3.6 Database Security

This section covers various basic aspects of security revolving the database functionality. Security is always evolving so keep yourself updated!

- Secure password storage & Basic Password Cracking
- Authentification & HTTPS (with Tomcat)
- DB security risks and countermeasures

SECURE PASSWORD STORAGE

Problem: How to store passwords in a database?

- Storing login credentials (e.g. for Web services) is a typical application for databases

User	Password
John Doe	securepw1
Trudy	123

- Storing passwords in plain text opens up security risks
  - When attackers get (partial) read access
  - Different kinds of attacking strategies: Dictionary attacks, Brute-force ...

Naive Solution

- Choose a cryptographic hash function (e.g. MD5, SHA1, ...)
- Store the passwords not in plain text, but as a hash value

User	Password
John Doe	a0719618388bf24f0d89b923df477712
Trudy	202cb962ac59075b964b07152d234b70

- On login: compute hash of input and compare

Cryptographic Hash Functions

Cryptograhic hash functions are "one-way" mathematical functions that are infeasable to invert

- Arbitrary size input "m"
- Fixed size output "h"

$\rightarrow$ hash(m) = h

But! there is no way to prove that a function is not invertible

$\rightarrow$ Difference of "it cannot be broken" and "nobody knows how to break it"

Properties of cryptographic hash functions

- Deterministic
- Given a hash value, it is infeasable to generate the message (pre-image resistance)
- It is infeasable to find two messages with the same hash value (collision resistance)
- Given a message, it is infeasable to find a different message with the same hash value (second pre-image resistance)

Use cases of cryptographic hash functions

- Verifying the integrity of messages and files
- Signature generation and verification
- Password verification
- Proof-of-work (deter DOS attacks, crypto-currency)
- File or data identifier

Attacks on Hashed Passwords

- You should always assume that the attacker knows everything except the plain password!
  - One or a collection of hashes of passwords
  - The algorithm used for the hashing

General Types of Attacks

- Preimage attack
  - Find a message with a specific hash value
  - For an ideal hash function the fastest way to compute a first or second preimage is through a brute-force attack

$\rightarrow$ For n-bit hash $\Rightarrow 2^n$ complexity

- Birthday attack (collision attack)
  - „It is more likely to find two random messages with the same hash value than the message for one specific hash value“
  - Complexity $2^{n/2}$

Bit-length	Possible outputs	75% chance of random collision
16	$2^{16} =\sim 6.4x10^{4}$	430
128	$2^{128} =\sim 3.4x10^{38}$	$3.1x10^{19}$
512	$2^{512} =\sim 1.3x10^{154}$	$1.9x10^{77}$

Attack example

- Naive attack: Precompute every possible password for a given hash function and then just look them up
  - Saves computing time when looking up multiple hashes
  - Costs an infeasible amount of space

    Example: Consider all possible combinations of 62 different letters [A-Za-z0-9] in 8 positions = 62^8

    Each pair needs the space of $\sim$ 24 Bytes (16 for the MD5 hash, 8 for the plain text in UTF-8)

  $\Rightarrow$ $\sim$ 4766 Terabyte

$\rightarrow$ Rainbow table

- Precomputed table for reversing cryptographic hash functions
- Chains of passwords & hashes to reduce space usage
  - $\star$ Time-space trade-off
  - $\star$ Increasing the length of the chain, decreases the size of the table, but increases time for look-ups

- Rainbow table
  - Usage of reduction functions (r1,r2,...) to reverse a hash value back into plain text (not the real inverse!)

    $$Plain_1 \underset{\text{h}}{\rightarrow} Hash_1 \underset{\text{r1}}{\longrightarrow} Plain_2 \underset{\text{h}}{\rightarrow} Hash_2 \underset{\text{r2}}{\longrightarrow} Plain_3...$$
  - Only store starting point and endpoint
  - For a given target hash value calculate the chain with it and compare to the stored endpoints
    - On a hit you know that the password might be inside the chain which can be recalculated from the starting point
    - It is not guaranteed due to collision in the Reduction-functions
  - To decrease collisions in the hash chains more than one reduction function are used periodically

Salted Hashes

- Assume that there are Rainbow tables, etc. for every standard hash function
- The attacker has the advantage of parallelism
  - Hash one PW and compare it to a lot of the stored PWs
  - Shares the cost of hashing over several attacked PWs
- Solution: Make the hash function individual for every user

$\Rightarrow$ Salted Hashes

- Add a unique code to every PW to break the hash function into different „families“ of hash functions

  Hash(m + salt) = h

- Breaks the parallelism advantage of the attacker
- **But!** Every user has to have an unique salt or else you could create Rainbow tables for the salted hash

  $\rightarrow$ If the PW is used again on a different platform, it should have a different salt

- How to generate salts that are as unique as possible?

  $\rightarrow$ Use randomness!

Salt Generation

- Cryptographically Secure Pseudorandom Number Generators (CSPRNG)
  - "Quality" of randomness required varies for different applications
    - Nonce require only uniqueness
    - One-time pads require also high entropy
  - Uses entropy obtained from a high-quality source
    - Operating system's randomness API
    - Timings of hardware interrupts, etc.

- Universally Unique Identifier (UUID)
  - 128 bit number, representation in 32 hexedecimals in 8-4-4-4-12 format
    - 123e4567-e89b-12d3-a456-426655440000
  - Often used as database keys
    - Microsoft SQL Server: NEWID function
    - PostgreSQL: UUID datatype + functions
    - MySQL: UUID function
    - Oracle DB: SYS_GUID function (not quite a standard GUID, but close enough)

Aside: Pepper

- A salt, but secret!

⇒ Just like a key

- Only increases security if the attacker has access to the hash, but not the pepper

→ Store pepper on a different "secure" hardware

Aside: "broken" MD5

- The MD5 Hash-function is considered broken

⇒ It is "easy" to find collisions

- But password hashing is not concerned about collisions
  - Preimage attacks are important!
- MD5 has other problems in that regard

→ One of the fastest cryptographic hash function to compute

Brute-force attacks

- Recall:
  - An ideal hash function has complexity 2^n to find the message of a specific hash value

- But:
  - What if these hash values can be computed really fast?
  - Modern hardware can compute millions of "easy" hash values in mere seconds

Slow Hash Functions

- Counter faster & faster hardware
  - Make deliberate slow algorithms

⇒ Key Derivation Function (KDF) with sliding computational cost
  - Hash = KDF(pw , salt , <u>workFactor</u>)

  - PBKDF2
  - bcrypt
  - scrypt
  - Argon2
  - ...

  - How many iterations?

  → As many as possible without hurting the user

PBKDF2

- Password-Based Key Derivation Function 2
  - Combines
    - A hash-based message authentication code (HMAC) function (MD5, SHA1,...)
    - Salt
  - Iterates a predefined time
    - Recommended in 2000: 1000 iterations
    - Recommended in 2011: 100000 iterations

bcrypt

- Based on the Blowfish block cipher
  - Eksblowfish (expensive key schedule Blowfish)
    - Use PW & Salt to generate a set of subkeys (P-array & S-box)
    - Iterate depending on the specified cost
  - Iterate 64 times:
    - Use standard Blowfish algorithm in ECB (Electronic Codebook) mode
    - Block encryption with the set of subkeys and the text "OrpheanBeholderScryDoubt"
  - Password length of up to 56 bytes
  - Uses 4KB RAM

Time-Space Tradeoff

- Specialized hardware is extremely efficient at multi-threading
  - Field Programmable Gate Arrays (FPGA)
  - GPUs

- But experience difficulties when operating on a large amount of memory

  $\Rightarrow$ Design memory-hard functions with exponential memory usage
    - scrypt
    - Argon2
    - ...

scrypt

- Used as proof-of-work algorithm in many cryptocurrencies (e.g. Dogecoin)

- Uses $PBKDF2_{HMAC-SHA256}$ amongst other algorithms
- Generates a large vector of pseudorandom bit strings which are accessed in pseudo-random order to produce the derived key

$\rightarrow$ Trade-off:

- Store the vector (high memory cost)

  vs

- Generate the elements of the vector as needed (high computational cost)

Argon2

- Winner of the Password Hashing Competition (PHC)(2013-2015)
- Based on the Blake2b hash function
- Variants of Argon2
  - Argon2d
    - data-dependent memory access
    - highest resistance against GPU cracking attacks
    - possible side-channel attacks
  - Argon2i
    - data-independent memory access
    - safest against side-channel attacks
  - Argon2id
    - hybrid of Argon2d & Argon2i

Closing Words of Advice

- Home-brew vs public standard hash algorithms
  - "Security through obscurity" (does not work!)
    * Code gets reverse engineered
    * Algorithm should be secure even if all information except the PW is known
    * Lots of testing on public algorithms

    $\rightarrow$ Still deemed secure even after many years

- Common or short passwords kill every secure hash algorithm
  - Recommended: 128 bit (of entropy) $\sim$ 22 chars

Implementation: How to

- CSPRNG in Java:
  - Java.security.SecureRandom
    - Seeds automatically
    - Uses the secure random function of an installed security Provider (e.g. SUN)

```java
import java.nio.charset.Charset;
import java.security.*;
import java.util.Arrays;

public class PasswordHash {
    public static void main(String[] args){
        //Checks the installed security Providers
        Provider[] providers = Security.getProviders();

        for(Provider prov : providers){
            System.out.println(prov.getName());
        }

        //Use an SecureRandom object
        SecureRandom sr = new SecureRandom();
        //SecureRandom sr = SecureRandom.getInstanceStrong();
        //SecureRandom sr = SecureRandom.getInstance("SHA1PRNG", "SUN");

        byte[] salt = new byte[20];
        sr.nextBytes(salt);
        System.out.println(Arrays.toString(salt));
        System.out.println(new String(salt,Charset.forName("ISO-8859-1")));
    }
}
```

[Filename: Servlet/PasswordHash.java]

- Argon2 in Java:
  - Original implemented in C
  - Two Java bindings:
    - https://github.com/phxql/argon2-jvm
    - https://github.com/kosprov/jargon2-api
  - Best included via Maven

```
<dependencies>
    <dependency>
        <groupId>com.kosprov.jargon2</groupId>
        <artifactId>jargon2-api</artifactId>
        <version>1.1.1</version>
    </dependency>
    <dependency>
        <groupId>com.kosprov.jargon2</groupId>
        <artifactId>jargon2-native-ri-backend</artifactId>
        <version>1.1.1</version>
    </dependency>
</dependencies>
```

Aside: Maven in Eclipse

- Maven plugin should be pre-installed
  - If not: Help → Install New Software...
  - Search for "m2e"

- Convert project into Maven project
  - Right Click → Configure → Convert to Maven Project ...

- Add listed dependencies to the project
  - Right Click -> Maven -> Add Dependency
  - OR: Add them manually to the pom.xml

- Argon2 in Java:
  - Follow instructions in the chosen repository (E.g. Jargon2)

```
import static com.kosprov.jargon2.api.Jargon2.*;
import java.util.Arrays;

public class TestArgon2 {
    public static void main(String[] args) {
        byte[] salt = "this is a salt".getBytes();
        byte[] password = "this is a password".getBytes();

        Type type = Type.ARGON2d;
        int memoryCost = 65536;
        int timeCost = 3;
        int parallelism = 4;
        int hashLength = 16;

        // Configure the hasher
         Hasher hasher = jargon2Hasher()
                .type(type)
                .memoryCost(memoryCost)
                .timeCost(timeCost)
                .parallelism(parallelism)
                .hashLength(hashLength);
```

```java
        // Configure the verifier with the same settings as the hasher
        Verifier verifier = jargon2Verifier()
                .type(type)
                .memoryCost(memoryCost)
                .timeCost(timeCost)
                .parallelism(parallelism);

        // Set the salt and password to calculate the raw hash
        byte[] rawHash = hasher.salt(salt).password(password).rawHash();

       System.out.printf("Hash: %s%n", Arrays.toString(rawHash));

        // Set the raw hash, salt and password and verify
        boolean matches = verifier.hash(rawHash).salt(salt).password(password).verifyRaw();

        System.out.printf("Matches: %s%n", matches);
    }
}
```

[Filename: Servlet/TestArgon2.java]

Regulars' table (Stammtisch) Knowledge

- Char[] is more secure than String
  - Strings are immutable

  ⇒ There is no way to delete it from memory before the Garbage Collector kicks in

- Allowing ultra long passwords enables DOS attacks
  - Passwords can be hashed beforehand to prevent that (e.g. with SHA-512)