Institut für Theoretische Informatik
Prof. Dr. Dennis Hofheinz
Übungsleiter: Thomas Agrikola

Stammvorlesung Sicherheit im Sommersemester 2017

Übungsblatt 5

Aufgabe 1.

(1.) In der Vorlesung wurde gezeigt, dass Lehrbuch-RSA-Signaturen nicht EUF-CMA-sicher sind, da ein Angreifer $\mathcal{A}$ beliebige Signaturen zu unsinnigen Nachrichten fälschen kann. Wir ändern nun das EUF-CMA Sicherheitsexperiment leicht ab, indem wir fordern, dass der Angreifer $\mathcal{A}$ zu einer bestimmten vom Challenger vorgegebenen Nachricht m^* eine Signatur σ^* fälschen muss, um das Spiel zu gewinnen. Er hat weiterhin ein Signaturorakel zur Verfügung, jedoch darf die Nachricht m^* natürlich nicht angefragt werden. Erfüllt das Lehrbuch-RSA-Signaturverfahren diesen neuen Sicherheitsbegriff? Beweisen Sie dies oder geben Sie einen erfolgreichen Angriff an.

(2.) Wir betrachten den Digital-Signature-Algorithmus (DSA) über der Gruppe $\mathbb{G} = Q(\mathbb{Z}_p^\times)$, für ungerades primes $p \in \mathbb{N}$. Dabei sei $Q(\mathbb{Z}_p^\times) := \{x^2 : x \in \mathbb{Z}_p^\times\}$ die Menge der Quadrate in $\mathbb{Z}_p^\times$.

(a) Erstellen Sie einen (DSA-)Public-Key $pk := (\mathbb{G}, g, g^x, (\mathsf{H}, h_1, h_2))$ sowie einen (DSA-)Secret-Key $sk := (\mathbb{G}, g, x, (\mathsf{H}, h_1, h_2))$. Dabei seien $p := 2q + 1$ (eine strong prime für q prim), $q = 11$, und eine Hashfunktion $\mathsf{H} : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{Z}_q^\times,\ (x_1, x_2) \mapsto h_1^{x_1} h_2^{x_2} \bmod q$ gegeben. (Wir setzen $g := 8, h_1 := 4, h_2 := 2$.)

(b) Signieren Sie die Nachricht $M = (7, 3)$ mithilfe des Secret-Keys aus (a).

(c) Verifizieren Sie die Signatur zur Nachricht M aus (b) mittels des Public-Keys aus (a).

Aufgabe 2. Führen Sie einen Diffie-Hellman-Schlüsselaustausch über der Gruppe $\mathbb{G} = Q(\mathbb{Z}_p^\times)$ ($Q(\mathbb{Z}_p^\times)$ bezeichne wie in Aufgabe 1 die Untergruppe der Quadrate), für ungerades primes $p \in \mathbb{N}$, durch. Wir setzen $p := 23$. Folgende Schritten werden ausgeführt:

(i) A und B einigen sich auf einen Generator g der Gruppe $\mathbb{G}$, sodass $\mathbb{G} = \langle g \rangle$ gilt. Für diese Aufgabe sei $g := 18$. Die Ordnung von $\mathbb{G}$ ist 11.

(ii) A wählt zufällig gleichverteilt ein $x \in \{0, \ldots, |\mathbb{G}| - 1\}$ und schickt $X := g^x \bmod p$ an B. A wählt hier $x := 6$.

(iii) B wählt zufällig gleichverteilt ein $y \in \{0, \ldots, |\mathbb{G}| - 1\}$ und schickt $Y := g^y \bmod p$ an A. B wählt hier $y := 8$.

(iv) A berechnet $K := Y^x = (g^y)^x = g^y \bmod p$ und B berechnet $K := X^y = (g^x)^y = g^{xy} \bmod p$. Somit haben sich beide auf K als gemeinsamen Schlüssel geeinigt.

Berechnen Sie X, Y und K.

Aufgabe 3. Sei ein 2-Parteien-2-Nachrichten-Schlüsselaustauschverfahren $\mathsf{KE} = (\mathsf{KE.Gen}, \mathsf{KE.Encap}, \mathsf{KE.Decap})$ gegeben, wobei die KE-Algorithmen wie folgt beschrieben sind:

- Die Parametergenerierung $\mathsf{KE.Gen}(1^k)$ erhält als Eingabe den Sicherheitsparameter $k \in \mathbb{N}$ und gibt einen State s und eine Nachricht X aus. (Die Nachricht X stellt dabei die erste Nachricht im Schlüsselaustauschverfahren dar.)

- Die Schlüssel-Encapsulation $\mathsf{KE.Encap}(X)$ erhält als Eingabe eine Nachricht X, gibt eine Nachricht Y und einen Schlüssel K aus. (Die Nachricht Y entspricht dabei der zweiten ausgetauschten Nachricht im Schlüsselaustauschverfahren.)
- Die Schlüssel-Decapsulation $\mathsf{KE.Decap}(s, Y)$ erhält als Eingabe zusammen mit einem State s die Nachricht Y und gibt einen Schlüssel K' aus.

Wir fordern Korrektheit: Für alle $k \in \mathbb{N}$, für alle $(s, X) \leftarrow \mathsf{KE.Gen}(1^k)$, für $(K, Y) \leftarrow \mathsf{KE.Encap}(X)$ gilt $\mathsf{KE.Decap}(s, Y) = K$.

Ein Benutzer A berechnet demnach $(s, X) \leftarrow \mathsf{KE.Gen}(1^k)$ und sendet X an einen Benutzer B. B wiederum berechnet $(Y, K) \leftarrow \mathsf{KE.Encap}(X)$ und sendet Y an A. Schließlich erhält A den Schlüssel $K \leftarrow \mathsf{KE.Decap}(s, Y)$.

Konstruieren Sie ein Public-Key-Verschlüsselungssystem $\mathsf{PKE} = (\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ aus KE.

Hinweis: Denken Sie an den Diffie-Hellman Schlüsselaustausch und seiner Beziehung zum ElGamal-Verschlüsselungsverfahren.

Aufgabe 4. Der ebenso geniale wie modebewusste Wissenschaftler und Superbösewicht Doktor Meta ist betrübt. Seinen Recherchen zufolge sind Säurebecken, Todesstrahlen und Raubtierkäfige nicht mehr zeitgemäß. Er möchte sein geheimes Untergrundlabor deshalb neu einrichten. Nachdem die Einführung seiner Kryptowährung MetaCoin nicht den gewünschten Erfolg hatte, reichen seine Finanzen dafür aber nicht aus. Im Internet hat er von einem Angriff namens „CRIME“ auf das TLS-Protokoll gelesen. Einerseits entspricht der Name genau seinem Geschmack, andererseits erhofft sich Doktor Meta mit Hilfe dieses Angriffs, die Sessioncookies der Kunden von *Villains'R'Us*[1] auszuspionieren. Dadurch könnte er sowohl sein Labor neu einrichten, als auch gleichzeitig konkurrierenden Superbösewichten schaden.

(a) Wir wollen das „Nachricht komprimieren, dann verschlüsseln“-Prinzip (welches den CRIME-Angriff ermöglicht) zuerst aus theoretischer Perspektive untersuchen. Sei dazu $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ ein IND-CPA-sicheres Public-Key-Verschlüsselungsverfahren mit dem Nachrichtenraum $\mathcal{M}$. Zu PKE ist ein PPT-Algorithmus $\mathcal{L}$ bekannt, für den die Wahrscheinlichkeit

$$\Pr\left[\mathcal{L}(pk, C) = |M| \mid (pk, sk) \leftarrow \mathsf{Gen}(1^k), C \leftarrow \mathsf{Enc}(pk, M)\right]$$

für alle Nachrichten $M \in \mathcal{M}$ gleich 1 ist, d.h. $\mathcal{L}$ kann aus einem Chiffrat die Länge des verschlüsselten Klartextes berechnen.

Zusätzlich sei $\mathsf{C} = (\mathsf{Comp}, \mathsf{Decomp})$ ein Kompressionsalgorithmus (Comp komprimiert, Decomp dekomprimiert). Gehen Sie für die Bearbeitung davon aus, dass Nachrichten gleicher Länge nicht zwingend auf die gleiche Länge komprimiert werden, sondern diese Länge umso kürzer ist, je redundanter die Nachricht war (beispielsweise würde die Nachricht $aaaaaaaaaa$ auf einen kürzeren Text komprimiert als $abcdefghij$).

Es sei nun $\mathsf{PKE}' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ gegeben durch:

- $\mathsf{Gen}'(1^k) = \mathsf{Gen}(1^k)$,
- $\mathsf{Enc}'(pk, M) = \mathsf{Enc}(pk, \mathsf{Comp}(M))$,
- $\mathsf{Dec}'(sk, C) = \mathsf{Decomp}(\mathsf{Dec}(sk, C))$.

Zeigen Sie: PKE' ist nicht IND-CPA-sicher.

Überlegen Sie, wie die Aufgabe zu lösen ist, wenn die Wahrscheinlichkeit, dass $\mathcal{L}$ die richtige Länge berechnet, nicht gleich 1, sondern nur *überwältigend* ist, d.h.

$$\Pr\left[\mathcal{L}(pk, C) = |M| \mid (pk, sk) \leftarrow \mathsf{Gen}(1^k), C \leftarrow \mathsf{Enc}(pk, M)\right] = 1 - f(k),$$

wobei $f(k)$ eine in k vernachlässigbare Funktion ist.

(b) Schreiben Sie ein Programm, beispielsweise ein Pythonskript, das den CRIME-Angriff bei eingeschalteter Komprimierung in TLS simuliert. (Nutzen Sie dazu auch die entsprechenden Folien aus der Vorlesung.) Vereinfacht können wir annehmen, dass ein POST-HTTP-Header wie folgt aussieht:

[1] Der führende Online-Händler für den Bedarf eines Superbösewichts – Superlaser, Haifischbecken, Reagenzgläser und viele weitere Artikel sind ständig auf Lager.

```
POST / HTTP/1.1
Host: host.edu
Cookie: <cookie data>
```

Das Ziel ist es das Cookie, also `<cookie data>`, aus den komprimierten und verschlüsselten Daten zu extrahieren. Die Komprimierung kann über Kompressionsbibliotheken, wie zlib in Python, realisiert werden. Das folgende unvollständige Pythonskript kann verwendet werden.

```
# This python script is based on a script by Tibor Jager.

import string
import zlib
import random

# random cookie data with length 8 to 20 consisting of letters, digits, and punctuation
cookie_data = ''.join(random.choice(string.letters + string.digits + string.punctuation)
  for x in range(random.randint(8,20)))

# POST HTTP header
HEADER = ("POST / HTTP/1.1\r\n"
          "Host: host.edu\r\n"
          "Cookie: " + cookie_data + "\r\n")

# use Python's compression funcion zlib
def comp(string):
    return zlib.compress(string)

# to make it easier, the chosen plaintext oracle only outputs
#  the length of a compressed message of the form HEADER + M
def chosen_plaintext_oracle(M):
    # concatenate HEADER and M
    ...
    # use compression comp
    c = ...
    # return length of a compressed message
    return len(c)

# we try to extract the cookie data here
cookie_extr = ""
cnt = 0
while len(cookie_extr) < len(cookie_data):
    minimum = 2**80
    candidate = ""
    # try different candidates and compare chosen_plaintext_oracle outputs
    for x in (string.letters + string.digits + string.punctuation):
        cnt+=1
        # query oracle with chosen plain texts
        ctxt_len = chosen_plaintext_oracle("Cookie: " + cookie_extr + ...)
        if ctxt_len < minimum:
           minimum = ctxt_len
           candidate = x
    cookie_extr += candidate

# print out result
...
```