Sicheres Passwort Hashing mit Salts

(Quelle: https://code-bude.net/2015/03/30/grundlagen-sicheres-passwort-hashing-mit-salts)

**Von mir hinzugefügter Teil*
(...) Teil von mir weggelassen

Wer Software entwickelt und dies im Web-Umfeld tut, der hat sicherlich schon das ein oder andere Login-System geschrieben oder zumindest Berührungspunkte in diesem Bereich gehabt. Neben der Logik eines sicheren Login- bzw. User-Systems an und für sich, ist das sichere Speichern von Passwörtern einer der wichtigsten Punkte während der Implementierung.

Selbst wenn der eigentliche Code des Logins zu 100 Prozent fehlerfrei und sicher ist (wovon man in der Praxis nie ausgehen sollte), so kann es durch Sicherheitslücken in der Serversoftware immer noch zu Einbrüchen bzw. Hacks kommen. Es gibt immer eine Variable, auf die man keinen Einfluss hat und so werden tagtäglich Webseiten gehackt, kompromittiert und komplette Datenbanken mit Usernamen und Passwörtern ausgelesen.

Um die Nutzer im Falle eines solchen Hacks bestmöglich zu schützen, sollten die Passwörter *nicht nur gehasht, sondern zusätzlich** durch Salts (=Salz; eine Methodik aus der Kryptologie) geschützt werden.

(...)

Was ist Passwort Hashing?

Passwort Hashing bedeutet das Anwenden einer Hashfunktion auf ein Passwort. Eine Hashfunktion (*"hash'" ist englisch und steht für zerhacken*) ist eine sogenannte Streuwertfunktion, die es ermöglichst große Eingabemengen auf (meist kleinere) Zielmengen zu projizieren. Oftmals haben die Zielmengen auch festgelegte Länge.

Für den Secure Hash Algorithm (kurz SHA) beträgt die Länge der Zielmenge z.B. immer 160 Bit, die wiederum oftmals als 40-stellige Hexadezimalzahl notiert wird.

Hat man also das Passwort "Geheim" und wendet darauf SHA1 an, so ergibt sich folgende Ausgabe:

```
4d376b70dad934828fb73fb4aab5d0217ff88d15
```

Lautet das Passwort hingegen "SuperLangesUndSehrSehrGeheimesPasswortMitVielenZahlen648945145649848949489489 4", ergibt sich folgender Hash (unter Anwendung von SHA):

```
43911599b264c56f819d8e93c6d7e9614689eabe
```

Trotz der viel längeren Eingabe hat die Zielmenge immer noch dieselbe Länge von 40 Zeichen. Hieraus ergibt sich ein Problem von Hashfunktionen. Da die Eingabemenge größer sein kann als die Zielmenge, kann es vorkommen, dass zwei unterschiedliche Eingaben den gleichen Hashwert erzeugen. Dies nennt man eine Kollision.

Wenn statt dem Passwort der Hash in der Datenbank gespeichert werden soll, so sind Kollisionen ein nicht erwünschter Seiteneffekt. Zudem ist nicht jede Hashfunktion zur Anwendung auf Passwörter geeignet. Aus diesem Grund gibt es eine Unterart der Hashfunktionen, die sogenannten kryptologischen Hashfunktionen.

(...)

Wenn für kryptologische Hashfunktionen dennoch Kollisionen gefunden werden, dann ist entweder eine Schwachstelle im Algorithmus entdeckt worden oder der Aufwand zum Finden einer Kollision ist im Verhältnis zum technischen Fortschritt nicht mehr groß genug um eine Kollisionsresistenz zu sichern. In diesem Falle spricht man davon, dass der jeweilige Algorithmus "broken" (=zerbrochen, kaputt) ist. Von der Nutzung solcher Algorithmen sollte abgesehen werden. (Dies gilt z.B. auch für die MD5-Funktion.)

Zum Passwort-Hashing sollten also **ausschließlich** kryptologische Hashfunktionen verwendet werden.

Der Standardablauf zur Verwaltung von Nutzerdaten und Passwörten unter Verwendung von Hashfunktionen sieht wie folgt aus:

1. Der Nutzer registriert sich mit Nutzernamen und Passwort.
2. Das Passwort wird vorm Speichern in der Datenbank gehasht und nur der Hash wird in der Datenbank abgelegt.
3. Will sich der Nutzer nun einloggen, gibt er Nutzername und Passwort an. Sein Passwort wird wieder gehasht und mit dem Hash verglichen, der in der Datenbank für seinen Nutzernamen abgelegt ist.
4. Stimmen die Hashes überein, wird der Nutzer eingeloggt, anderenfalls erhält er eine *generische* Fehlermeldung. (Die Fehlermeldung sollte hierbei immer nur sagen, dass die Zugangsdaten falsch sind. Wird angegeben, ob Passwort oder Nutzername falsch sind, kann der Angreifer daraus schon schließen, dass eines der beiden auf jeden Fall richtig ist.)

(...)

Mittels der Hashes kann sich der Angreifer nun nicht einloggen, da diese in Schritt 3 ja erneut gehasht und somit nicht mehr mit dem Hash des eigentlichen Passworts übereinstimmen würden.

Warum sind Hashes nicht per se sicher?

Nachdem letzten Absatz könnte man meinen, (kryptografische) Hashes seien per se sicher. Man könnte fälschlicherweise annehmen, dass, unter Anwendung der vier Schritte aus vorherigem Absatz, die Passwörter ausreichend gesichert seien und man alles getan hätte, was zu ihrem Schutz nötig ist. Leider ist dem nicht so!

Zwar haben wir richtigerweise festgestellt, dass sich ein Hacker nicht (ohne Weiteres) mittels der Hashes einloggen kann, jedoch gibt es dennoch Möglichkeiten vom Hash wieder auf das Passwort zu schließen. Und da Nutzer (leider) meist dieselben Anmeldedaten für mehrere Portale/Webseiten nutzen, liegt die Wiederherstellung des Passworts im Interesse des Hackers. Deshalb betrachten wir nun, wie Hashes "entschlüsselt" (oder auch gecracked) werden können, um uns dann in einem späteren Absatz dann damit zu beschäftigen, wie wir eben dieses Risiko weiter eindämmen können.

Prinzipiell kann man zwischen zwei Methoden Hashes zu cracken unterscheiden:

1. **Brute Forcing**; Beim Brute Forcing (brute force = rohe Gewalt) wird versucht, durch Ausprobieren aller möglichen Kombinationen, das gesuchte Ergebnis zu erhalten. Hat man also z.B. den Hash "e22a63fb76874c99488435f26b117e37" so bildet man systematisch für alle Kombinationen eines Alphabets (z.B. A-Za-z0-9) den entsprechenden Hashwert und vergleicht diesen mit dem vorhandenen Hash. Sind die Hashes gleich, schaut man, welches die letzte Eingabekombination war und kennt somit das Passwort. Da das Durchprobieren aller Möglichkeiten einen hohen Rechen- und Zeitaufwand bedeutet, kann versucht werden zuerst mittels sogenannten "Dictionaries" zu arbeiten. Ein Dictionary (= *Wörterbuch*) ist eine Liste mit gängigen Passwörtern. Zu dieser Liste werden dann die Hashes erzeugt und mit dem gesuchten Hash verglichen. Je größer die gehackte Datenbank ist, umso wahrscheinlicher ist es, dass ein Nutzer eines der gängigen Passwörter aus dem Dictionary genutzt hat.
2. **Lookup Tables**; Um das Performance-Problem von Brute Force Attacken zu umgehen werden Lookup Tables genutzt. Eine Lookup Table ist eine spezielle Datenstruktur, die vorberechnete Passwort-Hash-Pärchen enthält. Das Besondere an Lookup Tables ist, dass sie selbst bei Datenbeständen von mehreren Millionen Pärchen immer noch mehrere hundert Abfragen pro Sekunde verarbeiten können. Lookup Tables sind also schneller als reines Brute Force, dafür sind sie durch die vorberechneten Werte ebenso wie Dictionary-Attacken auf eine vorbestimmte Eingabemenge (an Passwörtern) begrenzt. Eine Sonderform sind die sogenannten "Rainbow Tables". Sie stellen eine Mischform aus Lookup Table und Brute Force Attacke dar. So ist ein Teil der

Hashes vorberechnet, um zu bestimmen, ob sich der Hash eines Passwortes in einer Submenge befindet.

Kommen wir nun dazu, warum wir Passwörter vor dem Hashen zusätzlichen Salten (= mit einem Salt versehen) sollten.

Nutzen von Salts beim Passwort Hashing

Mittels eines Salts lässt sich das im vorherigen Abschnitt beschriebene Lookup-Table-Verfahren außer Kraft setzen. Dies funktioniert, weil Lookup Tables davon ausgehen, dass das gleiche Passwort immer den gleichen Hash ergibt. Nutzen zwei User also das gleiche Passwort haben Sie normalerweise (wenn nur gehasht wird) den gleichen Hash.

Versieht man die Passwörter der einzelnen Nutzer nun aber mit einem zufälligen Salt, so ergeben die beiden Passwörter unterschiedliche Hashes, sodass eine Lookup Table nicht funktionieren kann, da vor dem Hack die Salts nicht bekannt sind und somit auch keine Lookup Table vorberechnet werden kann.

Brute Force Attacken sind hiervor jedoch immun. Bei ausreichend langen Passwörtern ist der Aufwand des Brute Forcing jedoch so hoch, dass das Entschlüsseln aller erbeuteten Passwörter nicht in annehmbarer Zeit erfolgen kann. Zudem kann durch Key Stretching das Brute Forcen weiter erschwert werden. (Mehr zu Key Stretching im nächsten Abschnitt.)

Best Practice: Passwort Hashing mit Salt

Bei der Verwendung von Salts sollten folgende Dinge beachtet werden, um eine maximale Effektivität zu erreichen.

- Der verwendete Salt sollte einzigartig pro Nutzer und Passwort sein. Das heißt, dass nicht nur bei der Registrierung eines Nutzers, sondern auch bei dem Wechsel seines Passworts ein neuer Salt generiert werden sollte.
- Der Salt sollte aus einer zufälligen Zeichenfolge bestehen. Zur Generierung des Salts sollte ein kryptografisch sicherer Zufallszahlengenerator (...) verwendet werden. Dies ist wichtig, da nicht jeder Zufallsgenerator wirklich zufällig ist. (...)
- Weiter sollte der Salt mindestens so lang sein wie das Ergebnis der verwendeten Hash-Funktion. Wenn der Salt zu kurz ist, ergeben sich zu wenig Variationen, sodass der Angreifer trotz Verwendung eines Salts seine Lookup-Tables für alle Salt-Variationen in annehmbarer Zeit vorberechnen könnte.
- Das Hashing an sich sollte durch Erhöhung des Rechenaufwands künstlich verlangsamt werden. Mittels des Salts kann sichergestellt werden, dass Lookup Tables nutzlos werden. Gegen reine Brute Force Attacken helfen Salts jedoch nicht. Um auch Brute Forcing möglichst uneffektiv zu machen, sollte ein Hashing Algorithmus genutzt werden, der Key Stretching unterstützt. Key Stretching Algorithmen sind besonders rechenintensiv, sodass die Hashrate um einen frei definierten Faktor verlangsamt werden kann. (...) Gängige Librarys zum Key Stretching sind zum Beispiel *crypt(5)*, *PBKDF2* oder *bcrypt*. Der Parameter beim Key Stretching sollte so gewählt werden, dass das Hashing möglichst langsam ist, schwache Geräte bzw. Webserver darunter aber nicht leiden. Wird mit zu vielen Iterationen gearbeitet, kann bei hohem Nutzeraufkommen zum Beispiel die Performance des Webservers leiden, auf dem die Hashing-Operationen ausgeführt werden.

(...)