Secure Password Storage

How to store passwords in a database?

Introduction

- Storing login credentials
  - Webservice

User	Password
John doe	securepw1
Trudy	123

- Security risks
  - Attacker gets (partial) read access
  - Dictionary attacks, Brute-force attacks

Naive Solution

- Choose a cryptographic hash function
  - MD5, SHA1, …
- Password not stored in plaintext, but hash value
- On login: compute hash and compare

User	Password
John doe	a0719618388bf24f0d89b923df477712
Trudy	202cb962ac59075b964b07152d234b70

Cryptographic Hash Functions

- „One-way“ mathematical function that is <u>infeasable</u> to invert
  - Arbitrary size input
  - Fixed size output
- hash(m) = h
- There is no way to prove that a function is not invertible
  - Difference „it cannot be broken“ and „nobody knows how to break it“

Cryptographic Hash Functions

- Properties
  - 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)

Cryptographic Hash Functions

- Use cases
  - 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

- Preimage attack
  - Find a message with a specific hash value
  - For an ideal hash function the fastes way to compute a first or second preimage is through a brute-force attack
    - For n-bit hash => 2^n complexity

Attacks on Hashed Passwords

- 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	$2^{16} = \sim 6.4 \times 10^4$	430
128	$2^{128} = \sim 3.4 \times 10^{38}$	3.1×10^{19}
512	$2^{512} = \sim 1.3 \times 10^{154}$	1.9×10^{77}

Attacks on Hashed Passwords

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

Attacks on Hashed Passwords

- Rainbow table
  - Usage of reduction functions to reverse a hash value back into plaintext (not inverse!)
    - $Plain_1$ -> $Hash_1$ -> $Plain_2$ -> $Hash_2$ -> ...
    - Only store start point and end point
    - Calculate chain with given hash value and compare to endpoints
  - Rainbow tables use more than one reduction function to decrease collisions in hash chains

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

Salted Hashes

- Solution: Make the hash function individual for every user

  => Salted Hashes

- Add a unique code to every PW to break the hash function into different „families“ of hash functions
- Hash(m + salt) = h

1)

Salted Hashes

- 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
  - If the PW is used on a different platform, it should have a different salt
- How to generate salts that are as unique as possible?
  - Use randomness!

Salt Generation

- Cryptographically Secure Pseudorandom Number Generators
  - "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.

2)

Salt Generation

- 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)
        - Hash = KDF(pw, salt, workFactor)
            - PBKDF2
            - bcrypt
            - scrypt
            - Argon2
    - How many iterations?
        - As many as possible

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
  - Iterate:
    - Use alternating PW and Salt
    - Block encryption with the set of subkeys
    - Replace some of the subkeys

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

=> Design memory-hard functions with exponential memory usage

  - scrypt
  - Argon2

Outro

- 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
      - Still deemed secure even after many years
- Common or short passwords kill every secure hash algorithm
  - Recommended: 128 bit (of entropy) ~ 22 chars

How to implement all of that?

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

How to implement all of that?

- CSPRNG in Java:
  - Java.security.SecureRandom

```
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")));

}
```

How to implement all of that?

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

How to implement all of that?

- 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

How to implement all of that?

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

```
import static com.kosprov.jargon2.api.Jargon2.*;

public class Jargon2RawHashExample {
    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;
```

How to implement all of that?

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

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

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

How to implement all of that?

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

```
// 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);
```

How to implement all of that?

- Argon2
  - Argon2d:
    - data-dependent memory access
  - Argon2i:
    - data-independent memory access
  - Argon2id:
    - hybrid of Argon2d & Argon2i
- Notes from the GitHub:
  - Argon2i is preferred for password hashing

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)

Resources

- https://security.stackexchange.com/questions/211/how-to-securely-hash-passwords
- https://github.com/p-h-c/phc-winner-argon2
- https://security.stackexchange.com/questions/25585/is-my-developers-home-brew-password-security-right-or-wrong-and-why
- https://security.blogoverflow.com/2013/09/about-secure-password-hashing/
- https://stackoverflow.com/questions/8881291/why-is-char-preferred-over-string-for-passwords?rq=1
- http://www.vogella.com/tutorials/EclipseMaven/article.html

References

1) https://www.maxim.com/.image/t_share/MTQ0MjczMjg0NDc5OTE5NDg3/custom-custom_size___what-salt-bae-memejpg.jpg

2) https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQUDYA-eslIUVeG1j4FJ5EJhZu64qJwWyw-o9eguWYw8GeG4hkF