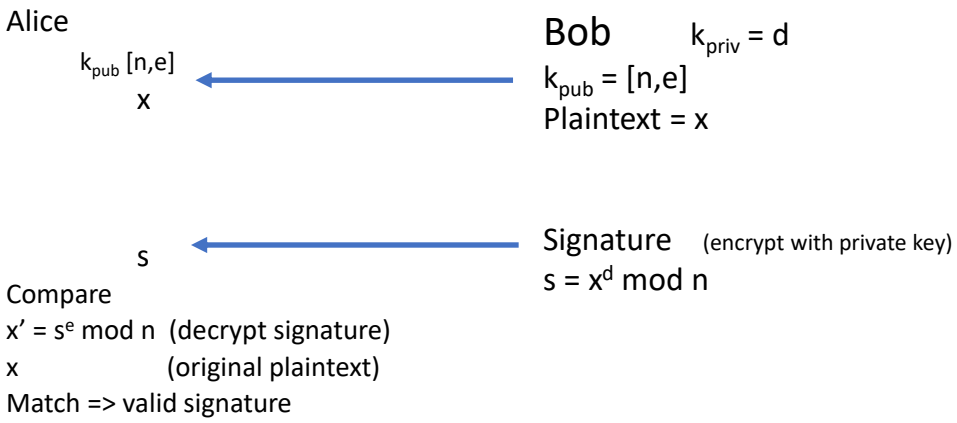
Basic Digital Signature

- Sender encrypts message with their private key
- If message can be decrypted with the correct public key:
  - Message actually came from sender (Message Authentication)
  - Sender cannot deny creation of the message (Nonrepudiation)
  - Nonrepudiation because we are using public/private key pair
  - Message has not been modified (Message Integrity)
- Does not provide Confidentiality
  - Everyone has the public key
  - For Confidentiality, the message must also be encrypted with other keys or symmetric encryption

Digital signatures use public key encryption, meaning asymmetric encryption with a public and private key. That is why it provides Nonrepudiation. We will discuss Message Authentication Codes (MACs) in a few slides. MACs are based on symmetric encryption, so they provide authentication and integrity, but not nonrepudiation (they are faster, too.)

A message encrypted with a private key can be decrypted by anyone with the public key (i.e., everyone.) The goal here is not encryption, though. If you can decrypt a message I encrypted with my private key, you know it came from me (authentication and nonrepudiation.) If it decrypts properly, you also know the message has not been tampered with (integrity.)

RSA Schoolbook—Bob signs message to Alice



Bob gets his public key to Alice. He posts the public key on his web site, uses a Public Key Infrastructure (PKI) system, whispers the public key in Alice's ear, whatever.

Bob gives Alice the plaintext, x. Note that the plaintext may have been encrypted for privacy with a different key pair or some other encryption method like AES.

Bob encrypts the plaintext with his private key, and gives the result (signature) to Alice.

Alice decrypts the signature with Bob's public key. She compares the result with the plaintext that Bob sent. If the two match, the message is valid, or properly signed. The plaintext is proven to have come from Bob, and it has not been changed since Bob sent it. (It could have been changed by noise in the communication channel, or by an attacker like Eve or Oscar.)

Digital Signatures in Practice

- Documents/messages are often long
- Public Key encryption—ciphertext length is ≈ key length (for one block)
- Often the message is hashed, then the hash is signed
- Most commonly used signatures are:
  - RSA (Factoring problem)
  - DSA (Digital Signature Algorithm, either discrete log problem or elliptic curve)
- As always, must be certain we have correct public key to avoid Man in the Middle (MITM) attacks

Remember that public key encryption can only encrypt in blocks that are as long as the key usually 2048 bits or 256 bytes. Using public key encryption to sign a file that is 2 TB in size would be painful. Instead, hash the file to reduce it to a small fingerprint (discussed later) and sign the hash instead of the file.

If Eve, Oscar, Murray, or some other evil person, has managed to give us a fake public key instead of the real one, we are in trouble. They could modify the data from Bob, then sign it with the fake private key that matches the fake public key they tricked us into using. Signatures only work if the public key we have is the one that Bob created, and not one from an attacker.

It's always more complicated

- Schoolbook RSA may be attacked
- Before signing, message is padded and hashed (twice), with a salt
- Encoding Method for Signature with Appendix (EMSA) Probabilistic Signature Scheme (PSS)
- Specification is part of PKCS#1

https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf

Remember how RSA encryption could be attacked? The same problems exist with RSA signatures. For encryption we solved that with PKCS#1 and OAEP padding. There is a similar thing for RSA signatures, except it is called PKCS#1 with EMSA. It adds a random salt so repeated signatures of the same thing give different signatures, and it securely pads the signature so it is as long as the key.

RSA signatures without PKCS#1 EMSA are insecure--good for CTFs but not much else.

RSA signature with PKCS#1 (optional)

- Padding1 and 2 are known to everyone
- Salt is random
- MGF is mask generation function, known to everyone, usually a form of SHA
- To verify, compute MGF of H, then Xor with maskedDB to get salt; then compute from top

https://rsapss.hboeck.de/rsapss.pdf

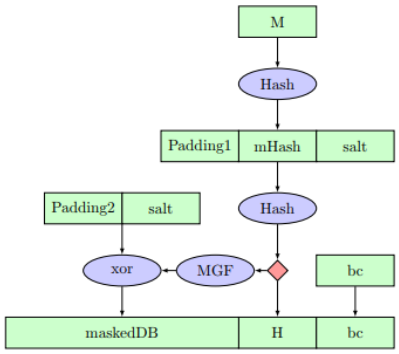


Figure 4: RSASSA-PSS according to PKCS #1 v2.1 / RFC 3447 errata[8]

Here is another block diagram with lots of boxes. Remember, there is a lot that you know ahead of time. The only variables are the message M and the salt

- The hash function and MGF are known
- Padding1, Padding2, and bc are known
- xor is its own inverse. `a xor b = c` means that `a xor c = b` and `b xor c = a`

The procedure to verify the signature is fairly simple.

- Compute the MGF (a hash) of H, and xor that with the maskedDB.
  - The result gives you Padding2||salt
  - Remove the known Padding2 and you have the salt.
- Once you have the salt, you can start at the top with M and compute the hash on the plaintext that Alice or Bob gave you. If you get the same hash that they did, you know the plaintext and signature are valid.

The salt adds the random component we need, and all the hashing ensures that a one bit change in the plaintext M causes a huge change in the signature.

DSA Signatures—Diffie-Hellman with a twist

- Use Discrete Log Problem (DLP) or Elliptic Curve (ECC), just as DH or ECDH do.
- Use a prime number ~2048 bits long, and a subgroup with number of members ~224 bits long
- Protect the private key by computing an "ephemeral" key in the subgroup
- Ephemeral key must only be used once
  - Sony PlayStation-3 private key was compromised due to ephemeral key reuse

https://arstechnica.com/gaming/2010/12/ps3-hacked-through-poor-implementation-of-cryptography/

http://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/

The security of the DSA signature is also based on the discrete logarithm problem (like Diffie-Hellman) or the generalized discrete logarithm problem (elliptic curve.)

The size of the modulus (2048 bits) and the subgroup (224 bits or more) is the same as regular Diffie-Hellman. If you use elliptic curves the keys can be much smaller, just as in Elliptic Curve Diffie-Hellman (ECDH.)

In addition to the private key, there is also an "ephemeral" key. Think nonce, initialization vector, or salt. It can only be used once. Any time you sign more than one message with the same ephemeral key, your signature can be broken.

DSA Signature

- Signature is two numbers (usually called r, s) calculated from:
  - Ephemeral key
  - Private key
  - SHA hash of message
- Signature is verified with similar calculation (but with the public key)
  - https://www.di-mgt.com.au/public-key-crypto-discrete-logs-4-dsa.html
- Good description of ECDSA (elliptic curve DSA) here:

http://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/

In DSA, the public key consists of:

1. A prime number p, ~2048 bits long
2. A divisor q, ~244 bits long, that divides p-1
3. α, which generates the group with q elements
4. $\beta = \alpha^d \bmod p$, where d is the private key. (Remember DHKE, where Alice gave $A = \alpha^a$ to Bob, and Bob gave $B = \alpha^b$ to Alice? It's the same idea.)

The private key is d.
The public key is p, q, α, β
The signature is two numbers this time, usually called r and s

Remember that there is also an ephemeral key used in computing the signature (r and s) that is kept secret and never reused.

Message Authentication Code (MAC)

- MAC uses a symmetric key instead of a public/private key pair
- MAC does not provide nonrepudiation
  - Key is shared, cannot say message came from a single person
- MAC does provide assurance of message integrity
  - If MAC is correct, message was not altered in transit
- MAC does provide message authentication
  - Person that computed MAC had to know symmetric key
- Just like symmetric encryption, computing MACs is much faster

Since MACs use symmetric keys, they do not provide nonrepudiation. The do provide authentication and integrity. Just like AES, they require much less computation than signatures (public key) do.

Every packet in an encrypted communication should have a MAC along with it. This proves the message hasn't been tampered with, and it prevents an attacker from inserting crafted packets into our channel. Remember WEP? WEP did not protect against attackers inserting crafted packets, and could be broken very quickly.

Usually, signatures are used in setting up the encrypted channel. Once the channel is established, nonrepudiation is no longer an issue. We can use the much faster MACs to ensure that an attacker cannot do an MITM attack against us. This is similar to the way encryption is done; use public key encryption to exchange a symmetric key and then use AES to encrypt the data.

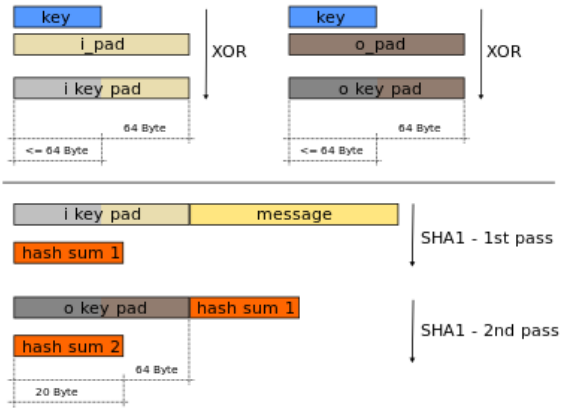
HMAC

- HMAC: hash-based MAC
  - Uses SHA hash function
  - Used in TLS
- Vulnerable to attack if message hash is just attached to message
- Message, padding, and key are hashed together, twice
- Hash function used is a version of SHA, often SHA-1 or 2

https://en.wikipedia.org/wiki/HMAC

https://tools.ietf.org/html/rfc2104

Unfortunately, you can't just attach a hash to a message and use it as a MAC. There are attacks that can break that. Instead, we have to use a block diagram just like we did for RSA (PKCS#1 and OAED OR EMSA).



Again, i_pad and o_pad are commonly known. Alice combines them with the key (symmetric encryption, one key) to create i_key_pad and o_key_pad. Then i_key_pad is prepended to the message and she hashes the result with SHA. She appends o_key_pad to the output and then hashes it again with SHA.

Bob also knows the key, and can create the same HMAC of the message. If the HMAC is the same as the one Alice sent him, he knows Alice sent it (Alice and Bob have the key) and that the message hasn't been tampered with.

Other MACs

- CBC-MAC Cipher Block Chain
  - Similar to CBC symmetric encryption, each block of plaintext is XOR'd with ciphertext of previous block
  - Unlike CBC encryption, only the last block of ciphertext is included as the MAC
- GMAC Galois Counter MAC
  - Similar to Galois Counter Mode symmetric encryption
  - Also does not provide encrypted output, just MAC
  - Very fast, can be computed in parallel
  - Used in IPSec (encrypted IP) because of speed

Many other MACs run a symmetric encryption algorithm on the plaintext in CBC mode. Remember that in CBC, the input to each block of encryption is the plaintext for that block xor'd with the output of the previous block. A change in the plaintext in any block will cause the output of the final block to change. Only the last block of ciphertext is included in the MAC; the previous blocks are all discarded.

The Galois Counter Mode MAC is very similar to the GCM mode in AES. GMAC has the advantage that much of it can be computed in parallel, so that it is very fast. The underlying math is based on Galois Extension Fields, where addition and multiplication are replaced by operations on polynomial coefficients. (The Galois field math is used to derive the algorithms, then the algorithms are implemented in software.)

IPSec is a protocol that can be used in layer 3 of networking in place of IPv4, or as an option in IPv6. It provides encryption. It is most often used in VPN connections between routers (site to site VPN) but can be used in other places. Since it is built in to the networking stack, it needs to be very fast.

Hash Functions

- Compute a short digest of a longer message/document
- Hashes are also used in computer science (hash tables)
  - Map variable length fields to fixed length index
  - Speeds access
  - Evenly distributes data across storage locations
  - Sometimes used in load balancing
- Cryptographic hashes have specific design goals, very different from hash tables

Both cryptography and computer science have hash functions, but they are quite different. Some capture the flag contests have conflated them.

Computer science often uses hash functions to increase the efficiency of storage tables when the index has variable length. If the index to a table is a person's last name, the index varies considerably in length, and is not distributed evenly. For example, there are a lot of Smiths and last names that start with S, but few that start with Q, X, or Z. A hash function will turn the last names into random-looking strings of the same length so they can be stored more efficiently. The common hash table (hash mapping in Java, dictionary list in Python) stores name/value pairs, and uses this technique on the index, name.

Cryptographic hash functions are specifically designed to be resistant to attacks. A good metaphor would be the Cyclic Redundancy Checksum (CRC) and the Message Authentication Code (MAC). Both CRC and MAC detect errors introduced by noise in the communication channel. However, the MAC must also resist attackers that deliberately try to undermine it. It is a trivial matter for an attacker to fool CRC; WEP used CRC instead of MAC and was completely broken.

Cryptographic Hash Goal—One-way

- When a message is hashed, it should be very hard (computationally infeasible) to determine what the original message was. This is also called pre-image resistance.
- Example: password hashes. An attacker should not be able to compute the password if they know the hash.
- Note:
  - Hash cracking uses a dictionary of common passwords and computes the hash of each password
  - It compares the hash from the dictionary password to the hash being cracked
  - Hash cracking does not compute the password from the hash, but is a type of brute force attack

Cryptographic hashes must be one-way. Once plaintext has been hashed, it should be impossible to determine the plaintext by looking at the hash. The entire reason for storing the hash of a password instead of the password itself is that it should be impossible to compute the password from the hash, i.e., one-way.

Hash cracking does not compute the password from the hash. Instead it uses the fact that people are not good at using random passwords. They tend to choose passwords from dictionary words, and perhaps add a number or symbol to the beginning or end to satisfy password checkers. (The password 'Winter2019!' satisfies many password complexity checkers, but is easy to crack.)

Hash crackers choose from a list possible passwords, and compute the hashes of the possible passwords until they find a hash that matches the target. It is a brute force attack made possible by people's poor choices of passwords.

1) Use Two Factor Authentication (2FA) or MFA (the M is multifactor) instead of a plain password.
2) Use a password manager (KeePass is a good one) so that you can use a separate random password for every account.

Cryptographic Hash Goal—Collision Resistance

- Collision: when two messages have the same hash
  - message length >> hash length, so collisions exist
  - There are an infinite number of messages that have the same hash
  - Finding a message with the same hash should be hard
  - Brute force search ≈ 2^(hash bit length/2) messages
- When a message is hashed, it should be very hard to modify the message and keep the hash unchanged
- Hashes are the basis for much of cryptography, so hashes are important

Since the message length can be anything (several TB, even) and the hash length is short (160 bits for SHA-1, 256 bits for SHA-256, etc.) it stands to reason that there must be files/message that have the same hash. These are called hash collisions.

The key is that it must be very hard (again, computationally unfeasible) to create two messages that have the same hash, or modify a message so that the hash remains unchanged.

If the hash were 160 bits long, you might think that the space you have to search to find a duplicate hash would be 2^{160}. However, because of the statistical problem called the birthday paradox the space is 2^{80}. The birthday paradox is simple: how many people do you need to have in a room to have a 50% probability of having two people with the same birthday? You might guess that you'd need about 365/2 people. Math shows that you only need 23 people to have a 50% chance of two people with the same birthday. (Paar, pg 300).

Current computers can handle searches on the order of 2^{60} or higher. SHA-1 should be safe against a pure brute force attack. However, faster attacks can reduce the search space enough that SHA-1 can be attacked.

Common Hash functions

- Obsolete
  - Researchers have found collision attacks for short length hashes
  - MD5 (Message Digest 5, by Rivest) 128 bit output, broken long ago
  - SHA-1 (NIST Secure Hash Algorithm 1) 160 bit output
    - Theoretically broken 2005, Google found practical attack in 2017
- Currently secure, SHA-2
  - Variants SHA-224, SHA-256, SHA-384, and SHA-512
  - SHA-###, where ### is number of output bits
- New version, SHA-3 released by NIST in 2015
  - Increased security, totally different method from SHA-2
  - Same output lengths, SHA3-224, SHA3-256, SHA3-384 and SHA3-512

The MD5 hash was broken, mainly because the length is too short. At 128 bits, the brute force space is 2^{64}. Different attacks have reduced that space to 2^{24} and 2^{39}, which is well within the capability of modern laptops. The MD5 hash is still useful when you are just verifying a message was not corrupted by noise, but it should not be used where an attacker may manipulate it.

SHA-1 was first broken by Google in 2017, so its use should be limited as well.
https://shattered.io/
https://www.schneier.com/blog/archives/2012/10/when_will_we_se.html
https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html
https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

The SHA-2 variants (SHA-224, SHA-256, SHA-384, and SHA-512) are safe for use. NIST is anticipating a time when SHA-2 hashes may be broken by computing advances or the discovery of faults, and conducted a competition in 2006 to find an alternative hash function. The winner was the Keccak algorithm, which is now SHA-3. SHA-3 applications are not commonly available by default, but applications are available on the Internet.

Linux Password Hashes

- Password hashes were once stored in /etc/passwd, world readable
  - jsmith:hash_used_to_be here:1001:1000:Joe Smith,Room 1007:/home/jsmith:/bin/sh
  - Hash could be cracked by dictionary attack against common passwords
- Hashes moved to /etc/shadow, which is only readable by root
  - jsmith:x:1001:1000:Joe Smith,Room 1007:/home/jsmith:/bin/sh
- Format of hash stored in /etc/shadow
  - idsalt$hashedpassword
  - $id tells what hash algorithm is used

In the 1980's, Linux kept both the user information and the user's password hash together in the /etc/passwd file, which is readable by any account on the machine. It was thought that hashes were invulnerable to hashes. Once dictionary attacks were discovered (compute hashes of dictionary words and then compare to the hash under attack) the hashes were moved to a new file, /etc/shadow. The shadow file is only readable by root, which prevents non-root users from attacking the password hashes. See the book, "Cuckoo's Egg" by Clifford Stoll.

There are several hash algorithms in use, so an identifier at the beginning specifies the algorithm. https://www.cyberciti.biz/faq/understanding-etcshadow-file/

Linux salt

- Need to add a random element to password hash
  - Otherwise, everyone who sets their password to “password” will have the same hash
  - Otherwise, attacker could pre-compute hashes for common passwords
- Salt is random, included in /etc/shadow without encryption
  - Salt is not secret, it just makes hash change each time
- /etc/shadow for user test, with super-secure password = “password”
  - test:6.CKfcoNo$FEhoYxtEtBLg1UoSqXulMJMaP/xRk2L2.QLF3zhUgOtF9HSy6TMEM1cCep1STf6WXl40Wel0pKhTK976sUFDU.:17688:0:99999:7:::
  - 6 says hash is SHA-256
  - Salt is .CKfcoNo
  - Password hash is FEhoYxtEtBLg1UoSqXulMJMaP/xRk2L2.QLF3zhUgOtF9HSy6TMEM1cCep1STf6WXl40Wel0pKhTK976sUFDU

Salt your hash

Linux adds a random number called a salt to the password before it hashes it. The salt is not secret; it is stored in unencrypted form along with the hash. Its purpose is to ensure that the same password receives a different hash every time the hash is computed. This forces an attacker to attack every password separately, and stops the attacker from precomputing password hashes.

Without a salt, the attackers could compute the hash for the password “password” ahead of time. Then, every time they saw that hash in an /etc/shadow file, they would know the user’s password was “password”. The salt is large enough that it is infeasible to store precomputed hashes.

Windows Password Hash

- Stored in SAM database
  - Supposed to be inaccessible, but there are tools…
- Does not use a salt!!!
  - Passwords and hashes can be pre-computed and stored in Rainbow Tables
- Two hash types, LM and NTLM
  - LM disabled by default in Vista and higher
- Never allow LM, it is easy to crack
- NTLM is easy to crack for short passwords, difficult for long passwords not made of dictionary words

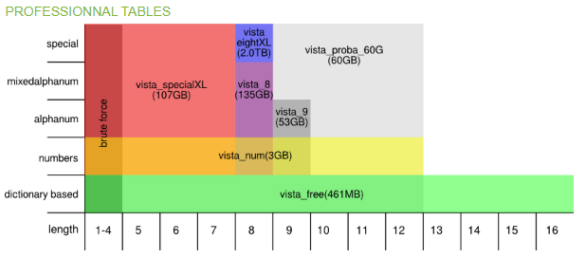
Windows password hashes are stored in the registry hive HKEY_LOCAL_MACHINE, or HKLM, in the SAM key (HKLM\SAM). Originally the hashes were encrypted with syskey, and syskey could be configured to demand a password when the machine booted so the hashes could be decrypted. Hardly anyone did this, and Microsoft "hid" the syskey password in the registry so machines could boot without the owner entering a password. The location of the syskey password was soon discovered in HKLM\SYSTEM, and many tools can now decrypt the hashes in the SAM database.

LANMAN, or LM hashes were disabled by default from Vista onwards, and should never be used. They are easy to crack.

NTLM hashes (or NTHash) are the standard password hashes for Microsoft. They are based on the old MD4 algorithm and are unsalted. Hashes can be computed very quickly compared to the modern Linux SHA-512 hash ($6), and can be computed ahead of time.

You will also see NTLMv1 and NTLMv2. These are not hashes, but challenge-response protocols that Microsoft uses to authenticate between computers. The NTLM hash can be extracted from NTLMv1 and v2 (NTLMv2 is current, v1 is deprecated.)

Rainbow Tables for Windows Hashes



https://www.objectif-securite.ch/en/ophcrack.php

- Precomputed Windows hashes (Rainbow Tables) are available from several locations
- The best defense is to use a long password, > 12 characters (longer if a nation-state is cracking your passwords.)

If you look at the small blue square, you'll see that you can purchase a 2.0 TB file called vista eightXL. It can crack almost all NTLM hashes of passwords that are 8 characters long and contain upper and lower case, numbers, and special characters. The typical complexity requirements for Microsoft accounts are eight characters contain three of the four categories (upper, lower, digits, and special.) This is not good.

The best thing you can do to prevent your passwords from being cracked is to make them long (>10-20 characters) and use more than dictionary words.

The best policy is to use a password manager like KeePass. Use a very long password (or two factor authentication) to open the manager. Have the manager create long random passwords for all your accounts.