Cryptography problems

Michael Sonntag
Institute of Networks and Security

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

Insecure cryptographic storage

- If there is cryptography (and it is not extremely weak), attackers will never target it directly: too much effort required
  - They will look for the keys, a place where the data is “momentarily” not encrypted, some auto-decrypt functionality…
- Any kind of “cryptographic material” is very important
  - Key generation: real random numbers should be used
  - Key storage: is the key itself encrypted?
  - Rotation: keys and certificates must be changed regularly
  - Hashes: no weak algorithms
  - Pwd→Key: salting should be used (better: key derivation function)
- Biggest problem: if you do **some** encryption, the data is probably quite important
  - A **bit** of encryption is **worse** than **no** encryption: you get a false sense of security!
  - So either do it correct, or don’t do it at all!

Insecure crypto storage: Examples

- Keys are stored directly in the program code or in the registry
  - Everyone who can read the file/registry can easily discover this fact and extract the key
- Backups are encrypted and the key is on the same medium
- Database with column encryption
  - Automatic decryption for queries → anyone with access to the database somehow (→ SQL injection!) can read these columns
  - Encryption should be external
    - Pass in the key as parameter or decrypt in the application
- Passwords are weakly hashed or do not use salting/repetitions
  - Rainbow table/brute force attacks!
- Certificates are used, but it is not verified who issued them
  - Or that they are issued by whom they are expected to be
- PWDs in configuration files - which are in a source code repository

Insecure crypto storage: Detection

- Code inspection:
  - Identify all data that needs encryption
  - Find all places where it is stored: there it should be encrypted
  - Check where the key(s) for this data are stored
    - Are they encrypted and salted? How can they be decrypted? Who can do this (→ automatic or tied to an account)?
  - Check the encryption algorithm (→ FIPS 140-2)
    - Only strong and standard algorithms and modes should be used
    - Check that it is an up-to-date standard implementation
    - E.g. does it include DH (→ forward secrecy)?
  - Check security of errors (messages, data deleted, logging…)
  - Verify that good random number generators are used
  - Enforce guidelines for the lifecycle of keys
    - Generation, distribution, revocation, expiration (=secure deletion!)
- Make sure that any encryption/signing/… takes place on the server and not on the client

Insecure crypto storage: Prevention

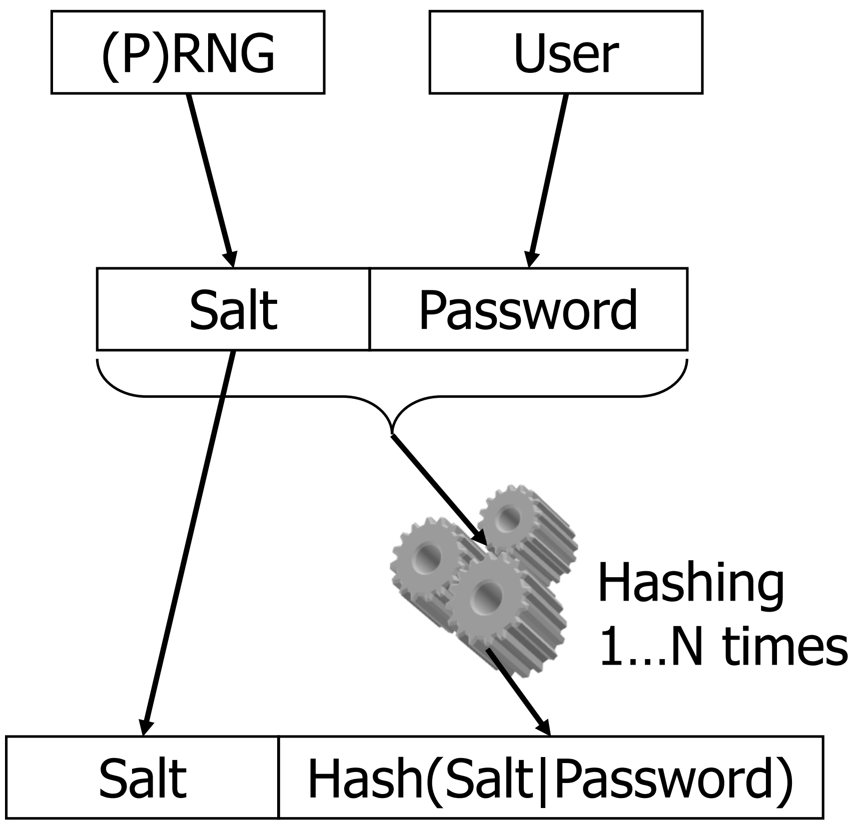
- Do not implement your own cryptographic library
- Never invent your own algorithm
  - Use only known good algorithms
  - Make sure the algorithm can be changed (securely!) easily
- Identify potential attackers and what data they might have access to: insiders, web server hacked, root hacked…
- Take great organizational care: key management is less a technical than an organizational issue
  - But also don't make it too cumbersome → people circumvent it
  - Example: backups should be encrypted, but the keys used for this should be stored (and backed up!) separately
- Enforce password/key strength and use salting
- Protect important data against unauthorized access
  - This should be checked by the application!

Insecure crypto storage: Passwords

- How to store passwords in a database
  - Create **new** random salt value for **each** new password (not: user!)
  - Store the salt in plain text
  - Concatenate salt and password and hash it
    - Securely: don't use MD5!
    - Better: use a "Key Derivation Function" (deliberately slow!)
  - Store the hash value in the database (alongside the salt)
- Checking passwords:
  - Look up the salt based on the username entered
  - Concatenate salt and entered password and hash it
  - Compare result with value from database
- Password recovery is then not possible anymore
  - Define methods for assigning a new password
    - Generating a random one and sending it per E-Mail, sending a link for resetting… → all insecure (but usable!)
    - Better: help desk + strict verification of person/caller → reset

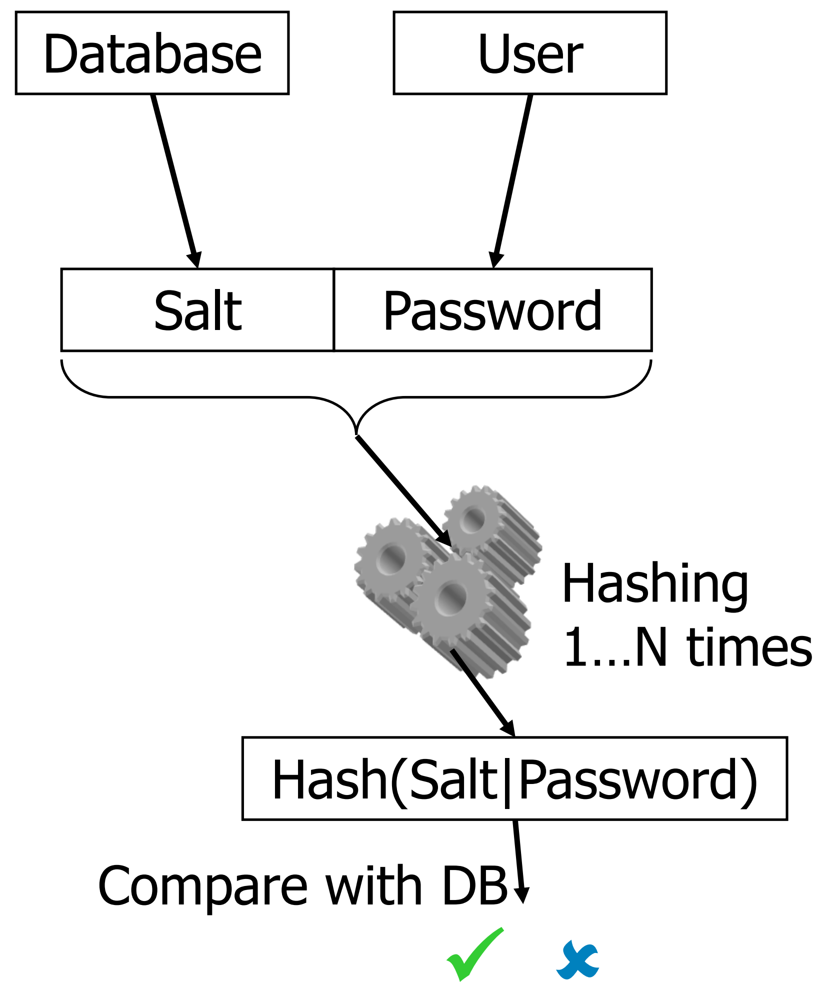
Insecure crypto storage: Passwords

Store password



Stored in DB
(Note: salt is cleartext!)

Check password



PBKDF

- PBKDF: Passphrase-Based Key Derivation Function
  - Even better than the measures described before
- Potential problems of previous method:
  - Hashes are fast and require little memory → dedicated hardware can (and has) been built to break hashes (see also GPUs)
- Solution: use a „hash“ function that is slow and requires lots of memory to compute
  - This does not solve the problem, but can make these kinds of attacks pointless, as too slow and expensive!
- Typical algorithms:
  - PBKDF2: small circuit and little RAM sufficient
  - scrypt: CPU/memory cost and parallelization can be set
  - Argon2: time, memory, and parallelization can be controlled
    - Specific versions against GPU cracking or side-channel attacks

Weak random number generation

- Deriving PRNG seed mt_rand from PHPSESSID
- PHPSESSID = md5(client IP . timestamp . microseconds1 . php_combined_lcg())
- php_combined_lcg() has two seeds
  - S1=timestamp XOR (microseconds2 << 11)
  - S2=pid XOR (microseconds3 << 11)
- Values:
  - client IP: attacker knows it
  - timestamp: disclosed in HTTP header; own clock (NTP!)
  - microseconds1: 0-999.999 (=1.000.000 values) ≈ 20 bit
  - microseconds2: microseconds1 + 0…3 ≈ 2 bit
  - microseconds3: microseconds2 + 1…4 ≈ 2 bit
  - pid: process id of current process; 0…32.767 ≈ 15 bit
- Result: brute-force is possible (39 bits!) - only PID and microseconds!
  - It is also doable in reality: Amazon EC2 GPU instance: 7,5 min

2012!

Reutov/Yunusov/Nagibin, http://blog.ptsecurity.com/2012/08/not-so-random-numbers-take-two.html

Weak random number generation

- Microseconds can be reduced in scope as well
  - Synchronize clocks through requests: send two requests so that the second one shows a timestamp one second later
  - Try several times with slight modifications → microseconds synchronized to almost zero!
- Result:
  - Prediction of future PHPSESSID values becomes possible
  - Seed for mt_rand was also based on timestamp, pid, and php_combined_lcg() → even more random number predictions possible
- Reason: too little entropy in session id generation
  - Values predictable and related to each other
- Prevention:
  - Use other (=secure) random number generators
    - /dev/urandom, openssl_random_pseudo_bytes

Bogk, http://berlin.ccc.de/~andreas/php-entropy-advisory.txt

Weak random number generation

- This is a special problem for virtual servers
  - If you are running a program on the same virtual server, you can get the (almost) identical time from your VM
  - This cannot be prevented, as both VMs must have a correct time set in them (or many things will fail)
    - If deliberately fudged slightly, this can be easily detected and accounted for
  - Using better random sources then becomes even more important
    - But also more difficult, as VMs often only get a significantly lower rate of “real” randomness
      - E.g. no direct access to disks or physical sensors

Certification Authority Authorization

- Which CA may issue a certificate for your website?
  - Any CA which desires to do so!
  - This is good for getting "some" certificate
  - But a rogue (or hacked) CA can issue a certificate for you too!
  - Result: MitM attack becomes possible
- Countermeasure: specify in DNS which CA is "acceptable"
  - This means that DNS requests to the client must be modified too for a successful attack
    - Therefore especially useful in combination with DNSSEC!
  - Implementation: add a new DNS record
    - "sld.tld. CAA 128 issue "letsencrypt.org"" → normal certificates
    - "sld.tld. CAA 128 issuewild "letsencrypt.org"" → wildcard certif. are OK
    - "128": Flags byte with Bit 7 set → "critical" → must be followed
      - New option "iodef" → report invalid certificate requests; not supported by all CAs
        - sld.tld. CAA 0 iodef „mailto:security@sld.tld"

Certification Authority Authorization

- Security aspects:
  - Slight advantage only: the CA itself must check for it
    - A rogue CA would simply not check!
    - Want your certificates in browser? You **have** to commit to checking!
  - No danger: if you mess up, you just have to change the DNS entry, which is typically easy and fast
    - No time limit/validity duration in this record!
- Where does it help:
  - Someone messes up, mistypes etc
  - Incorrect automated issuing of certificates
  - CA is hacked regarding "authorization" only, but not in the "issuing process"
- Potential improvement:
  - Browsers could check this too: is the certificate I received from one of the "approved" CAs?

See https://sslmate.com/caa/

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at