Kryptologie Labor

# Kompensationsarbeit

Simone Fink

28. Juni 2018

# Inhaltsverzeichnis

# 1

## Vorbereitung

**Ausgangssituation**: Sie haben ein HTML-Dokument mit folgendem Inhalt vorliegen:

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title>Signierter Vertrag</title>
	</head>
	<body>
		Ich, Alice, überweise Bob den Betrag von 100 Euro.
	</body>
</html>
```

Dieses Dokument wurde von Alice digital signiert.

# 2

# Kompensationsaufgaben

## 1. LB-KA 00

### a)

Erstellen Sie eine HTML-Datei nach obigem Muster.

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>Signierter Vertrag</title>
        </head>
        <body>
                Ich, Alice, überweise Bob den Betrag von 100 Euro.
        </body>
</html>
```

Abbildung 2.1: die erstelle HTML-Datei mit dem Namen "ueberweisung.html"

### b)

Berechnen Sie den SHA-256 Hash dieser Datei und verwenden Sie Ihr Programm aus LB-HVS 01 zum Signieren des Hashes. Prüfen Sie anschließend die Integrität der Datei, indem Sie die Signatur verifizieren.

Der SHA-256 Hash dieser Datei ist
5064b36096864eb0ab6df830b8b35e7ada2d20a46c95d423bfb191aa10db8ebb

Abbildung 2.2: Signierung und Verizifierung von der HTML- Datei

Wie in Abbildung 2.2 zu sehen ist, ist die Signatur nun gültig. Signierung und Verifizierung wurden über die Linux Bash Shell ausgeführt. Die Programme zur Signierung und Verifizierung wurden mit Hilfe von Clemens J. Zuzan erstellt.

### c)

Verändern Sie den Inhalt der Datei dahingehend, dass Alice den Betrag von 200 Euro an Bob überweist. Prüfen Sie nun erneut die Integrität der Datei. Ist die Signatur noch gültig?

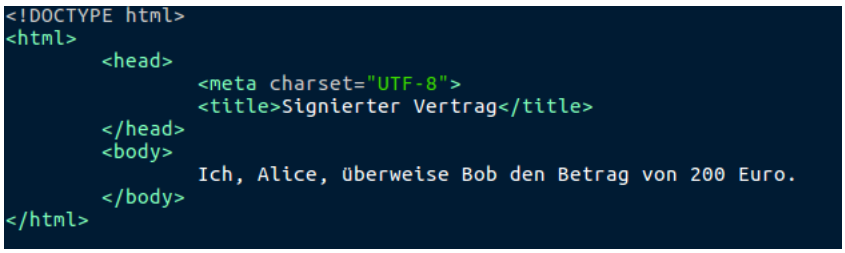

Abbildung 2.3: die veränderte HTML-Datei mit dem Namen "ueberweisung.html"

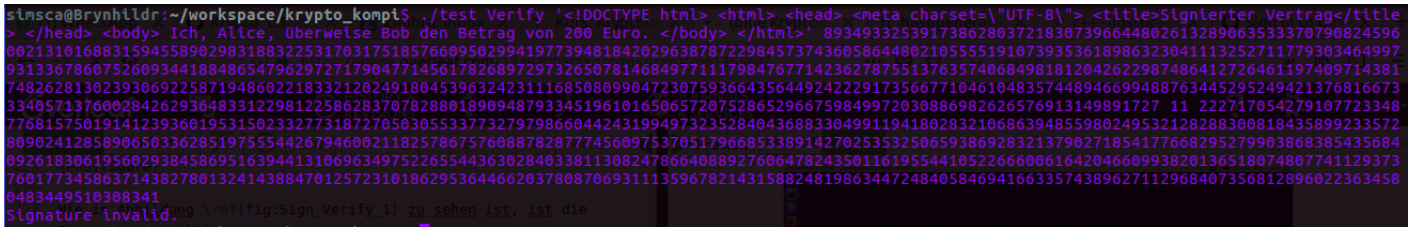

Abbildung 2.4: Verifizierung des Hashes der Originaldatei mit der gefälschten Datei

Wie in Abbildung 2.4 sichtbar ist, ist die Signatur nicht mehr gültig. Die Verifizierung wurde wieder über die Linux Bash Shell ausgeführt und das Programm mit Hilfe von Clemens J. Zuzan geschrieben.

# 2. LB-KA 01

Im Originaldokument überweist Alice an Bob den Betrag von 100 Euro. Es soll ein "gefälschtes"Dokument erzeugt werden, in welchem Alice den Betrag von 200 Euro an Bob überweist. Dieses falsche Dokument soll jedoch die exakt gleiche Signatur wie das Originaldokument aufweisen (*preimage attack*). Versuchen Sie durch kleine, aber für den Betrachter nicht sichtbare Änderungen im Markup eine Kollision der Hashwerte und somit eine idente Signatur zu erreichen. Schreiben Sie dafür ein geeignetes Programm zum Modifizieren des Markups und zum Berechnen des Hash-Wertes. Das Programm soll abschließend eine Variante des Dokuments mit identem Hash ausgeben.
**Wichtig**: SHA-256 hat als Ausgabe $2^{256}$ mögliche verschiedene Werte. Damit sind im Mittel $2^{255}$ Versuche notwendig, um eine Kollision zu erzeugen. Verwenden Sie für die Aufgabe daher eine stark vereinfachte (und in der Praxis unsichere) Variante, bei der nur die ersten vier Hexadezimalziffern des Hash- Wertes verwendet werden. Dies reduziert die im Mittel notwendigen Versuche.

## Händisches Ausprobieren

Wir haben: die ersten 4 Hexadezimalziffern des Hash- Wertes von der ersten originalen Datei: 5064
Wir suchen: die Datei, in der steht es werden 200 Euro überwiesen, mit den selben 4 Hexadezimalziffern des Hash- Wertes.
Nachdem der Datei **2203 Leerzeichen** am Ende eingefügt wurden, entsprechen die ersten 4 Hexadezimalziffern 5064.

## Programmieren

Es wird ein C++ Programm geschrieben, bei dem der gefälschten Datei hinten dran immer wieder Leerzeichen angehängt werden bis die ersten 4 Hexadezimalziffern des Hashes den ersten 4 des Originaldokuments entsprechen.

```
1 /*
2  * LB_KA_01.cpp
3  *
4  *  Created on: Jun 27, 2018
```

```
5   *       Author: Simone Fink (Hilfe von Clemens J. Zuzan)
6   */
7
8
9   #include <string.h>
10  #include <iomanip>
11  #include <sodium.h>
12  #include <sstream>
13
14
15  #include <iostream>
16  #include <gmp.h>
17  #include <gmpxx.h>
18  #include <fstream>
19  using namespace std;
20
21  //Erstellen eines Programms welches ein File engegennimmt, dieses
        veraendert
22  //und dann solange Leerzeichen hintendran haengt, bis die ersten 4
        Ziffern des Hashes
23  //den ersten 4 Ziffern des Originalhashes entsprechen
24
25
26  //Mit Hilfe von Clemens J. Zuzan
27  void libsodium_to_GMP(const unsigned char (&libsodium_value) [
        crypto_hash_sha256_BYTES], mpz_class &GMP_value)
28  {
29          stringstream s;
30          s << hex;
31          for (size_t i = 0; i < sizeof libsodium_value; i++)
32                  s << setw(2) << setfill('0') << (int)libsodium_value
                        [i];
33          const char * const string_as_hex = s.str().c_str();
34          mpz_set_str(GMP_value.get_mpz_t(), string_as_hex , 16);
35  }
36
37  //Mit Hilfe von Clemens J. Zuzan
38  string hashwert(const string &message){
39          unsigned char hash[crypto_hash_sha256_BYTES];
40
41          const size_t message_length = message.length();
42          crypto_hash_sha256(hash, (const unsigned char*)message.c_str
                ), message_length);
```

```
43
44          mpz_class hash_mpz;
45
46          libsodium_to_GMP(hash,hash_mpz);
47          return hash_mpz.get_str(4); //es werden nur die ersten 4
                Ziffern benoetigt
48  }
49
50  int faelschen(string origs, string fakes)
51  {
52          string message_orig;
53          string message_fake;
54          char c;
55
56          ifstream orig;
57          orig.open(origs.c_str());
58          if(!orig)
59          {
60                  cerr << "Fehler beim oeffnen der Originaldatei!" <<
                        endl;
61                  return -1;
62          }
63
64          ifstream fake;
65          fake.open(fakes.c_str());
66          if(!fake)
67          {
68                  cerr << "Fehler beim oeffnen der gefaelschten Datei
                        !" << endl;
69          }
70          while(orig >> c)
71          {
72                  message_orig += c;
73          }
74
75          while(fake >> c)
76          {
77                  message_fake += c;
78          }
79
80          //hashes der beiden dateien werden nun ausgerechnet
81
82          string hash_orig = hashwert(message_orig);
```

```
83          string hash_fake = hashwert(message_fake);
84
85          //das Inputfile wird nun nicht mehr benoetigt
86          orig.close();
87
88          //nun werden die hashes miteinander verglichen. wenn sie
               nicht gleich sind
89          //wird ein Leerzeichen ans outputfile angehaengt.
90
91          ofstream outputfile("Orig_Fake.html");
92          outputfile << message_fake;
93          char leer = ' ';
94          int leercnt;
95          while(hash_orig.compare(hash_fake) != 0)
96          {
97                  outputfile << leer;
98                  message_fake += leer;
99                  leercnt++;
100                 hash_fake = hashwert((const string)message_fake);
101         }
102         fake.close();
103
104         return leercnt;
105 }
106
107
108 int main(int argc, char** argv)
109 {
110         string orig(argv[1]);
111         string fake(argv[2]);
112         int leercnt = faelschen(orig, fake);
113         char c;
114         ifstream outputfile("Orig_Fake.html");
115         while(outputfile >> c)
116         {
117                 cout << c;
118         }
119         outputfile.close();
120         cout << "\n\n leerzeichen die hinzugefuegt wurden: " <<
               leercnt << endl;
121 }
122
123 }
```

Dieses Programm braucht sehr lange bis ein Ergebnis herauskommt. Ein Beispiel, wie man das Programm beschleunigen könnte, wäre die Realisierung von Threads.

## 3. LB-KA 02

Versuchen Sie nun zwei Dokumente zu erzeugen, ein "Originaldokument"(Alice an Bob 100 Euro) und ein "gefälschtes Dokument"(Alice an Bob 200 Euro), die beide den gleichen Hashwert haben (*second preimage attack* ). Gehen Sie dabei analog zur Aufgabe LB-KA 02 vor und erzeugen Sie durch kleine, aber für den Betrachter nicht sichtbare Änderungen im Markup beider Dokumente die Kollision. Überlegen Sie zunächst, worin der Unterschied zum obigen Angriff liegt und wie sich dieser auf die Anzahl der im Mittel notwendigen Versuche auswirkt. Schreiben Sie dafür wiederum ein geeignetes Programm zum Modifizieren des Markups beider Dokumente und zum Berechnen der Hash-Werte. Das Programm soll abschließend zwei Varianten der Dokumente mit identem Hash ausgeben.

**Überlegung:** Die Originaldatei wird so lange verändert, bis eine Hashkollision entsteht. Danach wird die gefälschte Datei so lange verändert, bis eine Hashkollision zum kollidierten Hash der Originaldatei ensteht.

```
1  /*
2   * LB_KA_02.cpp
3   *
4   *  Created on: Jun 27, 2018
5   *      Author: Simone Fink (Hilfe von Clemens J. Zuzan)
6   */
7
8
9  #include <string.h>
10 #include <iomanip>
11 #include <sodium.h>
12 #include <sstream>
13
14
15 #include <iostream>
16 #include <gmp.h>
17 #include <gmpxx.h>
18 #include <fstream>
19 using namespace std;
20
21 //Erstellen eines Programms welches zwei Files entgegennimmt
22 //und dann bei beiden solange Leerzeichen hintendran haengt, bis die
       ersten 4 Ziffern des Hashes
23 //den ersten 4 Ziffern des Originalhashes entsprechen
```

```
24
25
26 //Mit Hilfe von Clemens J. Zuzan
27 void libsodium_to_GMP(const unsigned char (&libsodium_value) [
       crypto_hash_sha256_BYTES], mpz_class &GMP_value)
28 {
29         stringstream s;
30         s << hex;
31         for (size_t i = 0; i < sizeof libsodium_value; i++)
32                 s << setw(2) << setfill('0') << (int)libsodium_value
                       [i];
33         const char * const string_as_hex = s.str().c_str();
34         mpz_set_str(GMP_value.get_mpz_t(), string_as_hex , 16);
35 }
36
37 //Mit Hilfe von Clemens J. Zuzan
38 string hashwert(const string &message){
39         unsigned char hash[crypto_hash_sha256_BYTES];
40
41         const size_t message_length = message.length();
42         crypto_hash_sha256(hash, (const unsigned char*)message.c_str
              (), message_length);
43
44         mpz_class hash_mpz;
45
46         libsodium_to_GMP(hash,hash_mpz);
47         return hash_mpz.get_str(4); //es werden nur die ersten 4
              Ziffern benoetigt
48 }
49
50 int faelschen(string origs, string fakes)
51 {
52         string message_orig;
53         string message_fake;
54         char c;
55
56         ifstream orig;
57         orig.open(origs.c_str());
58         if(!orig)
59         {
60                 cerr << "Fehler beim oeffnen der Originaldatei!" <<
                       endl;
61                 return -1;
```

```
62          }
63
64          ifstream fake;
65          fake.open(fakes.c_str());
66          if(!fake)
67          {
68                  cerr << "Fehler beim oeffnen der gefaelschten Datei
                        !" << endl;
69          }
70          while(orig >> c)
71          {
72                  message_orig += c;
73          }
74
75          while(fake >> c)
76          {
77                  message_fake += c;
78          }
79
80          //hashes der beiden dateien werden nun ausgerechnet
81
82          string hash_orig = hashwert(message_orig);
83          string hash_fake = hashwert(message_fake);
84
85          //nun werden die hashes miteinander verglichen. wenn sie
               nicht gleich sind
86          //wird ein Leerzeichen ans outputfile angehaengt.
87
88          char leer = ' ';
89          int leercnt1 = 0;
90          string hash_orig_neu = hashwert(message_orig);
91          do{
92                  orig << leer;
93                  message_orig += leer;
94                  leercnt1++;
95                  hash_orig_neu = hashwert(message_orig);
96          }
97          while(hash_orig.compare(hash_orig_neu) != 0);
98
99          int leercnt = leercnt1;
100
101         int leercnt2 = 0;
102         while(hash_orig.compare(hash_fake) != 0)
```

```
103         {
104                 fake << leer;
105                 message_fake += leer;
106                 leercnt2++;
107                 hash_fake = hashwert(message_fake);
108         }
109
110         leercnt += leercnt2;
111
112         fake.close();
113         orig.close();
114
115         return leercnt;
116
117
118 }
119
120
121 int main(int argc, char** argv)
122 {
123         string orig(argv[1]);
124         string fake(argv[2]);
125         int leercnt = faelschen(orig, fake);
126         cout << "\n\n leerzeichen die insgesamt in beiden Files
               hinzugefuegt wurden: " << leercnt << endl;
127 }
```

Hierbei stößt man auf das Geburtstagsparadoxon. Es ist um einiges schwerer eine Hashkollision von der Originaldatei zu erlangen, als bei der gefälschten Datei. Deswegen wird die Ausführung dieses Programmes noch länger dauern als bei LB_KA_01.