Passwort Sicherheit

Hashing

- Bestätigt die **Integrität** von Daten.

- Der gleiche Input ergibt immer den selben Output.
- Ungleiche Inputs sollen ungleiche Outputs ergeben.
- Es soll unmöglich sein aus dem Output auf den Input zu schliessen.
- Jede Änderung am Input soll in einer drastische Änderung des Outputs resultieren.

Passwort als md5-Hash speichern

Der md5 Hash von Passwort **1** lautet:
`c4ca4238a0b923820dcc509a6f75849b`

Der md5 Hash von Passwort **12** lautet:
`c20ad4d76fe97759aa27a0c99bff6710`

Der md5 Hash von Passwort **123** lautet:
`202cb962ac59075b964b07152d234b70`

Der md5 Hash von Passwort **1234** lautet:
`81dc9bdb52d04dc20036dbd8313ed055`

Der md5 Hash von Passwort **12345** lautet:
`827ccb0eea8a706c4c34a16891f84e7b`

Das Problem: *Brute-Force-Attack*

Theoretisch könnte ich jede mögliche Passwortkombination hashen und mit dem Hashwert in der Datenbank vergleichen.

Weil gleicher Input erzeugt gleicher Output!!!

Annahme:

- Passwort mit maximal 8 Zeichen
- Ziffern von a-z, A-Z, 0-9 und alle Sonderzeichen

= 94 mögliche Ziffern pro Zeichen = ergibt 94^8 mögliche Passwörter

Das Problem: Brute-Force-Attack

Eine schnelle GPU (nicht CPU) kann pro Sekunde 10^{10} Passwörter **md5**-hashen und mit einem hash-Wert in der Datenbank vergleichen.

94^8 / 10^{10} / 3600 / 24 = **ca. 7 Tage** um ein Passwort zu knacken ...

Noch schneller geht es mit einer Rainbow-Tabelle. Eine Tabelle, welche alle möglichen hash-Werte für eine bestimmte Passwortlänge und Ziffern-Auswahl enthält. Im Internet käuflich zu erwerben...

Der Aufwand für das Hashen entfällt, ich muss lediglich einen Lookup in der Rainbow-Tabelle machen > ca. 1 Minute Pro Passwort !!!

Lösungsansatz: Passwortlänge, Hash-Funktion

- Passwortlänge vergrössern:
  94^9 / 10^{10} / 3600 / 24 / 365 = **1.81 Jahre** um ein Passwort zu knacken
  94^{10} / 10^{10} / 3600 / 24 / 365 = **170.8 Jahre** um ein Passwort zu knacken

- Andere Hash-Funktion verwenden, welche mehr Rechenzeit benötigt.

Aber: Das Problem mit den Rainbow-Tabellen ist damit aber nicht gelöst.

Lösungsansatz: Salt

Wir versalzen das Passwort:

Zu jedem Passwort fügen **vor** wir dem Hashen einen individuellen zufälligen Wert hinzu welchen wir auch in der Datenbank sichern und Hashen die Kombination anschliessend.

$password = md5(‚12345‘); > Benutzereingabe

$salt = md5(‚abU56/.L?q‘); > Speichern in DB: *salt*

$password_salt = md5($password . $salt); > Speichern in DB: *password*

Lösungsansatz: Salt

Die Passwortlänge wird, ohne den Nutzer zu belasten, um 32 Stellen erhöht.

Für jeden individuellen Salt, also für jedes Passwort, müsste eine eigene Rainbow-Tabelle verwendet werden. Dieser Aufwand ist viel zu gross.

Ziel ist es also, die **Kosten für das Knacken des Passwort zu erhöhen**, nicht das Knacken zu verhindern!

Mit PHP

- **`password_hash($password, PASSWORD_DEFAULT);`**
  Erzeugt einen individuellen Salt und kombiniert diesen mit dem Passwort als bcrypt-Hash. Wird in der Datenbank als Passwort gespeichert. Maximaler Speicheraufwand: VARCHAR(255)

- **`password_verify($password, [DB-Hash]);`**
  Vergleicht das Passwort des Benutzers mit dem gespeicherten Hash aus der DB.
  Liefert **TRUE** zurück, wenn das Passwort übereinstimmt.

Links

Encoding, Encryption, Hashing:
https://danielmiessler.com/study/encoding_encryption_hashing/

Vorlesung über Passwort-Sicherheit:
https://www.youtube.com/watch?v=M7SWzGi0a50

Safe Password Hashing – PHP.net:
http://php.net/manual/en/faq.passwords.php

password_hash()
http://php.net/manual/en/function.password-hash.php

password_verify()
http://php.net/manual/en/function.password-verify.php