Aufgabe 3

Teil 1: Beschreibung von *MD5* mit Diskussion der Sicherheit[1]

Bei *MD5* (*Message-Digest Algorithm 5*) handelt es sich um eine kryptographische Hashfunktion. Aus einem beliebigen Datensatz kann über diese Einwegfunktion ein so genannter Hash-Wert berechnet werden, der keine direkten Rückschlüsse auf den ursprünglichen Datensatz zulässt. Diese Hash-Werte werden in verschiedenen Anwendungen verwendet. Als Beispiele seien hier die verschlüsselte Speicherung von Passwörtern, die Überprüfung von Nachrichten- oder Dateiintegrität, sowie digitale Signaturen genannt.

Eine übliche Metapher zur Beschreibung von Hashfunktionen ist der Blick in das Telefonbuch. Die Telefonnummer, zu einem Namen unter einer Adresse, lässt sich dank sinnvoller Sortierung nach Orten und Namen herausfinden. Eine Rückwärtssuche ist jedoch von Hand sehr aufwendig (und war bis 2004 in Deutschland sogar verboten[2]).

Funktionsweise

Beim so genannten *Padding* werden dem Datensatz Daten angehängt, bis die Größe ein Vielfaches von 64 Byte beträgt. Hierbei wird garantiert eine Eins an das Ende der ursprünglichen Nachricht angehängt und die letzten acht Byte repräsentieren eine Zahl, welche der Länge des ursprünglichen Datensatzes entspricht. Die Lücke dazwischen wird mit Nullen gefüllt. Um einen Datensatz beliebiger Länge auf einen 16 Byte Hash abzubilden, wird dieser in 64 Byte Blöcke aufgeteilt. Diese Blöcke werden nacheinander in vielen Runden mit einem 16 Byte großen *state* verknüpft. Es werden also quasi immer vier Bit eines Datenblockes mit einem Bit des Zustandes verknüpft.

Die Operationen und der Anfangszustand ($01.23.45.67.98.ab.cd.ef.fe.dc.ba.98.76.54.32.10_{16}$) bei *MD5* sind immer gleich, so dass der gleiche Datensatz immer auf den gleichen Hash abbildet. Somit kann z.B. ein Passwort überprüft werden, ohne dass es im Klartext gespeichert werden muss.

Sicherheit von *MD5*

Kollisionen: Ein Hauptproblem bei Hash-Funktionen ist die Gefahr von Kollisionen, also wenn zwei Datensätze auf den selben Hash abgebildet werden. *MD5* wurde im Jahre 1992 veröffentlicht, es wurden jedoch 1996 schon erste Kollisionen berechnet[3]. Eindrucksvoll wurde am *25C3* demonstriert, wie durch Kollisionsberechnung auf einem PlayStation-3-Konsolen-Cluster das Herausgeberzertifikat von *RapidSSL* gefälscht werden konnte[4]. Durch die *PKI-Vertrauenskette* werden die von *RapidSSL* signierten Zertifikate in allen gängigen Browsern akzeptiert und mit einem gefälschten Herausgeberzertifikat liesen sich somit beliebige Internetdienste als verifiziert und somit vertrauenswürdig zertifizieren. Aus diesem Grunde wird *MD5* im professionellen Bereich nicht mehr für Zertifizierungen verwendet.

Um einiges komplizierter sind *Preimage-Angriffe*, bei denen zu einem fest vorgegebenen Datensatz bzw. gegebenem Hash ein alternativer Datensatz mit gleichem Hash gefunden werden soll. Da dies bisher noch nicht nachweislich gelungen ist und *MD5* in diesem Bereich daher noch als sicher gilt, wird *MD5* immer noch häufig z.B. zur Verifizierung von Dateidownloads im Internet verwendet.

Auch zum verschlüsselten Speichern von Passwörtern spielen **gesalzene** *MD5*-Hash-Werte nach wie vor eine Rolle. Durch die Miteinberechnung eines meist zufällig erzeugten Wertes (*salt*), sind die momentan gängigen Attacken über *Rainbow-Tables* (siehe Teil 2) nicht mehr effektiv, da auch gleiche Passwörter durch die Miteinbrechnung des zufälligen Salt einen anderen Hash-Wert haben. Somit benötigt man noch mehr vorberechnete Daten. . . Der *Salt* wird meistens zusammen mit dem Hash-Wert gespeichert und kann zur erneuten Überprüfung einer Eingabe ausgelesen und mit dieser verrechnet werden, so dass für den Entwickler nur ein minimaler Mehraufwand bei der Implementierung gesalzener Hash-Werte entsteht.

[1] Dieser Teil basiert inhaltlich hauptsächlich auf [1].
[2] Siehe § 14 Absatz 4 der ehemaligen Telekommunikations-Datenschutzverordnung
[3] Beispiel und weitere Verweise: http://mathstat.dal.ca/~selinger/md5collision/
[4] *Rapid-SSL* Hack: http://www.win.tue.nl/hashclash/rogue-ca/

Ich werde in der aktuellen Situation nach wie vor per *MD5* gehashten, gesalzenen, regelmäßig geänderten und sicheren Passwörtern vertrauen, weiche aber in anderen Bereichen (Zertifizierung, Signierung...) auf die Nachfolger aus der *SHA*-Familie aus.

Teil 2: Möglichkeiten zum Knacken

Ich beschränke mich in dieser Aufzählung nicht nur auf einen direkten Angriff auf den Hash, sondern beziehe auch andere Möglichkeiten mit ein:

1. **Brut-Force:** Ausprobieren aller möglichen Varianten. Benötigt enorme Rechenkapazität (oder sehr viel Zeit) und ist momentan nur bei kurzen Passwörtern mit homogenen Zeichen wirklich realistisch.
2. **Wörterbuchattacke:** Es wird eine Liste häufig gebrauchter Passwörter, oder ein so genanntes Wörterbuch (Wortliste generiert aus Online Resourcen wie Wikipedia, OpenThesaurus, Übersetzungsdienste, Rechtschreibprogrammen...) zum Vergleich genommen. Ein solcher Angriff ist recht einfach und die Trefferquote ist hier leider sehr hoch.
3. **Vorberechnung:**
   a) **Vollständig:** Vorberechung aller möglichen Kombinationen und anschließendes Speichern in einer Datenbank. Benötigt nach enormer Rechenkapazität (oder sehr viel Zeit) eine enorme Festplattenkapazität und ist daher momentan noch nicht realistisch
   b) **Häufige Passwörter:** Vorberechnung der Hash-Werte häufig gebrauchter Passwörter oder von Wörterbüchern. Ist leider immer wieder erfolgreich [5] [6].
4. **Rainbow Tables** (ebenfalls nach [1]): Datenbanken mit vorberechneten Berechnungszyklen. Bei der Erstellung wird über so genannte Reduktionsfunktionen aus dem per *MD5* erstellten Hash wieder ein (anderer) Klartext erzeugt, welcher wieder mit *MD5* gehasht und anschließend wieder reduziert wird.... In der Datenbank wird anschließend nur der erste Klartext und der Hash am Ende der Kette gespeichert. Hierdurch reduziert sich der Speicherverbrauch einer *Rainbow Table* mit n Zyklen auf 1/n. Der zu einem Hash zugehörigen Klartext wird gefunden, in dem das eben beschriebene Verfahren mindestens n-1 Mal angewand wird. Einer der n Hash-Werte (das Original mit eingeschlossen) sollte in der Tabelle zu finden sein. Der gesuchte Klartext befindet sich in der von dem gefundenen Eintrag erzeugten Kette. Also muss diese Kette einfach neu erzeugt werden, womit der zu dem gesuchten Hash zugehörige Klartext gefunden wird. Dies ist in Abbildung 1 noch einmal graphisch dargestellt. Ähnlich dem *SETI@HOME*-Projekt gibt es Rechnerverbünde von Privatpersonen, die diese *Rainbow-Tables* berechnen und deren Dienste frei im Internet zur Verfügung stellen. Es scheint sich sogar schon jemand die Mühe gemacht zu haben, den Klartext zu dem im Übungsblatt angegebenen Hash zu berechnen. Der *MD5*-Hash *bb5459bf19132d4dc1340654c17331df* gehört zu dem Passwort *sonic*.
5. **Hacking:** Man schmuggelt der Zielperson ein Programm unter, welches diese dazu bringt das Passwort anzugeben oder versucht dieses in einer (unverschlüsselten) Verbindung abzufangen, was so Ähnlich letzte Woche an der Universität Ulm demonstriert wurde[7].
6. **Social Engineering:** Eine -abhängig von der Zielgruppe- immer noch effektive Variante um an Passwörter zu gelangen [2]. Indem man sich im *real life*, per Telefon oder digital als autorisierte Person (Polizeibehörde, IT-Betreuung, Kundenbetreuer...) ausgibt, erfragt man Passwörter oder bekommt Hinweise auf mögliche Passwörter durch Erforschung des sozialen Umfeldes.

[5] Tool zur „Überprüfung“ der Passwortsicherheit unter Windows mit Raindbow-Table üblicher Passwörter: http://ophcrack.sourceforge.net/
[6] Tool zur „Überprüfung“ der Passwortsicherheit mit Liste üblicher Passwörter unter Linux und Mac: http://www.openwall.com/john/
[7] Catching AuthTokens in the Wild: http://www.uni-ulm.de/en/in/mi/staff/koenings/catching-authtokens.html

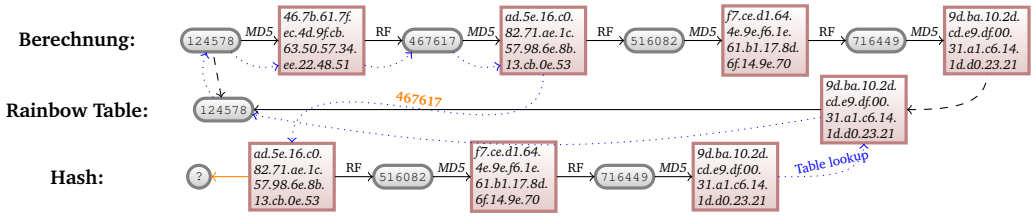

Abbildung 1: Eine *Rainbow Table* wird erzeugt, in dem erst Zyklen mit Hilfe von Reduktionsfunktionen (RF, hier: nehme die ersten 6 Ziffern des Hash-Wertes) erzeugt, aber nur der Ausgangswert und der letzte Hash gespeichert werden. Sucht man ein zu einem Hash zugehörigen Klartext, so wendet man die Reduktionsfunktion auf den Hash an und schaut, ob einer der neu erzeugten Hash-Werte in der *Rainbow Table* zu finden ist. Ist dies der Fall, so muss die Kette anhand des ebenfalls gespeicherten Ausgangswertes nur noch neu erzeugt werden, um den gesuchten Klartext zu finden. Die Qualität der *Rainbow Table* hängt also stark von der Reduktionsfunktion ab.

Teil 3: Brute-Force-Attacke

Listing 1: Brute-Force Attacke auf *MD5*-Passwort-Hash mit *Python*

```
#!/usr/bin/env python3
# -*- py-which-shell: "python3"; -*-

import string
import itertools
import hashlib

# hash to test
test = "bb5459bf19132d4dc1340654c17331df"
# max length to test
length = 5

found = False
for t in range(1,length+1):
    for n in itertools.product(string.ascii_lowercase, repeat=t):
        if hashlib.md5(b''.join(str(i) for i in n)).hexdigest() == test:
            found = True
            print(''.join(str(i) for i in n))
            break

if not found:
    print("Password not found!")
```

Literatur

[1] KUHN, N.: *Das Buch der geheimen Verschlüsselungstechniken*. Data-Becker, 2009. – ISBN 9783815829615 1, 2

[2] MITNICK, K.D. ; SIMON, W. ; DUBAU, J.: *Die Kunst der Täuschung: Risikofaktor Mensch*. mitp-Verlag, 2006. – ISBN 9783826615696 2