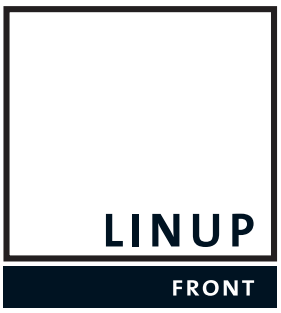
Sichere Web-Server mit Apache und SSL



Linup Front GmbH · Postfach 10 01 21 · 64201 Darmstadt · Deutschland
Telefon +49(0) 6151 9067 0 · Telefax +49(0) 6151 9067 299 · www.linupfront.de

Sichere Web-Server mit Apache und SSL
Revision: apw2:5c3a3b023f3b0957:2013-02-27
apw2:51ed1a9f97d5c472:2013-02-27 1–5, A
apw2:HQLmGlISujOK4M6g8LeWaF

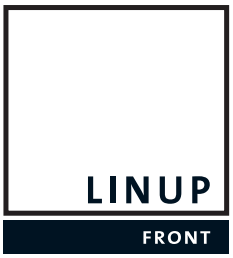
http://www.linupfront.de · info@linupfront.de

Autor: Anselm Lingnau
Technische Redaktion: Anselm Lingnau ⟨*anselm.lingnau@linupfront.de*⟩
Gesetzt in Palatino, Optima und DejaVu Sans Mono

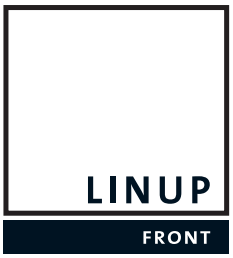
Registrieren Sie dieses Dokument on-line über den QR-Code links oder `http://shop.linupfront.de/register/HQLmGlISujOK4M6g8LeWaF/` für Aktualisierungen und interessante Angebote.

LINUP FRONT

Inhalt



Tabellenverzeichnis



Abbildungsverzeichnis

Vorwort

Der Kurs *Apache und SSL* ist für Administratoren von Web-Servern gedacht, die bereits Kenntnisse in der Konfiguration von Apache haben und »sichere« Web-Präsenzen betreiben wollen. Er gibt eine Einführung in die Grundlagen der Kryptographie, erklärt Intention, Ansatz und Struktur von OpenSSL und beschreibt die zur Erzeugung von Schlüsseln und X.509-Zertifikaten notwendigen Schritte. Anschließend wird erläutert, wie ein Apache-Server mit OpenSSL dafür konfiguriert werden kann, Web-Ressourcen über verschlüsselte und authentisierte Verbindungen anzubieten. Ferner erklärt der Kurs die Authentisierung von Benutzern über SSL-Client-Zertifikate.

Diese Schulungsunterlage soll den Kurs möglichst effektiv unterstützen, indem das Kursmaterial in geschlossener, ausführlicher Form zum Mitlesen, Nach- oder Vorarbeiten präsentiert wird. Das Material ist in Kapitel eingeteilt, die jeweils für sich genommen einen Teilaspekt umfassend beschreiben; am Anfang jedes Kapitels sind dessen Lernziele und Voraussetzungen kurz zusammengefasst, am Ende finden sich eine Zusammenfassung und (wo sinnvoll) Angaben zu weiterführender Literatur oder WWW-Seiten mit mehr Informationen.

Kapitel

Lernziele
Voraussetzungen

Zusätzliches Material oder weitere Hintergrundinformationen sind durch das »Glühbirnen«-Sinnbild am Absatzanfang gekennzeichnet. Zuweilen benutzen diese Absätze Aspekte, die eigentlich erst später in der Schulungsunterlage erklärt werden, und bringen das eigentlich gerade Vorgestellte so in einen breiteren Kontext; solche »Glühbirnen«-Absätze sind möglicherweise erst beim zweiten Durcharbeiten der Schulungsunterlage auf dem Wege der Kursnachbereitung voll verständlich.

Absätze mit dem »Warnschild« weisen auf mögliche Probleme oder »gefährliche Stellen« hin, bei denen besondere Vorsicht angebracht ist. Achten Sie auf die scharfen Kurven!

Die meisten Kapitel enthalten auch Übungsaufgaben, die mit dem »Bleistift«-Sinnbild am Absatzanfang gekennzeichnet sind. Die Aufgaben sind numeriert und Musterlösungen für die wichtigsten befinden sich hinten in dieser Schulungsunterlage. Bei jeder Aufgabe ist in eckigen Klammern der Schwierigkeitsgrad angegeben. Aufgaben, die mit einem Ausrufungszeichen (»!«) gekennzeichnet sind, sind besonders empfehlenswert.

Übungsaufgaben

Auszüge aus Konfigurationsdateien, Kommandobeispiele und Beispiele für die Ausgabe des Rechners erscheinen in `Schreibmaschinenschrift`. Bei mehrzeiligen Dialogen zwischen Benutzer und Rechner werden die Benutzereingaben in **`fetter Schreibmaschinenschrift`** angegeben, um Missverständnisse zu vermeiden. Wenn Teile einer Kommandoausgabe ausgelassen wurden, wird das durch »◁◁◁◁◁« kenntlich gemacht. Manchmal sind aus typografischen Gründen Zeilenumbrüche erforderlich, die in der Vorlage auf dem Rechner nicht stehen; diese werden als »▷ ◁« dargestellt. Bei Syntaxdarstellungen stehen Wörter in spitzen Klammern (»⟨*Wort*⟩«) für »Variable«, die von Fall zu Fall anders eingesetzt werden können; Material in eckigen Klammern (»[`-f` ⟨*Datei*⟩]«) kann entfallen und ein vertikaler Balken trennt Alternativen (»`-a`|`-b`«).

Wichtige Konzepte
Definitionen

Wichtige Konzepte werden durch »Randnotizen« hervorgehoben; die **Definitionen** wesentlicher Begriffe sind im Text fett gedruckt und erscheinen ebenfalls am Rand.

Verweise auf Literatur und interessante Web-Seiten erscheinen im Text in der Form »[GPL91]« und werden am Ende jedes Kapitels ausführlich angegeben.

Wir sind bemüht, diese Schulungsunterlage möglichst aktuell, vollständig und fehlerfrei zu gestalten. Trotzdem kann es passieren, dass sich Probleme oder Ungenauigkeiten einschleichen. Wenn Sie etwas bemerken, was Sie für verbesserungsfähig halten, dann lassen Sie es uns wissen, etwa indem Sie eine elektronische Nachricht an

```
unterlagen@linupfront.de
```

schicken. (Zur Vereinfachung geben Sie am besten den Titel der Schulungsunterlage, die auf der Rückseite des Titelblatts enthaltene Revisionsnummer sowie die betreffende(n) Seitenzahl(en) an.) Natürlich können Sie uns auch über das Telefon, Telefax oder die »gelbe« Post erreichen. Vielen Dank!

LINUP FRONT

1

Kryptografie

Inhalt

Lernziele

- Grundbegriffe der Kryptografie kennen
- Symmetrische und asymmetrische Kryptografie unterscheiden können
- SSL und seine Probleme kennen

Vorkenntnisse

- Grundlegende Computerkenntnisse

1.1 Grundbegriffe

Geschichte

Geheimschriften und Codes sind uns seit den alten Babyloniern und Ägyptern bekannt. Der römische Feldherr und spätere Kaiser Julius Caesar, so heißt es, sandte Nachrichten an seine Vertrauten in einer Geheimschrift: Er ersetzte jedes »A« durch ein »D«, jedes »B« durch ein »E«, jedes »C« durch ein »F« und so weiter:

```
ANGRIFF AM MORGEN
DQJULII DP PRUJHQ
```

Um den Text entschlüsseln zu können, musste man die »Drei Zeichen weiter«-Regel kennen und rückgängig machen können.

Seit diesen Anfängen der Kryptografie haben die Anforderungen an sichere Verschlüsselung sich etwas geändert. Heutige Verschlüsselungsverfahren setzen die Verwendung von Computern voraus und führen zu Codes, die nach dem Stand heutigen Wissens nur durch massivsten Rechenaufwand über Jahrmillionen (!) hinweg gebrochen werden können – mit anderen Worten, gar nicht.

Zunächst aber einige Begriffsklärungen:

Kryptografie ist die Kunst (oder Wissenschaft), Nachrichten so umzuwandeln, dass sie für Uneingeweihte nicht lesbar sind.

Kryptanalyse dagegen ist die Kunst (oder Wissenschaft), solche umgewandelten Nachrichten zu entziffern (ohne der Adressat oder anderweitig dazu eingeladen zu sein).

Kryptologie ist das Studium von Kryptografie und Kryptanalyse.

Klartext ist die ursprüngliche, zu versteckende Nachricht,

Chiffretext das Resultat der Operation.

Verschlüsselung ist der Prozess der Überführung von Klartext in Chiffretext und

Entschlüsselung der gegensätzliche Vorgang.

Kryptosystem nennt man eine Klasse von Verfahren zur Ver- und Entschlüsselung von Nachrichten. Die einzelnen Verfahren sind gekennzeichnet;

Schlüssel sind die Kennzeichen, die einzelne Verfahren innerhalb eines Kryptosystems identifizieren. In Caesars Kryptosystem »Verschieben um n Buchstaben« kann man n als Schlüssel bezeichnen.

Brechen von Kryptoverfahren

Wie werden Verschlüsselungsverfahren gebrochen? Caesars Verfahren kommt man leicht auf die Schliche – jedenfalls wenn man es mit einem längeren Stück Chiffretext zu tun hat und eine plausible Vermutung darüber anstellen kann, in welcher Sprache die Nachricht verfasst ist. In den allermeisten deutschen Texten zum Beispiel sind die häufigsten Buchstaben »E«, »N« und »I«. Wenn man also feststellt, welche Buchstaben am häufigsten im Chiffretext vorkommen – beispielsweise »H«, »Q« und »L« –, kann man von dort aus auf den Verschiebungsfaktor n schließen. Klassische Kryptanalyse beruht also auf einer Kombination aus analytischem Denken, mathematischen Methoden, Geduld und Glück. In der modernen Kryptanalyse stehen dagegen die mathematischen Verfahren, etwa zur Faktorisierung großer Zahlen oder zur Bestimmung diskreter Logarithmen, im Vordergrund.

vollständige Suche

Eine Möglichkeit, ein bekanntes Kryptosystem zu brechen, basiert auf **vollständiger Suche** (engl. *brute-force search*). Hierbei probiert man jeden möglichen Schlüssel aus, bis man einen Schlüssel findet, der »passt« – mit dem die Nachricht entschlüsselt werden kann. Gute Kryptosysteme haben darum einen Schlüsselraum (eine Menge möglicher Schlüssel), der so groß ist, dass eine vollständige Suche nicht praktisch möglich ist.

Praktische Grenzen

Was »praktisch« ist, kann sich mit der Zeit ändern. Das DES-Verfahren, das seit

den 1970er Jahren verwendet wird, hat 2^{56}, also ungefähr 10^{17} mögliche Schlüssel. Das war in den 1970ern eine ganze Menge, ist heutzutage aber nicht mehr viel – tatsächlich wurde es 1998 für möglich erklärt, für eine Million Dollar einen Spezialrechner bauen zu können, der eine beliebige DES-verschlüsselte Nachricht in ein paar Stunden entschlüsselt (und etwa zur selben Zeit ein ähnlicher Apparat tatsächlich gebaut [EFFDES99]). Aus diesem Grund wird inzwischen offiziell von der Verwendung von DES für sensitive Inhalte abgeraten.

Heutzutage ist man sich einig, dass wirklich sichere Kryptosysteme nur solche sind, deren Sicherheit allein auf der Tatsache beruht, dass der Schlüssel geheim ist (das Kryptosystem selbst kann durchaus veröffentlicht werden). Dies ist als »Kerckhoffs' Prinzip« bekannt, nach Auguste Kerckhoffs von Nieuwenhof (1835–1903), der es 1883 in seinem klassischen Artikel *La Cryptographie militaire* zuerst postulierte. Sicherheit, die in wesentlichen Teilen auf der Geheimhaltung des Kryptosystems beruht, ist nicht real, da man davon ausgehen muss, dass jemand, der sich wirklich dafür interessiert, auf die eine oder andere Art vom Verfahren Kenntnis erlangen wird. Viele der heute in der Wirtschaft und auf dem Internet verwendeten Kryptosysteme sind veröffentlicht und wurden von unabhängigen Experten analysiert, wobei keine nennenswerten Schwachstellen gefunden wurden; im diplomatischen und militärischen Bereich ist Geheimhaltung der Verfahren selbst immer noch wichtig, aber mehr aus Gewohnheit und aufgrund des Prinzips, dass man niemandem Informationen auf dem silbernen Tablett präsentieren sollte.

Kerckhoffs' Prinzip

Leider verlassen sich viel zu viele Firmen und Institutionen auf schlecht gemachte Kryptografie. Um nur einige Beispiele zu nennen, wurden in den letzten Jahren etwa das A5/1-Verfahren, mit dem GSM-Mobiltelefonate verschlüsselt werden, und das MIFARE-Classic-System, auf dem viele »Smartcards« für Ausweis-, Fahrschein- und Bezahlsysteme beruhen, gebrochen.

»Starke« Kryptosysteme haben einen großen Schlüsselraum, der nicht vernünftig vollständig durchsucht werden kann, und produzieren Chiffretext, der den üblichen statistischen Tests zufolge als »zufällig« angesehen werden muss. Außerdem widerstehen sie allen bisher bekannten Angriffen. Ein nicht von unabhängigen Experten untersuchtes Kryptosystem ist verdächtig. – Allerdings machen diese Eigenschaften ein Kryptosystem nicht automatisch »stark«. Es ist sehr schwer, zu beweisen, *dass* ein Kryptosystem tatsächlich stark ist, da viele schwache Kryptosysteme dieselben Eigenschaften aufweisen. Allerdings lassen sich manche Kryptosysteme auf als schwierig bekannte mathematische Probleme »reduzieren«, so dass man Aussagen machen kann wie »Wenn jemand dieses Kryptosystem brechen kann, dann kann er auch große ganze Zahlen schnell faktorisieren.« (Das Faktorisieren großer ganzer Zahlen gilt vom Standpunkt der benötigten Rechenzeit her als schwierig.)

»Starke« Kryptosysteme

Es gibt tatsächlich ein Kryptosystem, dessen absolute Unbrechbarkeit mathematisch bewiesen werden kann, nämlich den **Einmalschlüssel** (engl. *one-time pad*). Hierbei wird ein Klartext der Länge m mit einem Schlüssel derselben Länge verschlüsselt, etwa durch (modulare) Addition oder Exklusiv-Oder. Wenn der Schlüssel dabei eine »echt« zufällige Zahlenfolge ist (die »pseudozufälligen« Zufallszahlen der meisten Programmiersprachen reichen nicht aus), so dass alle möglichen Schlüssel gleich wahrscheinlich sind, so ist ausgehend von einem gegebenen Chiffretext auch jeder Klartext gleich wahrscheinlich. Damit ist es nicht möglich, festzustellen, ob eine gegebene Entschlüsselung tatsächlich dem ursprünglichen Klartext entspricht:

Einmalschlüssel

Klartext 1	`ANGRIFFXAMX`
Schlüssel 1	`15930712527`
Schlüsseltext	`BSPUINGZFOE`
Schlüssel 2	`94274058511`
Klartext 2	`SONNENBRAND`

(um nur zwei mögliche Klartexte zu nennen). Einmalschlüssel-Verfahren sind prinzipiell die sichersten Kryptosysteme überhaupt; ihr Nachteil besteht aber

darin, dass die Schlüssel irgendwie verteilt werden müssen und natürlich nicht der »Gegenseite« in die Hände fallen dürfen. Aus diesem Grund sind sie in der Praxis, jedenfalls was Computeranwendungen angeht, völlig unbrauchbar – jedes bisher vorgeschlagene Produkt auf der Basis von Einmalschlüsseln hatte mehr oder weniger schwerwiegende Probleme, die typischerweise mit der Erzeugung oder Verteilung der Schlüssel zu tun haben.

Im praktischen Sinne beruht Kryptografie auf der Idee, etwas großes Geheimes (den Klartext) durch etwas kleines Geheimes (den Schlüssel) zu schützen. Bei Einmalschlüsseln ist das kleine Geheime nicht wirklich kleiner als das große Geheime – wenn Sie also den Schlüssel vertraulich übertragen können, können Sie theoretisch auch gleich die Nachricht vertraulich übertragen (wobei Sie natürlich versuchen können, Ihren Kommunikationspartnern vorab Schlüsselmaterial »auf Vorrat« zu schicken).

Das Problem mit Schlüsseln für Einmalschlüssel-Verfahren ist, dass sie »echten« Zufall brauchen – und der ist nicht so leicht zu finden, wie man vielleicht denkt. Ihr Linux-System zum Beispiel stellt in der »Datei« `/dev/random` Zufallszahlen zur Verfügung, die aus Systemeffekten wie dem Zeitabstand zwischen Tastaturanschlägen, Maus- und Festplattencontroller-Interrupts abgeleitet werden – aber Voraussetzung dafür ist natürlich, dass diese Effekte auch passieren und beobachtet werden können. Mehr darüber steht in `random`(4).

Im wirklichen Leben wurde das berühmte »rote Telefon« zwischen dem Weißen Haus und dem Kreml in Moskau (jedenfalls der Gerüchteküche nach) mit einem Einmalschlüssel-Verfahren gesichert, was nur ging, weil das »Telefon« in Wirklichkeit eine Fernschreib- und keine Sprachverbindung war (ist?), so dass keine riesigen Mengen von Schlüsselmaterial benötigt wurden.

Ein theoretisch nicht brechbares Kryptosystem kann trotzdem gebrochen werden, wenn Bedienungsfehler passieren, etwa wenn sehr viele Nachrichten mit demselben Klartext anfangen (im militärischen Gebrauch etwa mit dem Datum und einer Einheitenbezeichnung) oder derselbe Klartext mit verschiedenen Kryptosystemen verschlüsselt wird. In der Geschichte der Kryptanalyse gibt es sogar Fälle, in denen es gelang, feindlichen Agenten bekannten Klartext unterzuschieben und diesen dann im Chiffretext wiederzufinden. Selbst Einmalschlüssel-Verfahren sind trotz ihrer theoretischen Sicherheit leicht zu brechen, wenn derselbe Schlüssel mehrmals zur Verschlüsselung verschiedener Klartexte verwendet wurde (daher der Name).

Glücklicherweise ist es nicht notwendig, dass ein Kryptosystem nicht brechbar ist. Es muss nur so lange Angriffen standhalten können, dass die damit verschlüsselten Daten wertlos sind, bis das Kryptosystem gebrochen wird.

Übungen

1.1 [2] Entschlüsseln Sie den folgenden Text: »JMWGLIVW JVMXD JMWGLX JVMWGLI JMWGLI« (Es handelt sich dabei um einen bekannten deutschen Satz.)

1.2 [3] Schreiben Sie ein Shellskript, das zu einem gegebenen Chiffretext sämtliche 26 »julianischen« Klartexte erzeugt.

1.2 Symmetrische Kryptosysteme

Ein Schlüssel zum Ver- und Entschlüsseln

Bei symmetrischen Kryptosystemen wird derselbe Schlüssel zum Ver- und Entschlüsseln verwendet. Das heißt, die kommunizierenden Parteien – traditionell

»Alice« und »Bob« genannt – einigen sich auf einen geheimen Schlüssel. Anschließend kann Alice eine Nachricht mit diesem Schlüssel verschlüsseln und den Schlüsseltext an Bob schicken. Bob verwendet den Schlüssel, um die Nachricht wieder zu entschlüsseln. Verfügen nur Alice und Bob über den Schlüssel, so können sie ungestört verschlüsselte Nachrichten austauschen, ohne dass ein Angreifer diese mithören könnte.

Gängige symmetrische Kryptosysteme

Gängige symmetrische Kryptosysteme sind beispielsweise DES, dreifaches DES (engl. *triple DES* oder »3DES«), der DES-Nachfolger AES (Rijndael), IDEA, RC4 oder RC5. Von diesen gilt DES nicht mehr als sicher, dreifaches DES dagegen schon; auch die anderen genannten Verfahren können noch benutzt werden. Auch die klassische »Enigma«-Verschlüsselungsmaschine des deutschen Militärs im zweiten Weltkrieg, bekannt aus vielen Romanen und Filmen, beruhte auf einem symmetrischen Kryptosystem.

Symmetrische Kryptosysteme sind normalerweise relativ effizient und lassen sich gut in Hardware implementieren. Mit Schlüssellängen zwischen 112 und 256 (oder notfalls mehr) Bit sind die Schlüsselräume hinreichend groß, um vollständige Suche aussichtslos zu machen: Die 56 Bit von DES sind heutzutage, wie erwähnt, zu wenig, da 1998 mit einem Aufwand von einer Million US-Dollar für Hardware die Entschlüsselung einer beliebigen Nachricht in 3,6 Stunden möglich war. Bei dreifachem DES mit 112-Bit-Schlüsseln ist unter denselben Voraussetzungen von einer Rechenzeit von etwa 10^{13} Jahren auszugehen; für eine vollständige Suche im 128-Bit-Schlüsselraum von IDEA wären schon rund 10^{18} Jahre nötig.

Man kann argumentieren, dass ein ansonsten fehlerfreies symmetrisches Kryptosystem mit einem 256-Bit-Schlüsselraum nicht durch »rohe Gewalt« zu brechen ist, weil die Energie im Universum nicht ausreicht, um alle Schlüssel auch nur aufzuzählen (geschweige denn eine interessante Nachricht damit zur Probe zu entschlüsseln). Diese Argumentation ist verlockend, da sie rein auf thermodynamischen Erwägungen beruht und keine Annahmen über Algorithmen und ähnliches macht. Das bedeutet, dass auch uns technisch millionenfach überlegene Außerirdische das Kryptosystem nicht brechen können, solange ihre Computer aus Materie sind und sich im Universum befinden.

Die Argumentation geht ungefähr so [Sch96, S. 157]: Um in einem physikalischen System Informationen darstellen zu können, wird Energie benötigt. Um (salopp gesagt) ein Bit zu kippen, brauchen Sie mindestens die Energie kT, wobei T die absolute Temperatur des Systems ist und k die Boltzmann-Konstante, namentlich rund $1{,}38 \cdot 10^{-23}$ J/K. Ein idealer Computer würde im besten Fall, bei der »Hintergrundtemperatur« des Universums von $T = 3{,}2$ K also, etwa $4{,}4 \cdot 10^{-23}$ J für einen Bitwechsel benötigen (wenn der Computer kälter sein soll, brauchen wir zusätzliche Energie, um ihn zu kühlen). Die Sonne liefert uns pro Jahr etwa $1{,}21 \cdot 10^{34}$ J, und das ist genug Energie für um die $2{,}7 \cdot 10^{56}$ Bitwechsel, also ungefähr um von 0 bis 2^{187} zu zählen (wie gesagt, Rechnen ist da noch nicht mit einkalkuliert). Eine Supernova produziert so um die 10^{44} J, und das reicht dafür aus, bis 2^{219} zu zählen. Das läßt uns noch eine gewisse Sicherheitsreserve.

Schlüsselverteilung

Das Problem bei symmetrischen Kryptosystemen ist weniger die Ver- und Entschlüsselung, sondern die Schlüsselverteilung. Alice und Bob müssen sich irgendwie im Vorhinein auf einen Schlüssel einigen, den sie zur Kommunikation verwenden wollen, denn der Schlüssel kann natürlich nicht Teil der verschlüsselten Nachricht sein. Das ist insbesondere dann schwierig, wenn man mit jemandem vertraulich kommunizieren will, den man gar nicht kennt und der sich weit von einem weg aufhält, und wird nur schlimmer, wenn viele Parteien beteiligt sind: Gesellt sich eine dritte Person zu Alice und Bob, sind bereits 3 Schlüssel nötig, damit die drei paarweise vertraulich kommunizieren können; wenn die vierte Person dazu kommt, werden es 6 Schlüssel und so weiter. Allgemein benötigen n Personen $k(n) = n \cdot (n-1)/2$ geheime Schlüssel, also für große Werte von n etwa n^2

viele, die *a priori* erzeugt und vertraulich verteilt werden müssen. Daraus folgt, dass symmetrische Kryptografie für wichtige Anwendungen wie elektronischer Handel – wo ein Anbieter potentiell Millionen von Kunden bedient, die er vorher nicht kennt – nicht zu gebrauchen ist.

1.3 Asymmetrische Kryptosysteme

Kryptosysteme, asymmetrische

Die heute übliche Lösung des Schlüsselverteilungsansatzes beruht auf **asymmetrischen Kryptosystemen**. Während in symmetrischen Kryptosystemen derselbe Schlüssel zum Ver- und Entschlüsseln der Nachricht verwendet wird, benutzen asymmetrische Kryptosysteme zwei verschiedene Schlüssel, einen **privaten** und einen **öffentlichen Schlüssel**. Der öffentliche Schlüssel kann allgemein bekanntgegeben werden – etwa im Telefonbuch, auf Visitenkarten oder in Gestalt eines Zertifikats auf einer Web-Präsenz, während der private Schlüssel geheimgehalten werden muss.

privater Schlüssel

öffentlicher Schlüssel

Angenommen, Alice und Bob verwenden ein asymmetrisches Kryptosystem. Wenn Alice Bob eine vertrauliche Nachricht schicken möchte, verschlüsselt sie sie mit Bobs *öffentlichem* Schlüssel. Eine solche Nachricht kann nur mit dem dazugehörigen *privaten* Schlüssel wieder entschlüsselt werden – und den hat ja nur Bob. Bob kann seine Antwort mit Alices öffentlichem Schlüssel verschlüsseln und Alice entschlüsselt sie mit ihrem privaten Schlüssel. Wo man $O(n^2)$ viele symmetrische Schlüssel brauchte, damit n Personen paarweise kommunizieren können, reichen jetzt noch n öffentliche Schlüssel – eine große Ersparnis.

Ersparnis

Die Idee asymmetrischer Kryptografie wurde zuerst 1976 von Whitfield Diffie und Martin Hellman veröffentlicht [DH76], die allerdings kein entsprechendes Kryptosystem vorschlugen, sondern lediglich eine Methode zum Schlüsselaustausch. Das bekannteste asymmetrische Kryptosystem ist das 1977 erfundene **RSA**, benannt nach seinen Erfindern Ronald Rivest, Adi Shamir und Leonard Adleman. RSA beruht darauf, dass es kein bekanntes Verfahren gibt, das große Zahlen in annehmbarer Zeit in Primfaktoren zerlegen kann, während Diffie-Hellman-Schlüsselaustausch auf diskreten Logarithmen beruht. Eine dritte Art von asymmetrischer Kryptografie beruht auf »elliptischen Kurven«, also Lösungen der Gleichung $y^2 = x^3 + ax + b$, deren diskrete Logarithmen ebenfalls schwierig zu berechnen sind. Die Kurven selbst lassen sich dagegen sehr effizient in Hardware berechnen.

RSA

Bei RSA sind die öffentlichen Schlüssel im wesentlichen Zahlen der Form $n = pq$ (die technischen Details sind etwas komplexer), wobei p und q »große« Primzahlen sind – »groß« heißt hier in der Gegend von ein paar hundert Dezimalstellen. RSA-Schlüssel sind also sehr lang (1024 Bits und mehr) im Vergleich zu Schlüsseln für symmetrische Kryptosysteme, bei denen 128 Bits schon eine ganze Menge sind. Das heißt nicht, dass RSA zehnmal (oder gar fast 2^{1000}mal) sicherer wäre als zum Beispiel IDEA. Die Sicherheit von RSA ergibt sich daraus, dass wir nicht wissen, wie wir n effizient in p und q zerlegen können.

Schlüssellänge

Sicherheit von RSA

Es gibt Verfahren zur Faktorisierung von Zahlen, die weniger ineffizient sind als der naheliegende Ansatz »Wir dividieren die Zahl k durch alle ganzen Zahlen kleiner $\sqrt{k}$, bis es aufgeht«. Das derzeit (Februar 2010) bekannteste ist das »allgemeine Zahlkörpersieb« (engl. *general number field sieve*), mit dem es Thorsten Kleinjung vom BSI im Dezember 2009 gelang, eine binär 768-stellige (bzw. dezimal 232-stellige) Zahl zu faktorisieren [KAF+10]. Dafür waren über zwei Jahre und der Einsatz etlicher hundert über das Internet verbundener Rechner nötig – das Projekt hätte nach Aussage der Autoren der Studie einen 2,2-GHz-AMD-Opteron-Prozessor mit einem Kern ungefähr 1500 Jahre (!) beschäftigt.

Wenn die Zahlentheorie und die Hardwaretechnik weiter so große Fortschritte machen wie bisher, dann kann man damit rechnen, dass es im Jahr 2013 möglich sein wird, 1024-Bit-Zahlen zu faktorisieren (jedenfalls sehen die Autoren von

[KAF+10] das so, und die sollten es wissen). 2048-Bit-RSA-Schlüssel sind also noch für eine Weile sicher. Man muss auch bedenken, dass die Hardwarefortschritte, die die Faktorisierung größerer Zahlen ermöglichen, es auf der anderen Seite erlauben, längere Schlüssel effizient zu verwenden. Da es wesentlich leichter ist, einen ein paar hundert Bit längeren RSA-Schlüssel zu erzeugen, als denselben Schlüssel zu faktorisieren, ziehen die Faktorisierer auf lange Sicht den kürzeren.

Der Längenunterschied zwischen öffentlichen RSA-Schlüsseln und den Schlüsseln von symmetrischen Verfahren wie IDEA beruht einfach darauf, dass bei den symmetrischen Verfahren im Prinzip jedes Bit im Schlüssel »zählt«. Ein öffentlicher RSA-Schlüssel dagegen ist ein Produkt von 2 Primzahlen einer bestimmten Länge k – aber nicht jede k-stellige Zahl ist tatsächlich eine Primzahl. Die k Bit unterscheiden also nicht zwischen 2^k potentiellen Schlüsseln, wie das bei den symmetrischen Verfahren wäre, sondern zwischen viel weniger. Aus diesem Grund braucht man längere Schlüssel, um dieselbe »Kompliziertheit« zu erreichen.

Eine andere Frage, die Sie sich stellen könnten, ist, ob es überhaupt genug Primzahlen mit (zum Beispiel) 1024 Binärstellen gibt, wenn jeder von uns eine für seinen oder ihren RSA-Schlüssel braucht. Zuerst läßt sich dazu sagen, dass es grundsätzlich unendlich viele Primzahlen gibt[1]. Die Anzahl der Primzahlen, die kleiner sind als eine gegebene Zahl r, hat die Mathematiker so sehr beschäftigt, dass man ihr einen eigenen Namen gegeben hat, nämlich $\pi(r)$ – und wir wissen inzwischen, dass $\pi(r)$ sich asymptotisch an $r/\ln r$ annähert. Wie dies eingegrenzt und schließlich festgenagelt wurde, ist eine spannende[2] Geschichte, die sich über gut 200 Jahre erstreckt; bewiesen wurde dieses »Primzahlentheorem« endgültig und unabhängig 1896 von Jacques Salomon Hadamard und Charles Jean de la Vallée-Poussin. Das Ganze lehrt uns, dass es $\pi(2^{1024}) - \pi(2^{1023})$, also etwas mehr als 10^{305} Primzahlen mit 1024 Binärstellen gibt[3]. In Anbetracht der Tatsache, dass das Universum nur um die 10^{80} Atome enthält, sollten die eigentlich ausreichen, zumal wenn wir beachten, dass wir mittelfristig sowieso zu 2048-Bit-Primzahlen übergehen sollen.

Das ganze funktioniert natürlich nur, solange niemand eine effiziente Faktorisierungsmethode findet (wobei »effizient« für Informatiker bedeutet, dass der Aufwand sich weniger als exponentiell zur Länge der zu faktorisierenden Zahl verhält). Wenn morgen jemand ein solches Verfahren entdeckt oder einen hinreichend leistungsfähigen Computer baut, der nicht an die Einschränkungen heutiger Rechner gebunden ist (Stichwort »Quantencomputer«), ist RSA erledigt – und damit, da ein solches Verfahren gleichzeitig die Lösung für diverse andere lange als »kompliziert« bekannte Probleme liefern würde, asymmetrische Kryptografie insgesamt. Da es nach unserem heutigen Kenntnisstand keine Alternative dazu gibt, können wir nur hoffen, dass dieser Fall nicht eintritt; als schwacher Trost mag dienen, dass die besten mathematischen Gehirne dieser Erde sich seit Jahrzehnten an dem Problem die Zähne ausgebissen haben, ohne dass ein entsprechender Durchbruch erzielt wurde. Auf der anderen Seite gibt es Leute, die glauben, dass man sich in den Kellern der NSA oder der Mafia längst ins Fäustchen lacht …

Quantencomputer

Von den Quantencomputern sollte man sich einstweilen noch nicht zu viel versprechen, was die Hardware angeht. Den Algorithmus für die effiziente Faktorisierung gibt es allerdings schon, namentlich »Shors Algorithmus« (nach Peter Shor, [Sho99] und [Aar07]).

[1] Bewiesen hat das schon Euklid, ungefähr um 300 vor Christus: Nehmen wir an, $p_1 = 2 < p_2, < ... < p_r$ sind alle Primzahlen. Wir betrachten die Zahl $P = p_1 p_2 \ldots p_r + 1$. Sei p eine Primzahl, durch die P teilbar ist. p ist keine der Zahlen $p_1, \ldots, p_r$, da sonst auch die Differenz von P und $p_1 p_2 \ldots p_r$, namentlich 1, durch p teilbar sein müßte (denken Sie ans Distributivgesetz!), aber das geht nicht. Also ist p eine Primzahl, die nicht in unserer Liste steht – ein Widerspruch!

[2] Naja, wenn Sie Mathematik spannend finden …

[3] Wolfram Alpha (`www.wolframalpha.com`) sei Dank!

Grundsätzlich sollten wir erwähnen, dass es nicht *sicher* (im Sinne von »mathematisch bewiesen«) ist, dass man große Zahlen faktorisieren können *muss*, um RSA zu brechen. Es könnte ein völlig anderes Verfahren geben, aber da dieses Verfahren dann sozusagen im Nebeneffekt nützlich zur Faktorisierung großer Zahlen wäre, wäre es den Faktorisierungs-Forschern wahrscheinlich schon aufgefallen. Zumindest aus dem Bauch heraus besteht also kein unmittelbarer Anlass zur Sorge.

Gegen gut gemachte symmetrische Kryptosysteme mit langen, zufälligen Schlüsseln ist dagegen mit Ansätzen, die auf vollständiger Suche beruhen, wenig zu wollen. 128 Bit sind sicherlich mehr als genug. Unangenehmerweise beruht die Sicherheit eines Kryptosystems nicht nur auf der Schlüssellänge (sonst wäre Kryptanalyse inzwischen ein höchst uninteressanter Zeitvertreib). Die traurige Tatsache ist, dass viele Kryptosysteme Schwächen haben, die Inhalte kompromittieren können, auch ohne dass man alle möglichen Schlüssel ausprobiert. Institutionen wie die NSA können es sich finanziell leisten, Herden von exzellenten Kryptografen zu beschäftigen, die Kryptosysteme nach Problemen absuchen. Ein solcher Ansatz ist in der akademischen Welt nicht möglich, so dass man sich auf *peer review* beschränkt – ein Kryptosystem wird veröffentlicht, und wenn es interessant genug aussieht, finden sich vielleicht wissenschaftliche Kollegen, die es genau unter die Lupe nehmen. Das muss nicht automatisch klappen – das Kryptosystem WEP für drahtlose LAN-Vernetzung war jahrelang veröffentlicht, bevor man einen ganz kapitalen Fehler in dem Algorithmus entdeckte –, aber *peer review* ist allemal viel besser als kein *peer review*. Kryptosysteme, deren Innereien geheimgehalten werden, »um die Sicherheit des Verfahrens nicht zu beeinträchtigen«, verdienen nach Kerckhoffs' Prinzip kein Vertrauen. (Der branchenübliche Name für solche Verfahren ist *snake oil* – ein Begriff für medizinische Wundermittel, die von herumreisenden Händlern gegen alle denkbaren Beschwerden verkauft wurden, von abstehenden Ohren bis zu eingeschlafenen Füßen, ohne jedoch wirklich zu wirken [Cur98].)

Sicherheit eines Kryptosystems

peer review

Asymmetrische Kryptosysteme wie RSA involvieren in der Regel Herumrechnen mit großen Zahlen und sind daher sehr rechenaufwendig, ganz im Gegensatz zu symmetrischen Kryptosystemen, die normalerweise nur Bits umkippen und durcheinanderwürfeln, was sich sehr effizient programmieren oder sogar in Hardware realisieren läßt. Außerdem lassen asymmetrische Kryptosysteme nur die Verschlüsselung kleiner Datenmengen zu. In der Praxis verwendet man deshalb einen Ansatz, der die Stärken von symmetrischer und asymmetrischer Kryptografie zu kombinieren versucht. Nehmen wir an, Alice möchte mit Bob kommunizieren:

Kombinierter Ansatz

1. Alice wählt einen zufälligen Schlüssel für ein symmetrisches Kryptosystem ihrer Wahl (zum Beispiel einen 128-Bit-Schlüssel für AES).
2. Alice verschlüsselt diesen Schlüssel, den **Sitzungsschlüssel** (*session key*) mit Bobs öffentlichem RSA-Schlüssel und verschickt ihn an Bob.

Sitzungsschlüssel

3. Bob entschlüsselt den Sitzungsschlüssel mit seinem privaten RSA-Schlüssel. In diesem Moment verfügen Alice und Bob über denselben Sitzungsschlüssel für AES.
4. Alice und Bob verwenden AES für ihre vertrauliche Kommunikation.

Nach diesem Strickmuster operieren alle heute gebräuchlichen kryptografischen Softwarepakete – SSL, SSH, PGP, GnuPG, …

Übungen

1.3 [3] Was sind die Primfaktoren von 58216819?

1.4 Kryptografische Hash-Funktionen und MACs

Integrität

Eine wichtige Aufgabe kryptografischer Verfahren ist die Sicherung der **Integrität** von Nachrichten: Man möchte erreichen, dass ein Inhalt – etwa eine elektronische Überweisung oder ein Vertrag – nicht manipuliert werden kann, ohne dass das auffällt. Da es oft nicht sinnvoll möglich ist, Versionen riesiger Dokumente zu vergleichen, verwendet man dafür kryptografische Hash-Funktionen.

Prüfsumme

Die Idee hinter kryptografischen Hash-Funktionen ist, dass ein großer Klartext T (möglicherweise Millionen von Bytes lang) auf eine »Prüfsumme« (engl. *message digest*) h fester Länge abgebildet wird. Die Hash-Funktion, H, soll die folgenden Eigenschaften haben:

Einfachheit Es ist einfach, $h = H(T)$ zu berechnen.

Unumkehrbarkeit Es ist schwierig, mit einer gegebenen Prüfsumme h einen Klartext T zu bestimmen, so dass $H(T) = h$ gilt.

Schwache Kollisionsresistenz Es ist schwierig, mit einem gegebenen Klartext T einen anderen Klartext T' zu finden, so dass $H(T) = H(T')$ gilt.

Starke Kollisionsresistenz Es ist schwierig, zwei beliebige Nachrichten T und T' zu finden, so dass $H(T) = H(T')$ gilt.

(»Schwierig« ist hier eine Abkürzung für »nicht mit vertretbarem Rechenaufwand möglich«.)

Ohne die schwache Kollisionsresistenz könnte Bob Alice in Schwierigkeiten bringen: Wenn Alice ein Dokument T signiert, indem sie eine digitale Signatur auf $h = H(T)$ anwendet, und Bob ein Dokument T' fabrizieren kann, das ebenfalls h als Prüfsumme hat, dann könnte er überzeugend behaupten, dass Alice das Dokument T' signiert hat statt T.

Ohne die starke Kollisionsresistenz könnte Alice Bob wie folgt in Schwierigkeiten bringen:

1. Alice erstellt zwei Versionen T und T' eines Vertrags. T ist vorteilhaft für Bob und T' ist unvorteilhaft für ihn.
2. Alice sorgt dafür, dass die beiden Versionen dieselbe Prüfsumme h haben. Dazu könnte sie zum Beispiel zusätzliche Leerzeichen am Zeilenende (vor dem Zeilentrenner) einfügen. Wenn sie das in 32 Zeilen tut oder nicht tut, kann sie schon 2^{32} Dokumentversionen erzeugen.
3. Alice bringt Bob dazu, T zu signieren – auf der Basis eines Verfahrens, wo in Wirklichkeit nur $H(T)$ signiert wird.
4. Alice ersetzt T durch T' und verklagt Bob.

Aus dem vorigen Absatz folgt, dass Sie bei jedem Dokument, das Ihnen zur (digitalen) Unterschrift vorgelegt wird, ein paar »kosmetische« Änderungen machen sollten. (Zusätzliche Leerzeichen funktionieren auch so herum.)

Kollisionsangriff

Um zu einer gegebenen Prüfsumme h ein Dokument T zu finden, so dass $H(T) = h$ gilt, müssen Sie, wenn h n Bit lang ist, mit »roher Gewalt« potentiell 2^n viele Dokumente ausprobieren. Wenn es Ihnen dagegen um zwei beliebige Dokumente mit derselben (beliebigen) Prüfsumme geht, müssen Sie nur $2^{n/2}$ Dokumente ausprobieren. Man spricht von einem **Kollisionsangriff** (oder »Geburtstagsangriff«, *birthday attack*[4]).

MD5
SHA-1

Gängige kryptografische Hash-Funktionen sind **MD5** und **SHA-1**. MD5 ver-

[4] Der Begriff basiert auf der folgenden Beobachtung: In einem Raum (naja Ballsaal) müssen sich 253 Personen befinden, damit die Wahrscheinlichkeit, dass eine davon genau am selben Tag im Jahr Geburtstag hat wie Sie, größer ist als 50%. Wenn allerdings nur irgend zwei beliebige Personen im Saal am selben Tag im Jahr Geburtstag haben sollen, dann wird die 50%-Schwelle schon bei 23 Personen (!) übersprungen. (Eine genauere Erklärung mit Herleitung finden Sie unter `http://de.wikipedia.org/wiki/Geburtstagsparadoxon`.)

wendet 128 Bit lange Prüfsummen, während die von SHA-1 160 Bit lang sind. Beide Verfahren sind heute diskreditiert; nach einem zeitgemäßen Nachfolger wird aktiv gesucht.

MD5 [RFC1321] kam schon in den 1990er Jahren in Zweifel, aber 2004 wurden die ersten Kollisionen demonstriert. Lenstra u. a. [LWW05] beschreiben die Konstruktion von zwei verschiedenen X.509-Zertifikaten mit unterschiedlichen öffentlichen Schlüsseln und derselben MD5-Prüfsumme. Sotirov u. a. [SSA+08] erklären, wie sie ein normales Zertifikat in ein Zertifikat für eine Zertifizierungsstelle mit derselben MD5-Prüfsumme umgewandelt haben. – MD5 wird von Zertifizierungsstellen wie VeriSign inzwischen nicht mehr für Zertifikate verwendet.

SHA-1 basiert auf denselben Grundideen wie MD5 und wird darum inzwischen auch mit Argwohn betrachtet. Es gibt Kollisionsangriffe, die mit rund 2^{52} Schritten statt der eigentlich theoretisch nötigen 2^{80} auskommen. Das ist zunächst nicht wirklich ein Problem, aber wirft einen Schatten auf das Verfahren. SHA-2, das Prüfsummen von 224 Bit oder mehr produziert, gilt noch als sicher.

Im Moment läuft ein Wettbewerb des NIST (US-Pendant zum DIN), bei dem nach einem Nachfolger für SHA-1 und SHA-2 gesucht wird. Mit einem Ergebnis wird 2012 gerechnet; derzeit sind noch 14 Kandidaten im Rennen. Der Sieger wird dann als »SHA-3« standardisiert.

Wenn Sie es sich aussuchen können, sollten Sie für neue Anwendungen von MD5 Abstand nehmen und SHA-1 (oder besser SHA-256, die 256-Bit-Version von SHA-2) verwenden.

Linux (oder genauer gesagt die GNU-Coreutils, die Bestandteil praktisch jeder wesentlichen Linux-Distribution sind) unterstützt die Kommandos `md5sum`, `sha1sum` und `sha256sum`, die MD5-, SHA-1- oder SHA-256-Prüfsummen berechnen:

```
$ echo HALLO | md5sum; echo H@LLO | md5sum
67fb5721c31f77a084c06e668239ef24  -
4c06fc4c28a0a015a47adcec3f2915f8  -
```

Die beiden Eingabetexte `HALLO` und `H@LLO` unterscheiden sich nur in einem einzigen Bit; trotzdem sind die resultierenden Prüfsummen völlig verschieden.

Anwendungen

Kryptografische Hash-Funktionen werden für viele Zwecke eingesetzt, zum Beispiel für das übliche Unix-Anmeldeverfahren, bei dem Benutzerkennwörter nicht im Klartext, sondern nur als kryptografische Prüfsumme gespeichert werden (in `/etc/shadow`). Gibt ein Benutzer ein Kennwort ein, wird es »gehasht« und das Resultat mit der gespeicherten Prüfsumme verglichen. Dies macht es einem Angreifer schwerer, Nutzen aus einer gemopsten Kennwortdatenbank zu ziehen.

Eine andere wichtige Anwendung ist die Sicherung der Integrität von Softwaredistributionen. Heutzutage werden oftmals kryptografische Prüfsummen von Softwarepaketen veröffentlicht, die der Empfänger heranziehen kann, um sich von der Integrität des Pakets zu überzeugen; die Hoffnung ist, dass »trojanisierte« Pakete auf diese Weise aufgedeckt werden können, bevor gutgläubige Benutzer sie installieren und Schaden davontragen. Leider ist das nicht so, da ein Angreifer außer einem modifizierten Softwarepaket auch dessen kryptografische Prüfsumme einschmuggeln kann, so dass die Manipulation verborgen bleibt. Um eine solche Manipulation aufzudecken, benötigt man sogenannte engl. *keyed hashes* oder engl. *message authentication codes* (**MACs**). Hierbei wird außer den Daten von Interesse noch ein geheimer Schlüssel mit »gehasht«, der nur dem Ersteller des MACs und seinen Empfängern bekannt ist. Ein Angreifer sollte solche MACs nicht fälschen können.

MACs

Übungen

1.4 [1] Bestimmen Sie die MD5-, SHA-1- und SHA-256-Prüfsummen des folgenden Textes: »Das Pferd frisst keinen Gurkensalat«.

1.5 [2] Wenn Sie mit einer kryptografischen Hashfunktion H, einer Nachricht N und einem Geheimnis g einen MAC als $H(g, N)$ (die Hashfunktion angewendet auf das Geheimnis gefolgt von der Nachricht) berechnen, laufen Sie Gefahr, dass ein Angreifer Ihre Nachricht um eigenes Material N' verlängern und mit einem gültigen MAC versehen kann. Es ist beser, zum Beispiel $H(g, H(g, N))$ als MAC zu verwenden. Warum?

1.6 [2] Bestimmen Sie (wie in der vorigen Aufgabe beschrieben) $H(g, H(g, N))$ mit den Parametern H = MD5, g = `blafasel` und N = `Blaukraut␣bleibt␣Brautkleid`.

1.7 [3] Das `APOP`-Kommando [RFC1725] ist eine Erweiterung des populären POP3-Protokolls, die den Versand von Kennwörtern im Klartext über das Netz vermeidet. Beschreiben Sie kurz die Arbeitsweise der Kennwortauthentisierung mit APOP und diskutieren Sie die Vor- und Nachteile der Verwendung von APOP gegenüber gewöhnlichem POP.

1.5 Digitale Signaturen

MACs haben wie symmetrische Kryptoverfahren das Problem, dass Sender und Empfänger ein gemeinsames Geheimnis brauchen. Ein anderer Ansatz zur Sicherung der Authentizität einer Nachricht beruht nicht auf gemeinsamen Geheimnissen, sondern auf asymmetrischer Kryptografie: Wenn Alice eine Nachricht mit ihrem privaten Schlüssel *signiert*, dann kann jeder, der über Alices öffentlichen Schlüssel verfügt, nachvollziehen, dass Alice die Nachricht signiert hat. In der Praxis signiert man wegen der bekannten Einschränkungen asymmetrischer Kryptoverfahren jedoch keine ganzen Nachrichten, sondern nur kryptografische Prüfsummen der Nachrichten.

Digitale Signaturen helfen, ein sehr fundamentales Problem der asymmetrischen Kryptografie zu lösen. Asymmetrische Kryptografie behebt zwar den größten Nachteil der symmetrischen Kryptoverfahren – die n^2 geheimen Schlüssel –, aber ersetzt es durch ein eigenes neues Problem, nämlich »Wie stelle ich sicher, dass Alices öffentlicher Schlüssel, den ich gerade von irgendwoher bekommen habe, wirklich Alices öffentlicher Schlüssel ist und nicht der von irgendeinem Angreifer?« Hierfür verwendet man gerne **Zertifikate**, bei denen eine vertrauenswürdige Instanz die »Echtheit« eines öffentlichen Schlüssels bestätigt und diese Aussage durch ihre digitale Signatur auf dem Zertifikat untermauert. In Kapitel 3 wird das genauer besprochen.

Authentizität von öffentlichen Schlüsseln

Zertifikate

Übungen

1.8 [3] In der öffentlichen Verwaltung und im Geschäftsverkehr ist die Rede davon, die digitale Signatur als Authentizitätsnachweis einer Willenserklärung unter gewissen Umständen rechtlich auf eine Stufe mit der »echten« Unterschrift zu stellen. Was halten Sie davon und warum?

Kommandos in diesem Kapitel

md5sum Bestimmt die MD5-Prüfsumme von Dateien oder Standardeingabe
md5sum(1) 20

sha1sum Bestimmt die SHA1-Prüfsumme von Dateien oder Standardeingabe
sha1sum(1) 20

Zusammenfassung

- Kryptografie ist seit dem Altertum bekannt.
- »Kerckhoffs' Prinzip« fordert, dass die Sicherheit eines Kryptosystems allein im Schlüssel liegen muss; das Verfahren selbst kann (und sollte) publiziert werden.
- Einmalschlüssel-Verfahren sind theoretisch beweisbar sicher, praktisch aber nicht sinnvoll einzusetzen.
- Symmetrische Kryptosysteme verwenden denselben Schlüssel zum Ver- und Entschlüsseln; sie sind effizient implementierbar, aber durch den erforderlichen Schlüsselaustausch unpraktisch.
- Asymmetrische Kryptoverfahren unterscheiden öffentliche und private Schlüssel. Sie beheben das Problem des Schlüsselaustauschs, aber erfordern wesentlich mehr Rechenaufwand. Über ihre letztendliche Sicherheit gibt es noch keine endgültige Aussage.
- Kryptografische Hash-Funktionen bestimmen zu einer gegebenen Eingabe eine Prüfsumme, die keine Rückschlüsse auf die Eingabe zulässt, aber sehr empfindlich von ihr abhängt. Zur Authentizitätsprüfung werden MACs eingesetzt, bei denen die betrachteten Daten zusammen mit einem Geheimnis »gehasht« werden.
- Asymmetrische Kryptografie ermöglicht digitale Signaturen, indem man die Daten (oder eine kryptografische Prüfsumme davon) mit seinem privaten Schlüssel verschlüsselt (signiert). Mit dem öffentlichen Schlüssel kann die Authentizität der Daten geprüft werden.

Literaturverzeichnis

Aar07 Scott Aaronson. »Shor, I'll do it«, Februar 2007. Laut Peter Shor "'die beste Darstellung von Quantencomputern für den Mann auf der Straße"'.
http://scottaaronson.com/blog/?p=208

Cur98 Matt Curtin. »Snake Oil Warning Signs: Encryption Software to Avoid«, April 1998. http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

DH76 Whitfield Diffie, Martin E. Hellman. »New Directions in Cryptography«. *IEEE Transactions on Information Theory*, November 1976. **IT-22**(6):644–654.

EFFDES99 EFF. »Cracking DES«, Januar 1999. http://www.eff.org/descracker.html

KAF+10 Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, et al. »Factorization of a 768-bit RSA modulus«. Cryptology ePrint Archive, Report 2010/006, 2010. http://eprint.iacr.org/2010/006

LWW05 Arjen Lenstra, Xiaoyun Wang, Benne de Weger. »Colliding X.509 Certificates«. Cryptology ePrint Archive, Report 2005/067, 2005.
http://eprint.iacr.org/2005/067

RFC1321 R. Rivest. »The MD5 Message-Digest Algorithm«, April 1992.
http://www.ietf.org/rfc/rfc1321.txt

RFC1725 J. Myers, M. Rose. »Post Office Protocol – Version 3«, November 1994.
http://www.ietf.org/rfc/rfc1725.txt

Sch96 Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996, 2. Auflage. ISBN 0-471-12845-7, 0-471-11709-9. Deutsch als *Angewandte Kryptographie* (Addison-Wesley).

Sho99 Peter W. Shor. »Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer«. *SIAM Review*, 1999. **41**(2):303–332. http://link.aip.org/link/?SIR/41/303/1

SSA+08 Alexander Sotirov, Marc Stevens, Jacob Appelbaum, et al. »MD5 considered harmful today«, Dezember 2008. http://www.win.tue.nl/hashclash/rogue-ca/

LINUP FRONT

2

SSL, TLS, OpenSSL und `mod_ssl`

Inhalt

Lernziele

- Die Prinzipien von SSL und TLS kennen
- OpenSSL kennen und installieren können
- `mod_ssl` kennen und installieren können

Vorkenntnisse

- Grundkenntnisse über Kryptografie (Kapitel 1)
- Kenntnisse über Konfiguration und Betrieb eines Apache-Servers
- Ggf. Kenntnisse über die Übersetzung und Inbetriebnahme von C-Programmen unter Linux

2.1 SSL

2.1.1 Grundlagen

Geschichte SSL, kurz für *Secure Socket Layer*, ist das am weitesten verbreitete Sicherheitsprotokoll für TCP/IP. SSL kann beliebige TCP-Verbindungen verschlüsseln und die Kommunikationspartner authentisieren. Es bildet die Grundlage für
HTTPS **HTTPS**, aber kann auch zur Verschlüsselung vieler anderer Protokolle wie POP3, IMAP, SMTP, FTP, TELNET, … dienen.

SSL war ursprünglich eine Erfindung der Firma Netscape. Das Geschäftsmodell von Netscape basierte auf der Idee, einen WWW-Browser weithin verfügbar zu machen, der mit einem WWW-Server (dem von Netscape implementierten) vertraulich kommunizieren konnte. Die Möglichkeit, damit zum Beispiel sicher Waren bestellen zu können, sollte ein Alleinstellungsmerkmal der Netscape-Browser-Server-Kombination werden und dazu führen, dass viele Organisationen den Server kaufen (der Browser wurde zwar – quasi – verschenkt, der Server hingegen gewiss nicht). Leider wurden alsbald Details von SSL publik, die eine frei verfügbare Nachimplementierung zuließen, welche in anderen WWW-Servern (etwa Apache) eingesetzt werden konnte. Dies führte mittelbar zum Versagen des Geschäftsmodells von Netscape und zum Ableben der Firma.

TLS 

Wenn wir über SSL reden, meinen wir auch **TLS** (engl. *Transport Layer Security*), das sozusagen die Fortführung von SSL als herstellerunabhängiger Standard unter der Ägide der IETF darstellt. TLS 1.0 [RFC2246] entspricht im wesentlichen SSL 3.0, enthält aber einige signifikante Unterschiede, die eine Interoperabilität von TLS 1.0 und SSL 3.0 ausschließen. Aktuell ist TLS 1.2 [RFC5246].

TLS 1.0 und der Nachfolgestandard TLS 1.1 [RFC4346] unterscheiden sich vor allem darin, dass TLS 1.1 zusätzlichen Schutz gegen bestimmte Angriffe bietet und das Registrieren von Parametern bei der IANA erlaubt. Neuigkeiten in TLS 1.2 gegenüber TLS 1.1 umfassen unter anderem die Abkehr von MD5 und SHA-1 bei pseudozufälligen Funktionen und digitalen Signaturen, die offizielle Aufnahme von TLS-Erweiterungen und AES in den Standard sowie administrative Vereinfachungen etwa bei der Aushandlung von Hash- und Signaturalgorithmen zwischen TLS-Clients und -Servern.

Einsatz in Anwendungen SSL und TLS setzen üblicherweise auf einem zuverlässigen Transportprotokoll wie TCP auf, während Anwendungsprotokolle wie HTTP, SMTP oder IMAP SSL und TLS als eine Art »erweitertes TCP« zu sehen bekommen. Nachdem eine Verbindung über SSL initialisiert ist, gibt es aus der Sicht eines Programms keinen Unterschied mehr zu einer normalen TCP-Verbindung.

Ursprünglich war TLS nur für TCP gedacht. Es gibt allerdings einen Standard für TLS über datagrammorientierte Protokolle wie UDP [RFC4347].

Es gibt zwei verschiedene Ansätze dafür, SSL in Anwendungen zu integrieren:

- Man kann ein Anwendungsprotokoll so, wie es ist, über eine mit SSL gesicherte Verbindung abwickeln. Das heißt, die »SSL-Version« eines Dienstes bekommt eine eigene Portnummer zugeordnet, und wenn ein Client sich auf dieser Portnummer mit einem Server verbindet, werden als allererstes Authentisierung und Verschlüsselung organisiert, bevor das eigentliche Anwendungsprotokoll ans Ruder kommt. Diese Vorgehensweise kommt vor allem bei HTTPS zum Einsatz, für das der TCP-Port 443 reserviert ist (»normales« HTTP verwendet den Port 80), aber sie ist auch bei anderen Diensten wie POP3S (TCP-Port 995) oder IMAPS (TCP-Port 993) anzutreffen.

- Man kann ein Anwendungsprotokoll so erweitern, dass eine Verbindung zunächst unverschlüsselt und unauthentisiert über den »normalen« Port aufgebaut und anschließend durch ein Kommando im Protokoll in eine verschlüsselte und authentisierte Verbindung umgewandelt wird. Dieser Ansatz wird zum Beispiel bei SMTP verwendet [RFC3207], wo Server als Erweiterung ein `STARTTLS`-Kommando anbieten können; bei der Begrüßung des Clients gibt der Server bekannt, dass er TLS unterstützt, und anschließend kann der Client mit `STARTTLS` die bestehende Verbindung in eine TLS-Verbindung umwandeln.


Traditionell wird SSL eher mit dem ersten und TLS eher mit dem zweiten Ansatz assoziiert. Das hat aber keine tiefere Bedeutung.

Wenn wir nicht ausdrücklich etwas Gegenteiliges verkünden, verwenden wir ab jetzt »SSL« und »TLS« als Synonyme, statt immer »SSL und TLS« zu sagen.

Protokollablauf Hier ist kurz beschrieben, wie eine SSL-Verbindung aufgebaut wird:

1. Der Client verbindet sich mit dem Server und schickt eine Liste der *cipher suites*, die er unterstützt. Eine *cipher suite* ist eine Kombination aus einer Schlüsselaustauschmethode, einem Kryptosystem für die Datenübertragung und einer kryptografischen Hash-Funktion zur Bestimmung von MACs.

2. Der Server wählt die stärkste *cipher suite* aus, die er auch unterstützt, und teilt dem Client diese Auswahl mit.

3. Anschließend schickt der Server sein Zertifikat an den Client.

   
   Dieser Schritt ist übrigens optional (!) – wenn die Authentisierung des Servers Ihnen nicht wichtig ist, können Sie SSL auch ohne Zertifikate verwenden. Statt (typischerweise) RSA mit dem öffentlichen Schlüssel im Zertifikat wird dann Diffie-Hellman-Schlüsselaustausch verwendet, für den kein öffentlicher Schlüssel und mithin kein Zertifikat nötig ist.

4. Der Client wählt eine Zufallszahl, verschlüsselt sie mit dem öffentlichen Schlüssel des Servers (aus dem Zertifikat) und schickt sie dem Server zurück.

5. Der Server entschlüsselt die Zufallszahl mit seinem privaten Schlüssel. (An dieser Stelle verabschieden die beiden sich von allen unbefugten Zuhörern, die nicht über den privaten Schlüssel des Servers verfügen.)

6. Beide Seiten verwenden die Zufallszahl zum Erzeugen allfälliger Schlüssel für die *cipher suite*.

Durch diese Aushandlung wird – sofern nichts schiefgegangen ist – eine gesicherte Verbindung etabliert, über die Client und Server kommunizieren können. Beim Verbindungsabbau ist nichts Besonderes zu beachten.


Nach dem Empfang des Serverzertifikats kann der Client theoretisch prüfen, ob das Zertifikat (noch?) gültig ist, etwa über eine Zertifikats-Rückrufliste oder indem er mit der Zertifizierungsstelle Kontakt aufnimmt. In der Praxis passiert das aber eher selten.

Natürlich gibt es a priori keine Garantie, dass nicht ein Angreifer die Verbindung zum Server abfängt und dem Client sein eigenes Zertifikat unterschiebt. Der Angreifer könnte dann eine eigene Verbindung zum eigentlichen Server aufbauen und die Daten mitlesen. Man spricht von einem *Man-in-the-middle*-Angriff. Um

Man-in-the-middle-Angriff

einen solchen Angriff zu vermeiden, muss der Client nicht nur die Signatur auf dem Zertifikat prüfen, sondern auch sicherstellen, dass die Institution, die das Zertifikat ausgestellt hat, vertrauenswürdig ist. WWW-Browser enthalten in der Regel eine Reihe von Zertifikaten, die der Hersteller des Browsers für vernünftig hält. Der Server kann auch vom Client ein Zertifikat fordern, das wird aber selten gemacht.

Nach der Feststellung der Authentizität einigen Client und Server sich auf die zu verwendenden kryptografischen Verfahren, etablieren entsprechend dem oben erklärten Strickmuster einen Sitzungsschlüssel und können so vertraulich kommunizieren. Die SSL-Verbindung sieht dann aus wie eine gewöhnliche TCP-Verbindung. SSL integriert verschiedene populäre Kryptosysteme und erlaubt es Client und Server, nur bestimmte davon anzubieten. Auf diese Weise ist es möglich, Verfahren, die sich als angreifbar erweisen, kompatibel durch andere zu ersetzen. Zum Beispiel bieten viele WWW-Browser die gängigen Kryptosysteme außer in der üblichen Stärke in »schwachen« Varianten mit 40- oder 56-Bit-Schlüsseln an, um (antiquierten) US-Exportrestriktionen Sorge zu tragen. Da die starken Verfahren inzwischen universell zur Verfügung stehen, können Sie die schwachen Varianten in der Browserkonfiguration ausschließen und laufen so keine Gefahr, irrtümlich doch nicht so vertraulich zu kommunizieren, wie Sie dachten.

US-Exportrestriktionen

Besonders perfide ist der Umstand, dass SSL zu Fehlersuchzwecken einen Modus unterstützt, bei dem das Protokoll zwar wie üblich abgewickelt wird, aber keine tatsächliche Verschlüsselung stattfindet. Sie sollten darauf achten, dass Sie diesen Modus nicht versehentlich aktivieren.

Übungen

2.1 [1] Wie viele Zertifizierungsstellen kennt Ihr Browser? Wie ist es möglich, Zertifizierungsstellen das Vertrauen zu entziehen?

2.2 [!2] Welche symmetrischen Verschlüsselungsverfahren unterstützt die Version von SSL bzw. TLS in Ihrem Browser? Welche asymmetrischen? Welche kryptografischen Hash-Verfahren?

2.3 [2] Warum verwendet HTTP den Ansatz mit einem separaten Port und nicht etwas wie das `STARTTLS`-Kommando von SMTP?

2.1.2 Probleme mit SSL und TLS

SSL und TLS sind verbreitet und funktionieren im großen und ganzen gut; dennoch sollten einige Probleme mit dem System nicht unerwähnt bleiben. Die meisten dieser Probleme lassen sich allerdings mit entsprechender Sorgfalt zumindest weitgehend entkräften.

Schlüsselverwaltung Ein SSL-Server braucht den zu seinem Zertifikat passenden privaten Schlüssel. Dieser private Schlüssel muss irgendwo auf dem Server zur Verfügung stehen, und das sinnvollerweise ohne dass er Angreifern zu leicht in die Hände fallen kann. Am besten bringen Sie ihn auf einer Smartcard unter, die unter normalen Umständen nicht ausgelesen werden kann. Wenn das nicht möglich ist, haben Sie im Grunde zwei Möglichkeiten: Den Schlüssel unverschlüsselt auf der Platte liegen lassen, oder ihn verschlüsseln. Die letztere Option ist sicherer, verlangt aber, dass zumindest bei einem Server-Reboot jemand zugegen ist, der das Kennwort für die Entschlüsselung des privaten Schlüssels angeben kann. In jedem Fall kann ein Angreifer, der `root`-Rechte auf Ihrem Server hat, den unverschlüsselten privaten Schlüssel erlangen, indem er z. B. den SSL-Server mit einem Debugger beobachtet.

privater Schlüssel

Smartcard

Reboot

Schlechte Zertifikate Mit dem privaten Schlüssel eines Servers kann ein Angreifer einen Server kompromittieren. Oft muss man sich aber gar nicht solche Mühe geben, wenn es gelingt, die Zertifizierungsstelle dazu zu bringen,

ein Zertifikat zu bestätigen, das unrichtige Informationen enthält. (VeriSign, die größte kommerzielle Zertifizierungsstelle, signierte Anfang 2001 einige Zertifikate, die fälschlich vorgaben, Microsoft zu gehören.) So ein Zertifikat sieht für alle Benutzer gültig aus. Prinzipiell kann eine Zertifizierungsstelle Zertifikate »zurückrufen« und für ungültig erklären, indem sie auf sogenannte Zertifikats-Rückruflisten (engl. *certificate revocation lists*, CRLs) gesetzt werden; in der Praxis kann es einerseits sehr lange dauern, bis ein als falsch bekanntes Zertifikat auf einer CRL landet, und andererseits kümmern Clients sich meistens nicht um CRLs (wann haben Sie zuletzt eine CRL heruntergeladen?). Die zeitnahe und effiziente Verteilung von CRLs ist auch problematisch. SSL unterstützt prinzipiell CRLs, aber da es keinen standardisierten Verteilungsmechanismus gibt, lassen CRLs sich zumeist (noch) nicht sinnvoll einsetzen. Dies ist ein starkes Argument dafür, private Schlüssel möglichst effektiv zu schützen.

Eine Alternative zu CRLs ist das »Online Certificate Status Protocol«, OCSP [RFC2560]. Die Idee hinter OCSP ist, dass SSL-Clients (etwa Browser) mit der Zertifizierungsstelle oder einem entsprechend befugten Dritten Kontakt aufnehmen, um die Gültigkeit eines ihnen vorgelegten Zertifikats »in Echtzeit« zu überprüfen.

Die gängigen Browser unterstützen heutzutage OCSP, aber ob die Zertifizierungsstellen das tun, steht auf einem ganz anderen Blatt.

Der Vorteil von OCSP gegenüber CRLs ist, dass eine Zertifizierungsstelle keine (möglicherweise gigantischen) Rückruflisten veröffentlichen muss. Auf der anderen Seite muss die Zertifizierungsstelle bereit sein, (möglicherweise gigantische Mengen von) OCSP-Anfragen sehr schnell zu beantworten. (Stellen Sie sich vor, Sie sind die Zertifizierungsstelle, die das Zertifikat für `www.google.com` ausgestellt hat.) Es gibt Mittel und Wege, die »Kosten« für OCSP auf die Zertifikatsinhaber abzuwälzen – Stichwort »OCSP Stapling« [RFC4366, Abschnitt 3.6] –, aber das ist noch viel weniger verbreitet als OCSP selbst.

Zertifikatsprüfung Clients müssen Zertifikate genau prüfen, bevor sie sich auf deren öffentliche Schlüssel verlassen. Dies ist zum Beispiel für HTTPS die Aufgabe des WWW-Browsers, und es zeigt sich, dass die in der Software enthaltenen Prüfroutinen oftmals fehlerhaft sind.

Zum Beispiel wurde 2009 bekannt, dass die meisten SSL-Implementierungen ein Nullbyte in der Mitte eines Servernamens im Zertifikat als Ende des Namens ansehen – was aber nicht der Spezifikation entspricht[1]. Das heißt, Sie können als Eigentümer der Domain `example.com` ein Zertifikat für den Server

`www.paypal.com`⟨*Nullbyte*⟩`.example.com`

beantragen (und bekommen das womöglich auch problemlos ausgestellt) und auf Ihrem Server veröffentlichen. Aus der Sicht der gängigen Browser sieht das aber aus wie ein Zertifikat für `www.paypal.com` – Sie könnten also ohne Weiteres als *man in the middle* aktiv werden [Mar09].

Es kommt sogar noch schlimmer: Wenn Sie sich ein Zertifikat für

*⟨*Nullbyte*⟩`.example.com`

besorgen, passt(e) das aus der Sicht der SSL-Implementierung von Mozilla (Hersteller von Firefox) auf *jede* Domain.

[1] Nullbytes in der Mitte eines Servernamens kommen im wirklichen Leben nicht vor. Da die offiziellen Standards für Zertifikate – X.509 – aber eine andere Datencodierung verwenden, stören sie auch nicht, sondern rutschen bei Aktionen wie dem Signieren von Zertifikaten in der Zertifizierungsstelle einfach so durch.

Selbst wenn das nicht der Fall ist, neigen viele Anwender dazu, beliebige Zertifikate blindlings zu akzeptieren. Beispielsweise bietet shop.example.com die Möglichkeit an, sich seinen »virtuellen Einkaufskorb« randvoll zu laden. Anschließend geht es an die Kasse, natürlich werbewirksam »SSL-gesichert«, doch o Wunder! Das Zertifikat gehört zu einem völlig anderen Server, nämlich ssl.inkasso.de (Name erfunden), dem Institut, das die Inhaber von shop.example.com der Bequemlichkeit und Rentabilität halber damit betraut haben, die Kreditkartenabwicklung, Bonitätsprüfung usw. für ihre Kunden zu übernehmen. Prinzipiell gibt es keinen Grund zu der Annahme, dass shop.example.com und ssl.inkasso.de *irgendetwas* miteinander zu tun haben – das Szenario ist von einem klassischen *Man-in-the-middle*-Angriff nicht zu unterscheiden, womit die angebliche »Sicherheit« von SSL völlig unterlaufen wird. Im wirklichen Leben scheint das aber kein Hinderungsgrund zu sein.

In einer kaum zu überbietenden Kombination aus Überheblichkeit und Ignoranz haben die Kreditkartenhersteller sich in der jüngeren Vergangenheit etwas einfallen lassen, das alle Versuche, Benutzer über die Gefahren von *Man-in-the-middle*-Szenarien und die erforderliche Vorsicht aufzuklären, ad absurdum führt. Die Rede ist von »3-D Secure«, besser bekannt als »SecureCode« (für MasterCard) und »Verified by Visa«. Die Idee dahinter ist, dass Kunden, die online etwas mit ihrer Kreditkarte bezahlen sollen, vom *Kartenaussteller* ein Formular präsentiert bekommen, wo sie ein Kennwort eingeben müssen. Eine der größeren Macken dieses Ansatzes ist, dass das entsprechende Formular von einem völlig anderen Server kommt als dem des Anbieters, in der Regel völlig anders aussieht, und daher für den Benutzer völlig ununterscheidbar von einem überaus plumpen *Man-in-the-middle*- oder Phishing-Angriff ist. Näheres zu diesem Thema findet sich in [MA10].

Zu guter Letzt sollte man darauf aufmerksam machen, dass viele Browser Dutzende von Zertifizierungsstellen kennen und als vertrauenswürdig akzeptieren, bei denen im Einzelfall nicht ohne weiteres klar ist, ob das auch angebracht ist. Zwar können Sie als Benutzer solche Voreinstellungen modifizieren; ob die große Masse der WWW-Anwender diese Möglichkeiten nutzt oder überhaupt kennt, darf aber getrost als zweifelhaft angesehen werden.

Schlechte Zufallszahlen SSL zieht zur Erzeugung von Sitzungsschlüsseln und anderen Geheimnissen Zufallszahlen heran, die auf dem betreffenden Rechner irgendwie generiert werden. Diese Zufallszahlen müssen »kryptografisch« zufällig sein; die (Pseudo-)Zufallszahlengeneratoren, die in die meisten Programmiersprachen integriert sind, reichen hierfür nicht aus. Betriebssysteme wie Linux sammeln zufällige Daten auf der Basis von physikalischen Prozessen – etwa den sehr genau gemessenen Zeitabständen zwischen Tastendrücken oder Mausbewegungen des Benutzers – und stellen sie Anwendungsprogrammen zur Verfügung (in Linux etwa über die »Geräte« /dev/random und /dev/urandom). Ein Maß für die »Zufälligkeit« von Zufallszahlen ist deren **Entropie**; wenn ein Bit mit derselben Wahrscheinlichkeit 0 sein kann wie 1, enthält es 1 Bit Entropie. Hat ein Datensatz 3 Bit Entropie, heißt das, dass man ihn mit einer Wahrscheinlichkeit von $1/2^3$ (1 von 8) auf Anhieb richtig erraten kann; nach höchstens 8 Versuchen bekommt man ihn auch so heraus, wobei man im Durchschnitt mit 4 Versuchen rechnen muss. Entsprechend muss man für einen 128-Bit-Schlüssel mit 128 Bit Entropie im Durchschnitt 2^{127} Versuche einkalkulieren (eine neununddreißigstellige Dezimalzahl) – ein ziemlich aussichtsloses Unterfangen, nur setzt das voraus, dass wirklich 128 Bit Entropie da sind, um Zufallszahlen zu erzeugen. Beispielsweise konnten Ian Goldberg und David Wagner 1996 Schwächen in der Initialisierung des Zufallszahlengenerators der Netscape-Implementierung von SSL Version 2 nachweisen, die dazu führten,

Entropie

dass maximal nur 47 Bit Entropie verwendet wurden. Mit einigen Tricks konnten Goldberg und Wagner diese Tatsache ausnutzen, um SSL-Sitzungen innerhalb von weniger als einer halben Minute zu kompromittieren [GW96].

Unsichere Kryptografie SSL und TLS als Verfahren sind inzwischen relativ gut analysiert worden. TLS gilt, sinnvollen Gebrauch vorausgesetzt, als hinreichend sicher. Frühere Versionen bis zur ebenfalls recht verbreiteten Version SSL 2.0 hatten allerdings mehr oder weniger gravierende Sicherheitslücken. Aus diesem Grund sollten Sie SSL 2.0 nicht mehr unterstützen, auch wenn Ihre Software Ihnen die Möglichkeit dazu gibt. Sie sollten außerdem, wie bereits erwähnt, die 40- und 56-Bit-Versionen stärkerer Verfahren ausschließen. Heute akzeptable Kryptosysteme sind RC4, 3DES oder (in neueren TLS-Versionen) AES.

Ganz ohne Probleme ist auch TLS natürlich nicht. Im August 2009 wurde ein Angriff auf SSL 3.0 und alle TLS-Versionen gefunden, der ungefähr wie folgt aussieht: TLS gibt bei einer *bestehenden* Verbindung den Kommunikationspartnern die Möglichkeit, neue Kryptografieparameter, Schlüssel usw. auszuhandeln. Das findet *innerhalb* der bestehenden TLS-Verbindung statt, das heißt, die Datenpakete für die neue Aushandlungsprozedur werden wie andere Datenpakete verschlüsselt, aber sind nicht anderweitig an die Verbindung gekoppelt. Der Angreifer baut also eine eigene Verbindung zum TLS-Server auf (er kann mit ihm kommunizieren, wie er möchte) und übernimmt dann die Verbindung zwischen Client und Server[2]. Dabei leitet er zunächst die Daten des Clients über seine eigene verschlüsselte Verbindung an den Server weiter. Der Client verhandelt aus seiner Sicht in eigener Sache *im Klartext* mit dem Server, während der Server schon eine verschlüsselte Verbindung sieht und denkt, dass der Client *neu verhandelt*. Sobald die Neuverhandlung abgeschlossen ist, kann der Angreifer nicht mehr mitlesen, aber das macht nichts.

Als Illustration des vorigen Absatzes: Im wirklichen Leben authentisieren Web-Server ihre Clients typischerweise einmal über Benutzername und Kennwort und setzen dann einen Cookie für die Sitzung, den der Client ab da mit jeder neuen Anfrage mitschickt. Wenn der Angreifer am Anfang einer HTTP-Anfrage beliebige Sachen einbauen kann, könnte er an `pizzadienst.example.com` etwas schicken wie

Cookie

```
GET /pizza?typ=4-jahreszeiten;adresse=angreifer HTTP/1.1
X-Ignoriere-Dies:
```

(ohne Zeilentrenner am Ende). Wenn der Client seinerseits etwas schickt wie

```
GET /pizza?typ=hawaii;adresse=client HTTP/1.1
Cookie: meincookie
```

ergibt sich insgesamt die Anfrage

```
GET /pizza?typ=4-jahreszeiten;adresse=angreifer HTTP/1.1
X-Ignoriere-Dies: GET /pizza?typ=hawaii;adresse=client HTTP/1.1
Cookie: meincookie
```

Die Client-Anfrage ab dem `GET` und die Antwort auf diese Anfrage werden zwischen Client und Server verschlüsselt übertragen und der Angreifer kann sie nicht mehr lesen. Das heißt, im Gegensatz zum

[2] Etwa weil er einen Router zwischen den beiden kontrolliert. In der Praxis würde er wahrscheinlich *erst* die Verbindung vom Client abfangen und *dann* seine eigene Verbindung zum Server öffnen, aber das ist egal.

klassischen *Man-in-the-middle*-Angriff hat der Angreifer keinen Zugriff auf möglicherweise vertrauliche Daten. Allerdings kann der Angreifer von »Nebeneffekten« profitieren, denn er bekommt seine Pizza auf die Rechnung des Clients. (Die Neuaushandlung der Verbindung findet zwischen dem `X-Ignoriere-Dies` und dem `GET` des Clients statt, aber da sie auf der SSL-Ebene passiert, bekommt der Web-Server davon nichts mit.)

Effizienz SSL ist, wie nicht anders zu erwarten, merkbar langsamer als gewöhnliches TCP/IP. Schon beim Verbindungsaufbau müssen durch den Austausch von Zertifikaten und Schlüsseln beträchtliche Datenmengen übertragen werden; außerdem wird hier (langsame) asymmetrische Kryptografie verwendet. Nachdem die Sitzung aufgebaut wurde, ist der Effizienzverlust nicht mehr ganz so krass, aber immer noch merkbar, hauptsächlich, weil mehr Daten übertragen werden müssen. Die symmetrische Kryptografie ist im Vergleich auf heutiger Hardware kein so großes Problem mehr. Die gängigen SSL-Implementierungen versuchen durchaus, auf Effizienzanforderungen Rücksicht zu nehmen, machen aber keine Kompromisse, was die Sicherheit angeht. Ein Verfahren, das behauptet, dasselbe zu tun wie SSL, aber viel schneller zu sein, sollte mit einigem Argwohn betrachtet werden.

Übungen

2.4 [1] Wie können Sie bei Ihrem Lieblingsbrowser Informationen über eine SSL-Verbindung wie die Identitäten des Servers und der Zertifizierungsstelle oder die Art des verwendeten Verschlüsselungsverfahrens abrufen?

2.5 [4] Vergleichen Sie die Geschwindigkeit einer HTTPS-Verbindung mit der einer gewöhnlichen HTTP-Verbindung, indem Sie die Zeit messen, die die Übertragung einer großen Ressource benötigt. (Dazu brauchen Sie wahrscheinlich einen SSL-Server gemäß Kapitel 4. Verwenden Sie ein Programm wie `wget` zum Anfordern von Ressourcen. Diese Übung eignet sich dazu, bei der Kursnachbereitung in Angriff genommen zu werden.)

2.2 OpenSSL

OpenSSL ist eine Implementierung des SSL-Protokolls als Bibliothek, die in Anwendungsprogramme integriert werden kann. OpenSSL ist von SSLeay abgeleitet, einer SSL-Implementierung, die von Eric A. Young und Tim J. Hudson zwischen 1995 und 1998 erstellt wurde. OpenSSL besteht aus einer Bibliothek, die neben diversen grundlegenden kryptografischen Verfahren alle Versionen des SSL-Protokolls inklusive TLSv1 implementiert, und einer Reihe von Programmen, die den Gebrauch dieser Verfahren erleichtern oder beispielhaft in Anspruch nehmen. Mit OpenSSL können Sie beispielsweise Zertifikate generieren und verwalten, Dokumente ver- und entschlüsseln oder Verbindungen zu SSL-Servern aufbauen.

Bibliothek

Werkzeuge

OpenSSL ist »freie Software«, das heißt, sie steht im Quellcode zur Verfügung und darf von jedem verwendet, angepaßt und weiterverbreitet werden. Insbesondere können Sie sich selbst davon überzeugen, dass die kryptografischen Verfahren korrekt implementiert worden sind und dass keine »Falltüren« existieren, die Unbefugten Zugang zu verschlüsseltem oder zu verschlüsselndem Material gewähren.

Die meisten Distributionen enthalten OpenSSL in vorübersetzter Form, so dass die Installation sich darauf beschränkt, das entsprechende Paket zu installieren. Den Quellcode von OpenSSL können Sie von `http://www.openssl.org/` abrufen, wenn Sie es bevorzugen, das Paket selbst zu übersetzen.

Installation

Zur Installation von OpenSSL brauchen Sie Perl 5 und einen ANSI-C-Compiler,

etwa GCC. Heutige Linux-Distributionen enthalten normalerweise beides; natürlich können Sie auch Perl und GCC von Quellcode installieren, aber das wird hier nicht weiter besprochen. Packen Sie die Distributionsdatei (`openssl.tar.gz` oder so ähnlich) an einer geeigneten Stelle aus und lesen Sie die darin enthaltene Datei `INSTALL`. Zur Installation sind die folgenden Schritte notwendig:

```
$ ./config
$ make
$ make test
$ make install
```

(den letzten Schritt müssen Sie wahrscheinlich als `root` durchführen). Das `config`-Skript passt das OpenSSL-Paket an die lokalen Gegebenheiten an und kann über diverse Optionen gesteuert werden; normalerweise installiert OpenSSL sich in das Verzeichnis `/usr/local/ssl`, aber Sie können das mit der Option `--prefix` ändern. Ferner können Sie einstellen, ob *shared libraries* erzeugt werden sollen oder nicht, ob plattformabhängiger Assembler-Code zur Beschleunigung eingebunden werden soll und ob bestimmte Verschlüsselungsverfahren mit eingebaut oder weggelassen werden sollen. Die Details stehen in der Datei `INSTALL` in der OpenSSL-Distribution.

Übungen

2.6 [1] Warum ist es wichtig, dass ein Softwarepaket wie OpenSSL als freie Software im Quellcode zur Verfügung steht?

2.7 [25] Finden und dokumentieren Sie eine Sicherheitslücke im aktuellen OpenSSL-Paket.

2.3 mod_ssl

Das Apache-Modul `mod_ssl` stellt eine Verbindung zwischen Apache und OpenSSL her. Mit `mod_ssl` unterstützt Apache **HTTPS**, also HTTP über SSL-Verbindungen; ein Apache-Server kann sich gegenüber Clients durch X.509-Zertifikate authentisieren und seinerseits Client-Authentisierung auf der Basis von clientseitigen X.509-Zertifikaten akzeptieren. `mod_ssl` wurde von Ralf S. Engelschall implementiert.

HTTPS

Seit Apache 2.0 wird `mod_ssl` mit dem Apache-Server mitgeliefert. Es ist also nicht mehr nötig, dass Sie es separat besorgen und vom Quellcode selbst übersetzen. Wenn Sie eine Linux-Distribution verwenden, sollten Sie trotzdem nachschauen, ob das normale Apache-Paket schon `mod_ssl` enthält oder ob Sie dafür noch etwas Anderes installieren müssen.

Die meisten Linux-Distributionen unterstützen `mod_ssl` direkt; es muss also kein großer Aufwand getrieben werden, um die Software lauffähig zu machen. Ihnen als Endbenutzer bleibt lediglich die Installation geeigneter Zertifikate, die Sie sich vorher verschaffen müssen (siehe hierzu die Folgekapitel). Ferner können Sie das Verhalten des Moduls und damit des Apache-SSL-Servers in relativ weiten Kreisen beeinflussen, indem Sie in der `httpd.conf`-Datei (der tatsächliche Name hängt von Ihrer Linux-Distribution ab) entsprechende Direktiven setzen.

Distributionen

Wenn Ihre Distribution keine vorkonfigurierten SSL-Direktiven in der `httpd.conf`-Datei mitliefert, dann ist es klug, die SSL-Konfiguration in eine eigene Datei, etwa `httpd-ssl.conf`, auszulagern, die in der eigentlichen `httpd.conf`-Datei mit `Include` eingebunden wird. Dies vereinfacht die Wartung von `httpd.conf`, wenn neue Versionen davon erscheinen.

Direktiven Die wichtigsten Direktiven für `mod_ssl` sind:

`SSLEngine on|off` Schaltet SSL für den betreffenden Server ein oder aus. Auch für virtuelle Server nützlich.

Seit Apache 2.1 ist auch der Wert `optional` erlaubt, der es erlaubt, eine bestehende HTTP/1.1-Verbindung nach [RFC2817] zu einer TLS-Verbindung hochzustufen. Allerdings gibt es noch keine Browser, die das unterstützen.

`SSLProtocol` [**+**|**-**]⟨*Protokoll*⟩ **...** Diese Direktive gibt an, welche SSL-Protokollversionen akzeptabel sind. Clients können nur eine der erlaubten Protokollversionen benutzen (Groß- und Kleinschreibung egal):

`SSLv2` Das ursprüngliche SSL-Protokoll von Netscape. Hat bekannte Sicherheitslücken und sollte darum am besten ausgeschlossen werden, um zu verhindern, dass Clients versehentlich seinen Gebrauch aushandeln.

`SSLv3` Die Nachfolgeversion von SSL 2.0 und der Vorgänger von TLS 1.0. Wird von fast allen Browsern unterstützt.

`TLSv1` Das Nachfolgeprotokoll von SSL 3.0. Wird heute von den meisten Browsern unterstützt. Neuere TLS-Versionen tauchen auf dem Radar von `mod_ssl` (noch) nicht auf.

`All` Alle Versionen.

Eine vernünftige Voreinstellung wäre

```
SSLProtocol All -SSLv2
```

`SSLCipherSuite` ⟨*Schlüsselliste*⟩ Gibt an, welche SSL-Verschlüsselungsverfahren akzeptabel sind. Details der Syntax und der möglichen Verfahren finden Sie in der Dokumentation zu `mod_ssl`.

`SSLRandomSeed` ⟨*Kontext*⟩ ⟨*Quelle*⟩ [⟨*Bytes*⟩] OpenSSL benötigt Zufallszahlen von »kryptografischer« Qualität, also solche, die nicht von Angreifern erraten werden können (die typischen »Zufallsgeneratoren« von gängigen Programmiersprachen kommen nicht in Frage). Die Bibliothek enthält einen guten Zufallszahlengenerator, der aber mit hinreichender Entropie. Diese Direktive beschreibt die Quellen von Entropie für die erste Initialisierung (⟨*Kontext*⟩ ist `startup`) oder für den SSL-Verbindungsaufbau (⟨*Kontext*⟩ ist `connect`). Die möglichen Quellen sind

`builtin` Die »eingebaute« Quelle ist immer vorhanden, liefert aber gerade beim Start sehr wenig Entropie (es stehen wenige »zufällige« Bits zur Verfügung). Wenn möglich, sollte zumindest für `startup` eine bessere Quelle verwendet werden.

`file:`⟨*Dateiname*⟩ Der benannten Datei werden zufällige Bytes entnommen. Linux verfügt über zwei »Geräte« namens `/dev/random` und `/dev/urandom`, die Entropie aus dem laufenden System sammeln und sich gut als Quelle eignen. Der Unterschied zwischen den beiden ist, dass `/dev/random` nur so viel Entropie liefert, wie tatsächlich zur Verfügung steht: Wenn Sie 256 zufällige Bytes lesen wollen und es sind gerade nur 100 vorhanden, dann blockiert der Lesevorgang, bis genug zufällige Bytes zur Verfügung gestellt werden können. `/dev/urandom` tut das nicht, aber dafür sind die resultierenden zufälligen Bytes möglicherweise nicht ganz so gut wie die von `/dev/random`.

`exec:`⟨*Programmname*⟩ **und** **`egd:`**⟨*Socketname*⟩ Noch zwei Möglichkeiten für Zufallsquellen: ein externes Programm und der *Entropy-Gathering Daemon*. Diese sind unter Linux nicht so wichtig, weil es die `*random`-Geräte gibt. Näheres steht in der `mod_ssl`-Dokumentation.

Zusammenfassung

- SSL und TLS sind die verbreitetsten Sicherheitsprotokolle und können beliebige TCP-Verbindungen verschlüsseln und authentisieren, insbesondere HTTP.
- Am Anfang einer SSL/TLS-Verbindung authentisiert der Server sich gegenüber dem Client über sein Zertifikat. Anschließend werden die zu benutzenden kryptografischen Verfahren ausgehandelt und Schlüssel ausgetauscht, bevor die Verbindung wie eine gewöhnliche TCP-Verbindung verwendet werden kann.
- Zur Prüfung der Authentizität des Servers prüft der Client die Signatur(en) auf dessen Zertifikat und entscheidet anhand der Vertrauenswürdigkeit der Zertifizierungsstelle.
- OpenSSL ist eine freie Implementierung von SSL und TLS zur Integration in eigene Software.
- `mod_ssl` stellt die Verbindung zwischen Apache und OpenSSL her. Apache kann damit SSL zur Server- und Client-Authentisierung einsetzen.

Literaturverzeichnis

GW96 Ian Goldberg, David Wagner. »Randomness and the Netscape Browser«. *Dr. Dobb's Journal*, Januar 1996.
http://www.eecs.berkeley.edu/~daw/papers/ddj-netscape.html

MA10 Steven J. Murdoch, Ross Anderson. »Verified by Visa and MasterCard SecureCode: or, How Not To Design Authentication«, Januar 2010.
http://www.cl.cam.ac.uk/~rja14/Papers/fc10vbvsecurecode.pdf

Mar09 Moxie Marlinspike. »Null Prefix Attacks Against SSL/TLS Certificates«, Juli 2009. http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf

RFC2246 T. Dierks, C. Allen. »The TLS Protocol Version 1.0«, Januar 1999.
http://www.ietf.org/rfc/rfc2246.txt

RFC2560 M. Myers, R. Ankney, A. Malpani, et al. »X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP«, Juni 1999.
http://www.ietf.org/rfc/rfc2560.txt

RFC2817 R. Khare, S. Lawrence. »Upgrading to TLS Within HTTP/1.1«, Mai 2000. http://www.ietf.org/rfc/rfc2817.txt

RFC3207 P. Hoffman. »SMTP Service Extension for Secure SMTP over Transport Layer Security«, Februar 2002. http://www.ietf.org/rfc/rfc3207.txt

RFC4346 T. Dierks, E. Rescorla. »The Transport Layer Security (TLS) Protocol Version 1.1«, April 2006. http://www.ietf.org/rfc/rfc4346.txt

RFC4347 E. Rescorla, N. Modadugu. »Datagram Transport Layer Security«, April 2006. http://www.ietf.org/rfc/rfc4347.txt

RFC4366 S. Blake-Wilson, M. Nystrom, D. Hopwood, et al. »Transport Layer Security (TLS) Extensions«, April 2006. http://www.ietf.org/rfc/rfc4366.txt

RFC5246 T. Dierks, E. Rescorla. »The Transport Layer Security (TLS) Protocol Version 1.2«, August 2008. http://www.ietf.org/rfc/rfc5246.txt

VMC02 John Viega, Matt Messier, Pravir Chandra. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly & Associates, 2002. ISBN 0-596-00270-X.
http://www.oreilly.com/catalog/openssl/

LINUP FRONT

3

Zertifikate

Inhalt

Lernziele

- Struktur und Einsatz von OpenSSL-Zertifikaten kennen
- Aufgaben und Tätigkeiten einer Zertifizierungsstelle (CA) kennen
- Eine eigene Zertifizierungsstelle einrichten können

Vorkenntnisse

- Kenntnisse über Kryptographie (Kapitel 1)
- Kenntnisse über SSL und OpenSSL (Kapitel 2)

3.1 Zertifizierung

public-key infrastructure
PKI

In einer Infrastruktur, die auf öffentlichen Schlüsseln beruht (**public-key infrastructure**, **PKI**), müssen Teilnehmer in der Lage sein, die öffentlichen Schlüssel ihrer Kommunikationspartner herauszufinden. Ist die Teilnehmergruppe klein genug, so ist das kein Problem (man kennt sich); ein Teilnehmerkreis dagegen, der potentiell das ganze Internet umfasst, braucht eine Methode, mit deren Hilfe sich die Authentizität der öffentlichen Schlüssel beliebiger Parteien sicherstellen lässt – in anderen Worten, die bestätigt, dass der öffentliche Schlüssel, der für eine Person, einen Web-Server oder ein Unternehmen kursiert, tatsächlich zu der betreffenden Person, dem Web-Server oder dem betreffenden Unternehmen gehört. Wie bereits in Kapitel 1 angedeutet wurde, verwendet man hierzu die Idee

Zertifizierung

der **Zertifizierung**: Eine »vertrauenswürdige« Instanz wird herangezogen, um die Echtheit eines öffentlichen Schlüssels zu bestätigen.

Die genaue Ausgestaltung dieser vertrauenswürdigen Instanz ergibt sich nicht

Vertrauensnetz

zwangsläufig. Das E-Mail-System PGP bedient sich dazu eines **Vertrauensnetzes** (engl. *web of trust*) – jeder Teilnehmer zertifiziert die Schlüssel der Teilnehmer, die er unmittelbar kennt, und andere Teilnehmer übernehmen dieses Vertrauen nach gewissen Regeln von ihm. Solange sich eine »Vertrauenskette« zu einer tatsächlichen Bestätigung der Authentizität herstellen läßt, gilt der Schlüssel als authentisch.

Zertifizierungsstellen

Der von (Open)SSL verfolgte Ansatz dagegen beruht auf »zentralen« **Zertifizierungsstellen** (engl. *certification authorities*, CAs), die ihr Mandat entweder von Staats wegen zugesprochen bekommen oder aufgrund ihrer Geschäftspraktiken und ihres guten Rufs Vertrauen »verdienen«. So gibt es Firmen wie Symantec (die 2010 das Zertifizierungsstellen-Geschäft von Verisign übernommen hat), Thawte (die sich in den 1990er Jahren den PKI-Markt zu etwa gleichen Teilen mit Verisign teilte, bis Thawte 1999 von Verisign übernommen wurde; heute ist Thawte ein Tochterunternehmen von Symantec) oder auch die T-Systems Enterprise Services GmbH, die eine Zertifizierung von öffentlichen Schlüsseln als Dienstleistung anbieten. Grundsätzlich kann jeder OpenSSL-Benutzer als Zertifizierungsstelle tätig werden – die notwendige Software ist frei verfügbar –, allerdings gibt es normalerweise keinen besonderen Grund, warum Sie als Anwender einer beliebigen anderen Person auf dem Internet vertrauen sollten, und in den meisten Ländern können an die Tätigkeit als »echte« Zertifizierungsstelle im Sinne der jeweiligen Gesetzgebung mehr oder weniger strenge Auflagen gekoppelt sein.

Wie auf dem Gebiet der Standardbetriebssysteme für PCs hat auch bei Zertifizierungsstellen Popularität nichts mit Kompetenz zu tun. Insbesondere die Firma VeriSign hat sich in der Vergangenheit diverse extreme Schnitzer geleistet, was ihrer Beliebtheit jedoch keinen Abbruch zu tun scheint.

Zertifikat

Das Resultat eines Zertifizierungsvorgangs nennt man (naheliegenderweise) ein **Zertifikat** – das Zertifikat wird von der Zertifizierungsstelle digital signiert, die damit die Verantwortung dafür übernimmt, dass die Zuordnung zwischen Inhaber und Schlüssel korrekt ist. Das Zertifikat benennt den öffentlichen Schlüssel, um den es geht, und die Identität des Inhabers; daneben enthält es die Identität der Zertifizierungsstelle und deren Signatur, die die Authentizität des Schlüssels bestätigt. Dazu kommen noch Verwaltungsinformationen wie beispielsweise das Datum der Zertifikatserzeugung und die maximale Gültigkeitsdauer des Zertifikats. Jeder, der das Zertifikat in die Hände bekommt, kann sich mit dem öffentlichen Schlüssel der Zertifizierungsstelle davon überzeugen, dass die Signatur auf dem Zertifikat stimmt, und sich anschließend entscheiden, ob die Zertifizierungsstelle als Institution so glaubwürdig ist, dass das Zertifikat als echt angesehen werden sollte. In der Regel machen das die WWW-Browser automatisch für einen, ohne im Detail nachzufragen.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=Test-CA, L=Darmstadt, C=DE/emailAddress=ca@example.com,
            O=Zertifizierungsstelle
        Validity
            Not Before: Oct 27 14:36:58 2003 GMT
            Not After : Oct 26 14:36:58 2005 GMT
        Subject: CN=www.example.com, L=Darmstadt,
            C=DE/emailAddress=webmaster@example.com, O=Beispiel GmbH,
            OU=Web-Server
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:d2:d1:a2:6b:22:4f:54:83:ae:60:6a:8f:37:05:
                    ◁◁◁◁◁
                    68:5b:57:b7:34:6e:f6:05:ba:04:5f:ba:f8:da:e0:
                    87:17:33:e0:7e:c6:54:b7:f3
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
    Signature Algorithm: md5WithRSAEncryption
        69:c4:fa:fb:85:6b:b4:5d:cd:da:e1:c1:51:10:cc:1c:09:bc:
        ◁◁◁◁◁
        7d:e6:59:f0:37:0f:30:fc:fa:d8:2b:c4:77:91:a9:8e:6f:af:
        d7:cd:dc:64
```

Bild 3.1: Ein X.509-Zertifikat

Übungen

3.1 [3] Diskutieren Sie die Vor- und Nachteile des *web of trust* à la PGP gegenüber der zentralisierten Architektur von OpenSSL.

3.2 X.509

Für die Details der Zertifikate richtet SSL sich nach der ITU-T-Empfehlung **X.509**, die einen Rahmen für Dienstleistungen wie Authentisierung unter der zentralen Idee eines »Verzeichnisses« (engl. *directory*) definiert. Bei X.509 handelt es sich allerdings in der Tat nur um eine Empfehlung, nicht um eine genau definierte Norm, so dass sich die tatsächlichen Umsetzungen von Firmen wie Netscape, Microsoft oder RSA im Detail unterscheiden. Beispielsweise kann ein X.509-Zertifikat, das für ein Netscape-Produkt akzeptabel ist, von einem Microsoft-Produkt aufgrund von verschiedenen Auffassungen über die Definition von X.509 abgewiesen werden. Ferner haben X.509-Zertifikate ein codiertes Format, das für menschliche Benutzer nicht sofort lesbar ist – Sie müssen sich also darauf verlassen, dass die Browser daraus das Richtige machen. X.509

X.509-Zertifikate enthalten die folgenden Informationen:

Versionsnummer Gibt an, welche Version des X.509-Standards für dieses Zertifikat gilt (inzwischen gibt es drei verschiedene). Diese Versionsnummer

bestimmt, welche anderen Informationen im Zertifikat zu finden sind.

Seriennummer Der Erzeuger des Zertifikats versieht das Zertifikat mit einer Seriennummer, um es von anderen selbsterzeugten Zertifikaten zu unterscheiden. Diese Seriennummer wird zum Beispiel benötigt, um das Zertifikat notfalls *zurückzurufen*.

Bezeichnung des Herausgebers Der Name der Institution, die das Zertifikat signiert hat (normalerweise eine Zertifizierungsstelle). Um das Zertifikat benutzen zu können, müssen Sie *dieser* Institution vertrauen (der Institution, deren Schlüssel zertifiziert wird, nicht notwendigerweise). Zertifizierungsstellen signieren üblicherweise ihre eigenen Zertifikate selbst.

Bezeichnung des Signaturalgorithmus Die Zertifizierungsstelle gibt hier an, welches Verfahren sie verwendet hat, um das Zertifikat zu signieren.

Gültigkeitszeitraum Ein Zertifikat gilt nur für eine begrenzte Zeit, die durch ein Anfangs- und ein Enddatum beschrieben wird. Wie lang diese Zeit ist, hängt vor allem von der Sicherheit des privaten Schlüssels ab, mit dem das Zertifikat signiert wurde – ist dieser kurz, so ist eher damit zu rechnen, dass er gebrochen wird und alle damit signierten Zertifikate ihren Wert verlieren. Der Gültigkeitszeitraum von Zertifikaten kann von Sekunden bis zu Jahrzehnten reichen.

Name, ausgezeichneter

Zu zertifizierender Name Der Name der Institution oder Person, deren öffentlicher Schlüssel zertifiziert wird. Dieser **ausgezeichnete Name** (engl. *distinguished name*, DN), wird im X.500-Format angegeben und ist eindeutig auf dem Internet.

Zu zertifizierender Schlüssel Der öffentliche Schlüssel der benannten Institution oder Person, zusammen mit einer Bezeichnung für das Kryptosystem, in dem der Schlüssel verwendet wird, und allfälligen weiteren Parametern.

Signatur Die kryptographische Signatur der Zertifizierungsstelle, die die Authentizität des Schlüssels bestätigt.

Darstellung

Diese Informationen werden gemäß ASN.1/DER (*Abstract Syntax Notation 1/Definite Encoding Rules*) codiert und sind damit systemunabhängig beschrieben. Je nach der verwendeten X.509-Version können die Zertifikate noch weitere Informationen enthalten; die heute aktuelle Version 3 zum Beispiel erlaubt die Angabe von **Erweiterungen**, etwa um den Einsatzbereich von Schlüsseln einzuschränken. Bild 3.1 zeigt ein X.509-Zertifikat in lesbarer Form (einige Zeilen wurden umbrochen).

X.509-Erweiterungen

Rein interessehalber: Wenn Sie ein Zertifikat vorliegen haben – etwa in der Datei `zert.pem` –, können Sie eine Ausgabe wie in Bild 3.1 wie folgt erzeugen:

```
$ openssl x509 -in zert.pem -text -noout
```

»PEM« steht für ein Dateiformat, in dem das nach ASN.1/DER dargestellte Zertifikat Base-64-codiert und mit einer Start- und einer Endzeile versehen wird. Das Ganze sieht dann ungefähr so aus:

```
-----BEGIN CERTIFICATE-----
MIIDgTCCAmmgAwIBAgIBATANBgkqhkiG9w0BAQUFADBgMRcwFQYDVQQDEw5MaW51
cCBGcm9udCBDQTEVMBMGA1UECxMMQ0EgKENsaWVudHMpMRowGAYKCZImiZPyLGQB
<<<<< Diverse ähnlich kryptische Zeilen entfernt
2nqdYG/qQsSpe805ikxDU8RIpkRnIcjoDVnHAkDth8Owc/tW2B56RzgyrHd72mS6
17t0SIc+J476xE5BWVC78waXaDhdgSzd6vtqqBirA0yV9gbjnQ==
-----END CERTIFICATE-----
```

Tabelle 3.1: Symantec-Zertifikate 2013 (Auswahl)

Name	Typ	Stärke	Zertifizierungspraxis	Preis
Digital ID	Client	—	Benutzer kann an der angegebenen Adresse Mail empfangen	$22,95/Jahr
Secure Site	Server	40–256 Bit	Organisationsüberprüfung	$399/Jahr
SSL Wildcard	Server	40–256 Bit	Organisationsüberprüfung	$1,999/Jahr
Secure Site Pro	Server	128–256 Bit	Organisationsüberprüfung	$995/Jahr
Secure Site (EV)	Server	40–256 Bit	Erweiterte Validierung	$995/Jahr
Secure Site Pro (EV)	Server	128–256 Bit	Erweiterte Validierung	$1,499/Jahr

Bei den Server-Zertifikaten gibt es Rabatt, wenn man sie für mehr als ein Jahr ausstellen läßt. Bei der »Organisationsüberprüfung« prüft Symantec, ob es Ihre Organisation gibt, ob Sie das DNS der Organisation kontrollieren und ob Sie überhaupt die Autorität haben, ein Zertifikat beantragen zu dürfen. Zum Thema »erweiterte Validierung« steht mehr auf Seite 42.

Die »ausgezeichneten Namen« oder DNs in Zertifikaten folgen dem X.500-Standard und sind damit hierarchisch aufgebaut. Ein DN wie DNs

```
CN=Hugo Schulz, OU=Entwicklung, O=Beispiel GmbH,
  L=Irgendwo, ST=Südrhein-Ostfalen C=DE
```

beschreibt eine Person namens (`CN`, *common name*) Hugo Schulz, die Mitglied der Abteilung (`OU`, *organizational unit*) »Entwicklung« der Organisation (`O`) »Beispiel GmbH« ist, die am Ort (`L`, *location*) »Irgendwo« im Bundesland (`ST`, *state*) »Südrhein-Ostfalen« des Staats (`C`, *country*) Deutschland (`DE`) angesiedelt ist. (X.500 ist eine umfangreiche und an vielen Stellen nur vage festgeschriebene Norm, für deren genaue Diskussion hier leider kein Platz ist.)

Übungen

3.2 [1] Geben Sie einen ausgezeichneten Namen (DN) an, der Sie auf der Basis Ihrer Organisationszugehörigkeit möglichst eindeutig beschreibt.

3.3 [2] Welchen Sinn könnte es haben, dass Zertifikate nicht nur ein Enddatum der Gültigkeit enthalten, sondern auch ein Anfangsdatum?

3.3 Zertifizierungsstellen

Eine Zertifizierungsstelle (CA) ist einfach eine Person oder Institution, die Zertifikate ausstellt. Wie schon erwähnt wurde, brauchen Sie dafür keine besonderen Privilegien, eine Kopie z. B. der OpenSSL-Software reicht aus. Es ist der individuellen Zertifizierungsstelle überlassen, sich Standards dafür aufzuerlegen, wann sie welchen Schlüssel zertifiziert; eine Firma, die eine Zertifizierungsstelle betreibt, würde zum Beispiel bestätigen, dass es sich beim Inhaber eines Zertifikats um einen Angestellten der Firma handelt. Eine kommerzielle Zertifizierungsstelle wie zum Beispiel Symantec bietet normalerweise mehrere Stufen der Zertifizierung zu unterschiedlichen Preisen an (Tabelle 3.1).

Sie können diverse Arten von Zertifizierungsstellen unterscheiden [GS02]:

Interne CA Eine Organisation zertifiziert ihre eigenen Mitglieder (z. B. Name, Position, …). Diese internen Zertifikate könnten für den Zugang zu organisationseigenen Informationsquellen benutzt werden oder der Authentisierung von Kunden gegenüber organisationseigenen Ressourcen dienen. Beispielsweise könnte ein Online-Wertpapierverwaltungssystem verlangen, dass Benutzer ein Zertifikat präsentieren, bevor sie Transaktionen über einem gewissen Geldbetrag durchführen dürfen.

Ausgegliederte CA für Angestellte Eine Organisation, die Zertifikate verwenden, aber selbst nicht als CA tätig werden möchte, könnte diese Dienstleistung ausgliedern.

Ausgegliederte CA für Kunden Analog hierzu könnte eine Organisation die Zertifizierung von Kunden extern vergeben, solange sie der externen CA vertraut.

Vertrauenswürdige Dritt-CA Eine Firma oder Regierung kann eine Zertifizierungsstelle betreiben, die öffentliche Schlüssel mit den gesetzlichen Namen von Personen oder Firmen verknüpft. Zertifikate einer solchen Zertifizierungsstelle können Sie als analog zu offiziellen Identitätsdokumenten wie Personalausweisen ansehen.

Um die von einer bestimmten Zertifizierungsstelle ausgestellten Zertifikate benutzen zu können, brauchen Sie den öffentlichen Schlüssel der Zertifizierungsstelle, damit Sie die Authentizität der digitalen Signatur auf den Zertifikaten überprüfen können. Zertifizierungsstellen verbreiten ihre eigenen öffentlichen Schlüssel über Zertifikate, die sie selbst signiert haben; das ist natürlich nicht der Gipfel der Sicherheit, aber irgendwem müssen Sie irgendwann vertrauen. Tatsächlich enthalten die gängigen WWW-Browser schon »ab Werk« Zertifikate für die wichtigsten CAs. Es ist normalerweise möglich, weitere Zertifikate zu installieren.

Selbstsignierte Zertifikate

Eine Zertifizierungsstelle muss festlegen, wie sie mit Zertifikaten umgeht. Diese Zertifizierungspraxis-Beschreibung (engl. *certification practices statement*, CPS) gibt an, unter welchen Bedingungen und wie eine Zertifizierungsstelle Zertifikate vergibt oder zurückruft – sie beschreibt die »Bedeutung« eines Zertifikats dieser Zertifizierungsstelle. Beim CPS handelt es sich um ein für Menschen geschriebenes Dokument, das ein wichtiges Kriterium zur Auswahl einer CA darstellt; es könnte neben den Zertifizierungskriterien zum Beispiel auch eine Erklärung enthalten, bis zu einem gewissen Betrag für Zertifizierungsfehler zu haften.

Zertifizierungspraxis-Beschreibung

Die neueste Mode auf dem Gebiet der Zertifikate sind sogenannte *Extended-Validation*- oder EV-Zertifikate. Das »CA/Browser Forum« (eine Industrievereinigung der wichtigsten Zertifizierungsstellen und Browser-Hersteller) hat sich darauf geeinigt, bestimmte Zertifikate nur nach besonders strengen Regeln zu vergeben und durch solche Zertifikate ausgewiesene Server dann im Browser besonders kenntlich zu machen (typischerweise durch ein grün hinterlegtes URL-Feld; mit »normalen« Zertifikaten versehene Server zeigen an derselben Stelle zum Beispiel Blau). Zertifizierungsstellen, die EV-Zertifikate ausgeben wollen, müssen sich einer jährlichen Prüfung (engl. *audit*) unterziehen. Das CA/Browser Forum veröffentlicht sowohl die Anforderungen für die Erteilung von EV-Zertifikaten als auch die Anforderungen für die CA-Prüfung.

Aus der Sicht eines Web-Server-Betreibers liegt die Latte für die Erteilung eines EV-Zertifikats nicht nennenswert höher, als sie sowieso liegen sollte: Die Anforderungen sehen im Wesentlichen vor, dass eine CA überprüfen muss, dass die antragstellende Organisation an der behaupteten Adresse existiert, geschäftlich tätig und ordentlich als »aktiv« im Handelsregister oder einem vergleichbaren amtlichen Verzeichnis eingetragen ist. Es gibt Ausnahmeklauseln für Regierungsorganisationen und andere Organisationen, für die die angegebenen Kriterien keinen Sinn ergeben.

Als sichtbaren Unterschied für Benutzer (außer dem freundlichen Grünton) enthalten die Zertifikate neben dem üblichen Namen der zertifizierten Organisation und dem Rechnernamen noch die Adresse des Firmensitzes (nichts Besonders) und die Angabe des Orts, wo die Organisation zum Beispiel ins Handelsregister eingetragen ist. Dazu kommt, falls vorhanden, noch die Nummer des Registereintrags.

Im Gegensatz zu »normalen« Zertifikaten können Sie EV-Zertifikate nicht mit den üblichen Mitteln (etwa OpenSSL) selber erzeugen. Browser erkennen gültige EV-Zertifikate daran, dass diese einen CA-spezifischen Code enthalten[1], den der Browser mit einer fest eingebauten Liste von CAs und Codes vergleichen kann. (Nicht jeder Browser hat dieselbe Liste, ähnlich wie auch bei CAs.) Damit Sie selber Zertifikate generieren können, die wie EV-Zertifikate behandelt werden, müssen Sie also nicht bloß einen geeigneten Code ins Zertifikat schreiben, sondern auch den Browser-Quellcode anpassen, damit der Browser Ihre Zertifikate als »EV-Zertifikat« erkennt. Das funktioniert natürlich nur, wenn Ihre Benutzer dann auch »Ihren« Browser einsetzen.

Über den Sinn und Unsinn von EV-Zertifikaten kann man eine Weile philosophieren. Eigentlich sollte es sich von selbst verstehen, dass die CAs die gründlichen Maßstäbe, die für EV-Zertifikate gelten, auf *alle* ihre Zertifikate anwenden. Alles andere ist letzten Endes nichts anderes als ein Eingeständnis ihrer eigenen Bequemlichkeit und/oder ihres Unvermögens, ihre Arbeit vernünftig zu machen. Ein Blick in die Preisliste von Symantec zeigt, dass ein »gewöhnliches« Zertifikat für $399/Jahr über den Ladentisch geht (Stand: Februar 2013), während für ein entsprechendes EV-Zertifikat schon $995/Jahr oder mehr zu berappen sind. Gleichzeitig sagen die Zertifizierungsstellen mehr oder weniger durch die Blume, dass nur die teuren EV-Zertifikate die Mühe wert sind, weil »die Benutzer« ja sonst nicht erkennen können, dass der betreffende Web-Server wirklich »sicher« ist. – EV-Zertifikate sind also vor allem eine geschickte Methode der Zertifizierungsstellen, um *de facto* die Preise mal eben auf das Dreifache (oder so) zu erhöhen, denn wer möchte sich als respektabler Online-Händler schon mit einem Zertifikat sehen lassen, über das schon der *Hersteller* sagt, dass es nicht mehr das tut, wofür es mal gedacht war?[2] Der letztendliche Sicherheitsgewinn dürfte am Ende des Tages minimal sein, da der durchschnittliche Benutzer sowieso nicht weiß, was sich hinter einem Zertifikat (geschweige denn einem EV-Zertifikat) verbirgt und das auch nicht wirklich wissen möchte. Denn um sicherzugehen, müßte er ja einerseits – egal ob für eines normales oder EV-Zertifikat – das CPS der ausstellenden Zertifizierungsstelle finden und lesen und andererseits überprüfen, ob er wirklich deren korrekten öffentlichen Schlüssel hat. Und das macht bekanntlich sowieso niemand.

Übungen

3.4 [2] Besorgen Sie sich ein CPS einer Zertifizierungsstelle Ihres Vertrauens (?) und studieren Sie es.

3.4 Erzeugung von Zertifikaten

Die OpenSSL-Software macht es möglich, diverse Arten von Zertifikaten zu erzeugen. Neben Zertifikaten zur Identifizierung von WWW-Servern und -clients ist es auch möglich, Zertifikate zu erstellen, die zur Zertifizierung anderer Zertifikate taugen. Ein solches Zertifikat dient als Basis einer eigenen Zertifizierungsstelle. Eine Zertifizierungsstelle bekommt **Zertifizierungsanfragen** (engl. *certificate signing requests*, **CSRs**), prüft diese und erteilt auf ihrer Basis Zertifikate, indem sie die CSRs digital signiert.

Zertifizierungsanfragen
CSRs

Ausgangspunkt für alle weiteren Experimente mit OpenSSL ist die Einrich-

Einrichtung einer eigenen Zertifizierungsstelle

[1] Ganz präzise gesagt im *certificate policies extension field*. Eine Liste finden Sie zum Beispiel unter `http://en.wikipedia.org/wiki/Extended_Validation_Certificate`.

[2] Stellen Sie sich vor, Ihr Metzger sagt Ihnen etwas wie »Nehmen Sie nicht mein Schweinenackensteak, das ist völlig vergammelt und eklig. Hier habe ich doch dieses ganz hervorragende Rinderfilet für Sie, praktisch geschenkt für das Dreifache!«

tung einer eigenen Zertifizierungsstelle. Theoretisch wäre es möglich, mit einem »offiziellen« Zertifikat von einer Firma wie Symantec anzufangen, aber sinnvollerweise experimentieren Sie erst einmal in eigener Regie und holen die bürokratischen Teile dann bei Bedarf nach. Außerdem ist ein offizielles Zertifikat vielleicht überhaupt nicht nötig. – Eine ernstgemeinte private (etwa unternehmensweite) Zertifizierungsstelle sollten Sie übrigens auf einem Rechner einrichten, der nicht anderweitig verwendet wird. Am Netz muss bzw. sollte er auch nicht sein; am besten benutzen Sie einen Notebook-Rechner, der bei Nichtgebrauch in einem Safe eingeschlossen wird.

Weitere Voraussetzung ist, dass die OpenSSL-Software installiert wurde. Sinnvollerweise legen Sie für die Zertifizierungsstelle ein eigenes Verzeichnis – etwa `/usr/local/openssl/testCA` – an. Dieses Verzeichnis sollte Unterverzeichnisse `certs` und `private` haben, und im wirklichen Leben müssen Sie darauf achten, dass niemand Unbefugtes schreibend auf `certs` zugreifen kann, da es ihm sonst möglich ist, dort »trojanische« Zertifikate zu installieren, die anschließend zum Signieren von anderen Zertifikaten benutzt werden. Das Verzeichnis `private` sollte für überhaupt niemanden außer dem Eigentümer zugänglich sein, da dort die privaten Schlüssel der Zertifizierungsstelle abgelegt werden. Zur Initialisierung der Zertifizierungsstelle bietet sich eine Befehlsfolge wie diese an:

```
$ echo $USER
hugo
$ /bin/su -
Password: xxx123                                   Oder was auch immer …
# mkdir -p /usr/local/openssl/testCA
# chown hugo:users /usr/local/openssl/testCA
# exit
$ cd /usr/local/openssl/testCA
$ mkdir private certs
$ chmod 700 private certs
$ echo 01 >serial
$ touch index.txt
```

Zertifikatsverwaltung

Die Dateien `serial` und `index.txt` werden zur Zertifikatsverwaltung durch OpenSSL benötigt. `serial` enthält die »Seriennummer« des nächsten zu erzeugenden Zertifikats, und `index.txt` verzeichnet die bisher erzeugten Zertifikate zu Archivzwecken und zur Generierung von CRLs.

Konfigurationsdatei für OpenSSL

Der nächste Schritt ist das Anlegen einer Konfigurationsdatei für OpenSSL. Im Prinzip könnten Sie auch von den Standardvorgaben ausgehen, aber es ist mehr als wahrscheinlich, dass Sie, spätestens wenn Sie selbst zum Beispiel Client-Zertifikate erzeugen wollen, sowieso von diesen Vorgaben abweichen möchten. Also können wir das auch gleich richtig erklären. Die Konfigurationsdatei dient im besonderen dazu, dass die OpenSSL-Kommandozeilenwerkzeuge daraus Informationen beziehen, die sie für die Zertifikatserstellung brauchen. Durch geeignete Konfiguration sparen wir uns hier später einige Tipp- (und Merk-)Arbeit.

Beispiel für die Konfiguration

Bild 3.2 zeigt ein Beispiel für die Konfiguration: Die Datei ist in Abschnitte eingeteilt, die jeweils mit einem Schlüsselwort in eckigen Klammern beginnen. Der erste Abschnitt, `[ca]`, enthält Vorgaben für das OpenSSL-Kommandozeilenprogramm `ca`, das sich (wer hätte es gedacht?) mit den Funktionen einer Zertifizierungsstelle befasst. Die einzige darin vorhandene Direktive, `default_ca`, gibt den Namen eines Abschnitts in der Konfigurationsdatei an, der weitere Details über die Zertifizierungsstelle enthält.

Es ist durchaus möglich, in derselben Konfigurationsdatei Vorgaben für diverse Zertifizierungsstellen zu machen, die dann natürlich in verschiedenen Abschnitten stehen müssen.

Angaben über die Zertifizierungsstelle

Der zweite Abschnitt, `[testCA]`, enthält entsprechend Angaben über die Zerti-

```
[ca]
default_ca = testCA

[testCA]
dir              = /usr/local/openssl/testCA
certificate      = $dir/cacert.pem
database         = $dir/index.txt
new_certs_dir    = $dir/certs
private_key      = $dir/private/cakey.pem
serial           = $dir/serial

default_crl_days = 7
default_days     = 730
default_md       = sha1

policy           = testCA_policy
x509_extensions  = certificate_extensions

[testCA_policy]
commonName             = supplied
localityName           = supplied
countryName            = supplied
emailAddress           = supplied
organizationName       = supplied
organizationalUnitName = optional

[certificate_extensions]
basicConstraints = CA:false

[req]
default_bits       = 2048
default_keyfile    = /usr/local/openssl/testCA/private/cakey.pem
default_md         = sha1
prompt             = no
distinguished_name = testCA_dn
x509_extensions    = testCA_extensions

[testCA_dn]
commonName         = Test-CA
localityName       = Darmstadt
countryName        = DE
emailAddress       = ca@example.com
organizationName   = Zertifizierungsstelle

[testCA_extensions]
basicConstraints   = CA:true
```

Bild 3.2: Eine Beispiel-Konfigurationsdatei für OpenSSL

fizierungsstelle. Die Verzeichnisse und Dateien, über die dieser Abschnitt redet, haben wir größtenteils gerade eben angelegt. Die anderen drei Direktiven, `default_crl_days`, `default_days` und `default_md`, geben respektive den Zeitabstand an, in dem CRLs veröffentlicht werden (wird in die CRL geschrieben), die Gültigkeitsdauer des Zertifikats und das kryptographische Hash-Verfahren für digitale Signaturen in Zertifikaten. Alle diese Parameter können beim Aufruf des `openssl`-Programms durch Kommandozeilenoptionen überschrieben werden.

Anforderungen an Zertifikate

Die `policy`-Direktive gibt den Namen des Abschnitts in der Datei an, der die Anforderungen an Zertifikate beschreibt. Hier können Sie bestimmen, welche Felder ein *distinguished name* in einem CSR enthalten sein müssen (oder dürfen), damit er akzeptabel ist. Für jedes Feld (bzw. jede Direktive) gibt es drei erlaubte Werte: `match`, `supplied` oder `optional`. Der Wert `match` bedeutet, dass der Wert des betreffenden Feldes mit dem des gleichnamigen Felds im Zertifikat der Zertifizierungsstelle übereinstimmen muss. Als `supplied` gekennzeichnete Felder müssen im CSR vorhanden sein, während `optional`-Felder nicht notwendigerweise vorhanden sein müssen.

Erweiterungen

X.509-Zertifikate können außerdem auch Erweiterungen enthalten. Die Direktive `x509_extensions` gibt an, in welchem Abschnitt diese näher beschrieben sind (hier `[certificate_extensions]`). Die einzige Erweiterung, die wir hier angeben, ist `CA:false` – sie bedeutet, dass die von unserer Zertifizierungsstelle erzeugten Zertifikate nicht verwendet werden können, um wiederum neue Zertifikate zu zertifizieren.

Für Zertifizierungsstellen ist das ein probates Mittel, um Autorität an »Unterzertifizierungsstellen« zu delegieren. Eine Prüfung des Zertifikats muss dann die Authentizität der Zertifikate bis hin zu einem als vertrauenswürdig bekannten CA-Zertifikat verfolgen – das Server-Zertifikat muss von einer Sub-CA signiert sein, deren Zertifikat wiederum von einer CA signiert ist und so weiter, bis Sie bei einem Zertifikat ankommen, das von der tatsächlichen CA signiert wurde. Anfang 2003 wurde ein Implementierungsfehler in diversen populären Browsern (darunter Konqueror und Internet Explorer) publik, der dazu führte, dass diese Erweiterung nicht ausgewertet, sondern *alle* Zertifikate als vertrauenswürdig zum Signieren weiterer Zertifikate angesehen wurden, so dass jeder Inhaber eines VeriSign-Zertifikats prinzipiell selbst Zertifikate machen konnte, die als von VeriSign signiert galten …

OpenSSL verwendet normalerweise eine systemweite Konfigurationsdatei in `/etc/openssl/openssl.cnf` (oder so ähnlich). Wir können bequem unsere eigene Konfigurationsdatei bekannt machen, indem wir ihren kompletten Pfadnamen in die Umgebungsvariable `OPENSSL_CONF` eintragen:

`OPENSSL_CONF`

```
$ export OPENSSL_CONF=/usr/local/openssl/testCA/openssl.cnf
```

Zertifikat für die Zertifizierungsstelle

Das Zertifikat für die Zertifizierungsstelle generieren wir mit dem Kommando `req` des `openssl`-Programms. Da wir es nur hierfür verwenden, können wir die Parameter für den Aufruf fest in der Konfigurationsdatei vorschreiben; das hat nicht nur den Vorteil, dass wir sie nicht auf der Kommandozeile angeben müssen, sondern dient außerdem als Dokumentation, wie das Zertifikat angelegt wurde. Ferner ist dies die einzige Möglichkeit, ein X.509v3-Zertifikat mit Optionen zu erhalten. Die entsprechende Konfiguration befindet sich im `[req]`-Abschnitt: `default_bits` gibt an, dass ein 2048 Bit langer privater Schlüssel erzeugt werden soll – Standard wären 512 Bit, und das ist heutzutage zu unsicher, erst recht für ein Schlüsselpaar, das einer Zertifizierungsstelle zugrunde liegen soll! Die zusätzliche Sicherheit ist den nötigen Rechenaufwand bestimmt wert. Der Schlüssel wird in die in `default_keyfile` benannte Datei geschrieben (die `$dir`-Variable ist leider nur im `ca`-Abschnitt sichtbar) und eine mit dem in `default_md` angegebenen Hash-Verfahren bestimmte Prüfsumme signiert. Hier sollten Sie »sha1« eintragen; das früher verbreitete Verfahren MD5 gilt heute als unsicher. `prompt` bekommt den

Wert `no`, damit `openssl` nicht nach den Parametern des DN fragt; sie sind in dem in `distinguished_name` benannten Abschnitt `testCA_dn` angegeben. (Das optionale Feld haben wir uns gespart.) Zum Schluss geben wir per X.509-Option an, dass *dieses* Zertifikat in der Tat zum Signieren anderer Zertifikate zu gebrauchen sein soll.

Das Zertifikat für die Zertifizierungsstelle lässt sich dann leicht mit

```
$ openssl req -x509 -days 1825 -newkey rsa:2048 -out cacert.pem
Generating a 2048 bit RSA private key
.....................................+++
.....................................+++
writing new private key to '/usr/local/openssl/testCA/private/cakey.pem'
Enter PEM pass phrase:blafasel
Verifying - Enter PEM pass phrase:blafasel
$ _
```

erzeugen. Auf die resultierende Datei `cakey.pem` sollten Sie sehr gut aufpassen – geht sie verloren, steht der private Schlüssel für die Zertifizierungsstelle nicht mehr zur Verfügung, und das ist der »Super-GAU«: Sie können keine neuen Zertifikate mehr beglaubigen und die existierenden nicht mehr zurückrufen (was fast schlimmer ist). Wenn Sie ein neues Schlüsselpaar generieren, müssen alle Server und vor allem Browser, die Zertifikate der Zertifizierungsstelle akzeptieren sollen, den neuen öffentlichen Schlüssel mitgeteilt bekommen, und das ist ein größeres Ärgernis.

Die *pass phrase* dient zur Freischaltung des privaten Schlüssels der Zertifizierungsstelle vor der Zertifizierung anderer Schlüssel. Es ist möglich, auf eine *pass phrase* für den Schlüssel zu verzichten, allerdings bedeutet das, dass jeder, der irgendwie an den privaten Schlüssel gekommen ist, neue Zertifikate im Namen der Zertifizierungsstelle beglaubigen kann (noch ein »Super-GAU«). Sie sollten das also tunlichst sein lassen. Wird der private Schlüssel kompromittiert, ist damit die komplette Zertifizierungsstelle kompromittiert; Sie können also weder den in der Vergangenheit ausgestellten noch allfälligen in der Zukunft ausgestellt werdenden Zertifikaten noch trauen.

Die `-days`-Option gibt an, wie lang das Zertifikat der Zertifizierungsstelle gilt. Da mit dem Ablauf dieses Zertifikats die von der Zertifizierungsstelle ausgestellten Zertifikate nicht mehr verifiziert werden können, sollte die Gültigkeitsdauer mit Bedacht gewählt werden: Eine zu kurze Gültigkeitsdauer zwingt Sie dazu, schon bald alle ausgestellten Zertifikate erneuern zu müssen, während eine zu lange Gültigkeitsdauer im schlimmsten Fall bedeuten kann, dass die Zertifizierungsstelle kompromittiert wird, weil Fortschritte in der Computertechnik es möglich gemacht haben, den öffentlichen Schlüssel zu brechen. Ein Zertifikat mit einem 2048-Bit-Schlüssel sollte allerdings für die nächsten 10 Jahre (oder so) sicher sein.

Zertifikat anschauen

Das erzeugte Zertifikat können Sie sich mit einem Kommando wie

```
$ openssl x509 -in cacert.pem -text -noout
```

anschauen. Die Ausgabe sollte so ähnlich aussehen wie Bild 3.3 (die Schlüssel sind natürlich mit großer Sicherheit anders). Beachten Sie, dass »Issuer« und »Subject« denselben DN enthalten – die Zertifizierungsstelle beteuert die Korrektheit des eigenen öffentlichen Schlüssels. Ferner weisen die »X.509 Basic Constraints« dieses Zertifikat als Zertifikat einer Zertifizierungsstelle aus.

Herzlichen Glückwunsch: Wenn Sie bis hierhin mitgearbeitet haben, sind Sie nun stolzer Besitzer einer X.509-Zertifizierungsstelle. Im nächsten Kapitel erklären wir Ihnen, wie Sie mit Ihrer Zertifizierungsstelle Zertifikate zum Beispiel für WWW-Server generieren können.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            a2:48:2a:d9:11:19:29:0c
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: CN=Test-CA, L=Darmstadt, C=DE/emailAddress=ca@example.com,
            O=Zertifizierungsstelle
        Validity
            Not Before: Jan 30 23:42:26 2013 GMT
            Not After : Jan 29 23:42:26 2018 GMT
        Subject: CN=Test-CA, L=Darmstadt, C=DE/emailAddress=ca@example.com,
            O=Zertifizierungsstelle
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:ad:8d:7b:25:bf:5d:55:67:70:f1:96:3d:9b:36:
                    ◁◁◁◁
                    04:2b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha1WithRSAEncryption
        a3:06:1a:03:e4:27:58:e4:a4:12:ff:e3:9b:ae:bf:97:e9:52:
        ◁◁◁◁
        3c:1c:e5:6a
```

Bild 3.3: Ein selbstsigniertes Zertifikat für eine Zertifizierungsstelle

Übungen

3.5 [2] Wie können Sie sich von der Authentizität des selbstsignierten Zertifikats einer Zertifizierungsstelle überzeugen?

Zusammenfassung

- Ein Zertifikat stellt den Zusammenhang zwischen einer Person, einer Organisation oder einem Server und einem öffentlichen Schlüssel her.
- Zertifikate werden von Zertifizierungsstellen (CA) herausgegeben. Die Vertrauenswürdigkeit eines Zertifikats entspricht der Vertrauenswürdigkeit der Zertifizierungsstelle.
- Zertifikate für OpenSSL folgen dem X.509-Standard. Die Namen von Personen, Organisatoren oder Servern sind als »ausgezeichnete Namen« (*distinguished names*) im Sinne von X.500 angegeben.
- Man kann verschiedene Arten von Zertifizierungsstellen unterscheiden: interne und ausgegliederte CAs sowie vertrauenswürdige Dritt-CAs.
- Zertifizierungsstellen signieren auch Zertifikate mit ihrem eigenen öffentlichen Schlüssel, die dann zur Authentizitätsprüfung anderer Zertifikate verwendet werden. WWW-Browser enthalten oft eine Anzahl solcher Zertifikate für die gängigsten Zertifizierungsstellen.
- Um eine Zertifizierungsstelle einrichten zu können, benötigt man ein entsprechendes selbstsigniertes Zertifikat. Dieses erhält man mit dem Kommando »`openssl req`«.

Literaturverzeichnis

GS02 Simson Garfinkel, Gene Spafford. *Web Security, Privacy & Commerce*. Sebastopol, CA: O'Reilly & Associates, 2002, 2. Auflage. ISBN 0-596-00045-6.
`http://www.oreilly.com/catalog/websec2/`

VMC02 John Viega, Matt Messier, Pravir Chandra. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly & Associates, 2002. ISBN 0-596-00270-X.
`http://www.oreilly.com/catalog/openssl/`

4

Server-Authentisierung

Inhalt

Lernziele

- Server-Zertifikate erstellen und installieren können
- Schritte zum Zugriff auf authentisierte Server verstehen und bewerten können
- Zertifikate für Zertifizierungsstellen (CAs) im Browser installieren können

Vorkenntnisse

- Kenntnisse über OpenSSL und Zertifikate (Kapitel 3)
- Kenntnisse über Konfiguration und Betrieb eines Apache-Servers
- Kenntnisse über Bedienung eines WWW-Browsers

4.1 Einführung

Mit OpenSSL können beide Kommunikationspartner – Client und Server – einer TCP-Verbindung authentisiert werden. Die für das World-Wide Web bei weitem wesentlichere Anwendung ist die Authentisierung von Servern: Man möchte den Benutzern das Gefühl geben, dass sie ihre Online-Bestellungen und persönlichen Daten wie Adresse und Kreditkarteninformationen tatsächlich an das Versandhaus *X* schicken und nicht an den Konkurrenten *Y* oder irgendeinen kriminellen *Z*. Die umgekehrte Richtung – Authentisierung von Benutzern gegenüber dem Server – ist für die meisten WWW-Anwendungen viel zu aufwendig; das finanzielle Risiko, dem einen oder anderen »Irrläufer« aufzusitzen, ist wesentlich eher tragbar als der Umsatzverlust, der daraus entsteht, dass Kunden sich aufwendig SSL-Zertifikate verschaffen und diese in ihrem Browser installieren müssen und dann vielleicht doch entnervt zur Konkurrenz wechseln. Aus diesem Grund setzen Anbieter in der Regel einfachere Authentisierungsmethoden ein (etwa über Benutzernamen und Kennwörter), die für die Kunden mit weniger Umstand verbunden sind, selbst wenn nicht die optimale Sicherheit erreicht wird. Für ein Versandhaus ist das kein fundamentales Problem, denn die herkömmlichen Bestellungen – über (gelbe) Post oder Telefon – sind ja auch nicht besonders authentisiert. Die Infrastruktur, um mit fehlgeleiteten Bestellungen oder säumigen oder zahlungsunwilligen Kunden umzugehen, muss also ohnehin vorhanden sein.

Authentisierung von Servern

Im Grunde ist es sehr einfach, einen Apache-Server für Authentisierung über SSL zu konfigurieren. Sie müssen sich nur ein Server-Zertifikat verschaffen und dieses an die richtige Stelle im Dateisystem kopieren, wo der Server es beim Start finden kann. Die meisten Distributionen enthalten die notwendige Konfiguration und unter Umständen schon ein Test-Zertifikat zum Ausprobieren. Mit diesem Test-Zertifikat ist im wirklichen Leben natürlich kein Staat zu machen, so dass Sie es durch ein »echtes« ersetzen müssen, sobald Sie sich überzeugt haben, dass der Server mit SSL funktioniert.

Server-Zertifikat

Der Haken an der Sache ist eher, dass Sie für beste Ergebnisse ein Zertifikat brauchen, das von einer wohlbekannten Zertifizierungsstelle beglaubigt ist – »wohlbekannt« in dem Sinne, dass die gängigen Browser das Zertifikat der Zertifizierungsstelle enthalten. Wenn Sie also einen Apache-Server von allgemeinem Interesse ans Netz bringen wollen, kommen Sie kaum daran vorbei, die Dienste einer Firma wie Symantec in Anspruch zu nehmen. (Sie können natürlich auch versuchen, den Benutzern Ihrer Seiten das Zertifikat Ihrer eigenen Zertifizierungsstelle zukommen zu lassen; die Installation eines solchen Zertifikats in den Browser ist jedoch möglicherweise vom durchschnittlichen Anwender etwas viel verlangt – insbesondere wenn Sie diesen Anwendern auch noch einbläuen müssen, das nicht blindlings zu tun, sondern sich vorher bei Ihnen von der Authentizität dieses Zertifikats zu überzeugen.)

Für gewisse Anwendungen – etwa die, wo Sie es nur mit einem geschlossenen, überschaubaren Benutzerkreis zu tun haben –, ist es absolut möglich und sinnvoll, selbst eine Zertifizierungsstelle zu betreiben. Im vorigen Kapitel haben Sie ja gesehen, wie Sie eine einrichten können. Dies gilt insbesondere, wenn Sie auch die Benutzer über Zertifikate authentisieren wollen, da Sie durch *do-it-yourself* eine Vertrauensstufe erreichen können, die Sie über ein Symantec-Zertifikat nie schaffen, und dabei auch noch einen Haufen Geld sparen können. In diesem Fall müssen Sie natürlich dafür sorgen, dass den Benutzern das Zertifikat Ihrer Zertifizierungsstelle zur Verfügung steht. Eine Möglichkeit dafür ist, ihnen einen Browser zu geben, der das Zertifikat bereits installiert hat.

4.2 Erzeugen eines Server-Zertifikats

Die Erzeugung eines Server-Zertifikats für einen Apache-Server ist ein zweistufiger Prozess:

1. Sie als Inhaber des Servers erzeugen einen *Certificate Signing Request* (**CSR**). Dies ist eine Datei, die Sie an eine Zertifizierungsstelle Ihres Vertrauens schicken können, und die Ihren öffentlichen Schlüssel und den DN Ihres Servers enthält. CSR

2. Die Zertifizierungsstelle überzeugt sich (wie auch immer) davon, dass die Informationen in Ihrem CSR vernünftig sind, und stellt auf dessen Basis ein – von der Zertifizierungsstelle signiertes – Zertifikat aus, das sie Ihnen zurückschickt.

Dieses Zertifikat können Sie Ihrem Server zugänglich machen, der es dann bei entsprechenden Anfragen an Browser schickt. Die Browser können die Authentizität des Zertifikats prüfen, indem sie anhand des öffentlichen Schlüssels der Zertifizierungsstelle, den sie dem bei ihnen lokal vorliegenden Zertifikat derselben entnehmen können, die Gültigkeit der Signatur der Zertifizierungsstelle auf Ihrem Zertifikat nachvollziehen. Gelingt dies, so hat der Browser den »echten« öffentlichen Schlüssel Ihres Servers zur Verfügung und kann damit die Authentizität Ihres Servers selbst prüfen, indem er, wie schon beschrieben, eine Zufallszahl verschlüsselt und an Ihren Server schickt. Ihr Server antwortet mit der korrekt entschlüsselten Zahl und hat damit gezeigt, dass er den passenden privaten Schlüssel zu dem öffentlichen Schlüssel aus seinem Zertifikat besitzt. Den hat aber nur der »echte« Server – Q. E. D.

Betrachten wir also zunächst, wie Sie einen CSR erzeugen können. Die Vorgehensweise ist der zur Erzeugung des Zertifizierungsstellen-Zertifikats nicht unähnlich – auch hier verwenden wir das OpenSSL-`req`-Kommando, allerdings mit einigen zusätzlichen Parametern, aber wir geben die Daten für den DN diesmal interaktiv ein. (Achten Sie darauf, diesen Schritt in einer Shell auszuführen, in der die `OPENSSL_CONF`-Variable undefiniert ist. Die Konfiguration aus dem vorigen Kapitel ist nur für die Erzeugung des Zertifikats der Zertifizierungsstelle gedacht.) CSR-Erzeugung

```
$ openssl req -newkey rsa:2048 -keyout server.key \
>    -out server.csr
Generating a 2048 bit RSA private key
..++++++
.........++++++
writing new private key to 'server.key'
Enter PEM pass phrase:123456
Verifying - Enter PEM pass phrase:123456
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:Darmstadt
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Beispiel GmbH
Organizational Unit Name (eg, section) []:Web-Server
Common Name (eg, YOUR name) []:www.example.com
Email Address []:webmaster@example.com

Please enter the following 'extra' attributes
```

```
$ openssl req -in server.csr -text -noout
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=DE, L=Darmstadt, O=Beispiel GmbH, OU=Web-Server,
          CN=www.example.com/emailAddress=webmaster@example.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:d2:d1:a2:6b:22:4f:54:83:ae:60:6a:8f:37:05:
                    ◁◁◁◁◁
                    87:17:33:e0:7e:c6:54:b7:f3
                Exponent: 65537 (0x10001)
        Attributes:
            unstructuredName         :Beispiel GmbH
            challengePassword        :bla blubb fasel
    Signature Algorithm: sha1WithRSAEncryption
        12:2f:25:cb:cc:3b:ff:1f:71:fe:a6:3e:ec:7b:72:4d:4d:7a:
        ◁◁◁◁◁
        6b:ca:99:c7:0b:09:13:60:3c:91:27:65:a8:8a:cf:95:53:51:
        b6:25
```

Bild 4.1: Der Certificate Signing Request (CSR)

```
to be sent with your certificate request
A challenge password []:bla blubb fasel
An optional company name []:Beispiel GmbH
$ _
```

Was Sie als DN-Bestandteile eingeben, ist im Großen und Ganzen ziemlich egal – es muss halt der Zertifizierungsstelle glaubwürdig vorkommen. Der einzige DN-Bestandteil, für den das nicht gilt, ist der *common name* (hier `www.example.com`): Dieser muss bis auf den i-Punkt dem Namen Ihres Web-Servers im DNS entsprechen, sonst maulen die Browser, wenn er ihnen sein Zertifikat vorlegt. Das *challenge password* hat zunächst keine tiefere Bedeutung; es wird im CRL abgelegt, und manche Zertifizierungsstellen benutzen es (Ihre Zertifizierungsstelle wird Sie gegebenenfalls wissen lassen, was Sie da hineinschreiben sollen). Den resultierenden CSR sehen Sie in Bild 4.1.

Zertifikat erstellen

Den CSR schicken Sie jetzt der Zertifizierungsstelle. Der Einfachheit halber und damit Sie sehen, wie Sie gegebenenfalls selber ein Zertifikat erstellen können, ziehen wir jetzt die im vorigen Kapitel definierte Zertifizierungsstelle heran, um aus dem CSR ein Zertifikat zu machen. Dazu setzen wir voraus, dass der eben erzeugte CSR (`server.csr`) im Verzeichnis der Zertifizierungsstelle steht. Die Variable `OPENSSL_CONF` sollte wieder auf die Konfigurationsdatei der Zertifizierungsstelle zeigen.

```
$ openssl ca -in server.csr
Using configuration from /usr/local/openssl/testCA/openssl.cnf
Enter pass phrase for /usr/local/openssl/testCA/private/cakey.pem:blafasel
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'DE'
localityName          :PRINTABLE:'Darmstadt'
organizationName      :PRINTABLE:'Beispiel GmbH'
```

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: CN=Test-CA, L=Darmstadt, C=DE/emailAddress=ca@example.com,
            O=Zertifizierungsstelle
        Validity
            Not Before: Jan 30 23:57:58 2013 GMT
            Not After : Jan 30 23:57:58 2015 GMT
        Subject: CN=www.example.com, L=Darmstadt,
            C=DE/emailAddress=webmaster@example.com, O=Beispiel GmbH,
            OU=Web-Server
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:d2:d1:a2:6b:22:4f:54:83:ae:60:6a:8f:37:05:
                    ◁◁◁◁◁
                    68:5b:57:b7:34:6e:f6:05:ba:04:5f:ba:f8:da:e0:
                    87:17:33:e0:7e:c6:54:b7:f3
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
    Signature Algorithm: sha1WithRSAEncryption
        69:c4:fa:fb:85:6b:b4:5d:cd:da:e1:c1:51:10:cc:1c:09:bc:
        ◁◁◁◁◁
        7d:e6:59:f0:37:0f:30:fc:fa:d8:2b:c4:77:91:a9:8e:6f:af:
        d7:cd:dc:64
-----BEGIN CERTIFICATE-----
MIIDBTCCAe2gAwIBAgIBATANBgkqhkiG9w0BAQQFADByMRAwDgYDVQQDEwdUZXN0
LUNBMRIwEAYDVQQHEwlEYXJtc3RhZHQxCzAJBgNVBAYTAkRFMR0wGwYJKoZIhvcN
◁◁◁◁◁
4WtHLvtjk58yZnZwt5gYgiVNmKjYAKqGRI847C1M+8VXlY595lnwNw8w/PrYK8R3
kamOb6/Xzdxk
-----END CERTIFICATE-----
```

Bild 4.2: Das neue Server-Zertifikat

```
organizationalUnitName:PRINTABLE:'Web-Server'
commonName             :PRINTABLE:'www.example.com'
emailAddress           :IA5STRING:'webmaster@example.com'
Certificate is to be certified until Jan 30 23:57:58 2015 GMT (730 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Certificate:
  ◁◁◁◁◁
Data Base Updated
$ _
```

Sie müssen als erstes das Kennwort für den privaten Schlüssel der Zertifizierungsstelle eingeben – dieser wird ja gebraucht, um das neue Zertifikat zu signieren.

Anschließend zeigt OpenSSL die verschiedenen Bestandteile des DN an. Als Zertifizierungsstelle würden Sie diese Parameter nun prüfen und sich zum Beispiel davon überzeugen, dass die beschriebene Organisation tatsächlich einen WWW-Server unter dem angegebenen Namen betreibt. Wenn die Prüfung zu Ihrer Zufriedenheit ausfällt, können Sie die Frage

```
Sign the certificate? [y/n]:
```

mit »y« wie »ja« beantworten und das Zertifikat damit offiziell beglaubigen. Als nächstes werden Sie noch einmal gefragt, ob Sie alle in diesem Arbeitsgang betrachteten Zertifikate (eins) wirklich in die Datenbank schreiben wollen; wenn Sie dies bejahen, werden die Zertifikate zur Kontrolle auf dem Bildschirm ausgegeben (Bild 4.2). Außerdem landen die neuen Zertifikate in dem Verzeichnis, das wir in der Konfigurationsdatei der Zertifizierungsstelle (Bild 3.2) als `new_certs_dir` angegeben haben; der Dateiname in diesem Verzeichnis entspricht der (hexadezimalen) Seriennummer des Zertifikats. Die »Datenbank« in der Datei `index.txt` enthält ein Verzeichnis aller bisher ausgegebenen Zertifikate; dies ist wichtig, um später gegebenenfalls Zertifikate zurückrufen zu können.

Als Zertifizierungsstelle würden Sie nun das neu erzeugte Zertifikat (`certs/01.pem`) an den Antragsteller des CSR zurückschicken. Bei uns ist das eine einfache Kopie:

```
$ cp certs/01.pem ~/server.crt
```

4.3 Installation eines Server-Zertifikats

Damit unser Apache-Server das neue Server-Zertifikat auch benutzen kann, müssen wir es nur noch zusammen mit dem dazugehörigen privaten Schlüssel (den wir der Zertifizierungsstelle natürlich *nicht* geschickt haben!) an den richtigen Platz in der Apache-Konfiguration kopieren. Die Chancen stehen gut, dass Ihre Apache-Konfiguration schon für SSL-Zertifikate vorbereitet ist oder gar ein Testzertifikat enthält; dieses können wir ohne weiteres überschreiben. Die relevanten Direktiven sind:

`SSLCertificateFile` ⟨*Datei*⟩ In dieser Datei steht das Zertifikat des Servers.

`SSLCertificateKeyFile` ⟨*Datei*⟩ In dieser Datei steht der dazugehörige private Schlüssel. Es ist auch möglich, den Schlüssel in dieselbe Datei zu schreiben wie das Zertifikat; in diesem Fall wird diese Datei nicht angeschaut.

`SSLCertificateChainFile` ⟨*Datei*⟩ Diese Datei enthält *alle* Zertifikate einer »Zertifikatskette« vom Server-Zertifikat zu einem Zertifikat einer Zertifizierungsstelle. Verwenden Sie diese Datei, wenn Ihr Zertifikat nicht unmittelbar von einer Zertifizierungsstelle signiert wurde, sondern von jemandem, der von einer Zertifizierungsstelle ein Zertifikat bekommen hat, mit dem er selber weitere Zertifikate signieren kann und so weiter. Auch diese Zertifikate lassen sich gegebenenfalls direkt ins `SSLCertificateFile` schreiben.

Die entsprechenden Direktiven nehmen Sie am besten in eine Definition für einen IP-basierten virtuellen Server auf. Sie müssen darauf achten, dass Ihr Apache-Server auf dem HTTPS-Port, dem Port 443, auf Anfragen lauscht, und dass `mod_ssl` aktiviert ist. Hier ein Vorschlag für die dazugehörigen Zeilen in `httpd.conf`:

```
LoadModule ssl_module /usr/lib/apache/2.4/mod_ssl.so
Listen *:443
<VirtualHost _default_:443>
  SSLEngine on
  SSLCertificateFile /etc/apache/ssl/server.crt
```

```
  SSLCertificateKeyFile /etc/apache/ssl/server.key
</VirtualHost>
```

Dann müssen Sie nur noch die Zertifikatsdatei `server.crt` und die Schlüsseldatei `server.key` an die durch die Direktiven `SSLCertificateFile` und `SSLCertificateKeyFile` gegebenen Plätze kopieren.

Wenn Sie eine der SUSE-Distributionen verwenden, sollten Sie das Zertifikat nach `/etc/httpd/ssl.crt/server.crt` und den privaten Schlüssel nach `/etc/httpd/ssl.key/server.key` kopieren. Außerdem müssen Sie in der Datei `/etc/sysconfig/apache2` dafür sorgen, dass `ssl` im Wert der Variablen `APACHE_MODULES` vorkommt, sowie `SSL` in die `APACHE_SERVER_FLAGS` eintragen.

Ein Neustart des Apache-Servers (Neuladen der Konfiguration reicht leider nicht aus) sollte dann genügen, um den SSL-Web-Server zu aktivieren. Bei dieser Gelegenheit müssen Sie das Kennwort für den verschlüsselten privaten Schlüssel des Servers eingeben:

```
# /etc/init.d/apache2 start
Starting web server: apache
Apache/2.2.14 mod_ssl/2.2.14 (Pass Phrase Dialog)
Some of your private key files are encrypted for security reasons.
In order to read them you have to provide us with the pass phrases.

Server localhost:443 (RSA)
Enter pass phrase:123456

Ok: Pass Phrase Dialog successful.
.
# _
```

Eine weitere SUSE-Eigenheit ist, dass der Apache beim Start nur sehr kurz darauf wartet, dass Sie die Passphrase eingeben – zwei Sekunden müssen gemäß Standardeinstellung reichen. Wenn Sie nicht gerade über die Reaktionszeit eines Jagdfliegers und die Fingerfertigkeit eines Konzertpianisten verfügen, könnte das eine gewisse Herausforderung darstellen. Sie tun besser daran, in der Variablen `APACHE_START_TIMEOUT` in der Datei `/etc/sysconfig/apache2` eine längere Frist festzulegen. Je nach der Komplexität Ihrer Passphrase ist ein Wert von 10 bis 15 Sekunden angemessen.

Wenn es Sie grundsätzlich stört, dass Sie beim Start Ihres Apache-Servers zugegen sein müssen, um die Passphrase einzugeben, können Sie entweder die Verschlüsselung des privaten Schlüssels entfernen (ein Kommando wie

```
$ openssl rsa -in server.key -out server-np.key
Enter pass phrase for server.key: 123456
writing RSA key
$ _
```

erledigt das problemlos) oder die Passphrase anderweitig »eintippen«. Zum Beispiel könnten Sie die Direktive

```
SSLPassPhraseDialog exec:/usr/local/sbin/apache-passphrase
```

definieren und in `/usr/local/sbin/apache-passphrase` ein Skript wie

```
#!/bin/sh
echo 123456
```

hinterlegen. Sie sollten dann dafür sorgen, dass dieses Skript nur für den Benutzer les- und ausführbar ist, der für den Apache verwendet wird (etwa `wwwrun` bei der SUSE oder `www-data` bei Debian GNU/Linux).

Test Überzeugen Sie sich, dass der Apache auf dem Port 443 lauscht und auf eine Anfrage an `https://www.example.com/` mit der üblichen Indexseite antwortet – in gängigen Browsern sollte an einem zugeklappten Vorhängeschloss oder einem ganzen (nicht zerbrochenen) Schlüssel zu erkennen sein, dass die Seite über eine HTTPS-Verbindung geholt wurde.

Es ist damit zu rechnen, dass Ihr Browser die HTTPS-Verbindung nicht widerstandslos aufbaut – er sollte zumindest anmeckern, dass er das Zertifikat des Servers nicht akzeptiert, weil er das Zertifikat der Zertifizierungsstelle nicht zur Verfügung hat und darum die Authentizität des Server-Zertifikats nicht prüfen kann. Normalerweise können Sie als Benutzer da Ausnahmen veranlassen; ansonsten erklärt der nächste Abschnitt, wie Sie Ihrem Browser das Zertifikat Ihrer Zertifizierungsstelle beibringen können.

Übungen

4.1 [!3] Erzeugen Sie wie angegeben einen CSR und ein Zertifikat für Ihren WWW-Server (achten Sie auf den korrekten engl. *common name*). Installieren Sie den Schlüssel und das Zertifikat und vergewissern Sie sich, dass Ihr Server mit SSL funktioniert.

4.2 [2] Sie haben oben Ihren HTTPS-Server mit dem URL `http://www.example.com/` getestet. Was passiert, wenn Sie statt dessen `http://localhost/` angeben, und warum?

4.4 Weitere Server-Konfiguration

Neben den Pfadnamen für Zertifikate und Schlüssel können Sie auch verschiedene andere »serverseitige« Aspekte von `mod_ssl` konfigurieren. Zum Beispiel:

4.4.1 Mehrere virtuelle SSL-Server in einem Apache

Apache kann mehrere virtuelle Server gleichzeitig verwalten, die SSL benutzen. Allerdings sind dabei einige Besonderheiten zu beachten, die aus dem Umstand herrühren, dass Apache schon beim SSL-Verbindungsaufbau das korrekte Zertifikat für den virtuellen Server liefern muss. Bei namensbasierten virtuellen Servern kann Apache aber erst am Wert der `Host`-Kopfzeile erkennen, welcher virtuelle Server tatsächlich gemeint ist – aber da muss die Verbindung längst stehen (sonst könnte der Browser ja nichts schicken, auch keine `Host`-Zeile). Als Server-Betreiber haben Sie die folgenden Möglichkeiten:

IP-Adresse
- Am einfachsten ist es, jedem virtuellen Server eine eigene IP-Adresse zuzuordnen. Dann ist von Anfang an klar, welches Zertifikat gebraucht wird. Allerdings ist es (jedenfalls bis zur flächendeckenden Einführung von IPv6) möglicherweise schwierig, IP-Adressen in der geforderten Anzahl zu bekommen – wenn es nur um eine Handvoll geht, dann sollte Ihr Provider allerdings mit sich reden lassen. Dieser Ansatz funktioniert immer.

Wildcard-Zertifikat
- Solange alle Server in derselben Domain sind, können Sie ein »Wildcard-Zertifikat« verwenden, also eins, das für etwas wie `*.example.com` ausgestellt ist. Allerdings ist unklar, wie so ein Zertifikat genau interpretiert werden soll, und es ist auch möglich, dass es zur Verifikation von mehr Servern als erwünscht verwendet werden kann. Wir raten von diesem Ansatz ab.

- Sie können Zertifikate erzeugen, die für mehr als einen Servernamen gelten. Gemäß [RFC2459] ist es möglich, weitere Servernamen in der `subjectAltName`-Erweiterung eines Zertifikats zu hinterlegen. Dazu muss die OpenSSL-Konfiguration für die Erzeugung des CSR etwas enthalten wie

  subjectAltName

  ```
  [req]
  req_extensions = v3_req
  [v3_req]
  subjectAltName = DNS: foo.example.com, DNS:bar.example.net
  ```

  Sie sollten außerdem einen dieser Namen als `CN` für das Zertifikat angeben, damit den Formalien Genüge getan ist. Dieser Ansatz funktioniert jedenfalls für Apache mit OpenSSL und allen gängigen Browsern. Allerdings ist es lästig, dass alle Namen sich dasselbe Zertifikat teilen und darum ein neues Zertifikat ausgestellt werden muss, wenn sich ein Name ändert, wegfällt oder ein neuer dazukommt.

- In [RFC4366, Abschnitt 3.1] wird eine TLS-Erweiterung namens »Server Name Indication« (SNI) standardisiert, mit der ein Client schon beim Aufbau einer Verbindung bekanntgeben kann, mit welchem (virtuellen) Server er reden möchte – sozusagen eine vorgezogene `Host`-Zeile. Der Server kann dann sofort das richtige Zertifikat liefern. Die Unterstützung dafür ist auf der Browser-Seite relativ gut (die aktuellen Versionen von Firefox, Opera, Internet Explorer, Google Chrome und Safari spielen alle mit – unter Windows allerdings nur mit Vista und Windows 7, nicht Windows XP); auf der Server-Seite tut man sich da noch etwas schwerer (Apache kann SNI seit Version 2.2.12). Dies ist eindeutig die Zukunft.

  Server Name Indication

In der Apache-Konfiguration können Sie wie üblich mehrere `<VirtualHost>`-Blöcke vorsehen. Bei mehreren IP-basierten virtuellen Servern oder SNI bekommt jeder Server eine eigene Zertifikat/Schlüssel-Kombination mit `SSLCertificateFile`- und `SSLCertificateKeyFile`-Direktiven; bei einem Zertifikat mit `subjectAltName` können Sie entweder in mehreren `<VirtualHost>`-Blöcken auf das Zertifikat (und den Schlüssel) verweisen oder einen `<VirtualHost>`-Block mit `ServerAlias` einsetzen. Ansonsten entspricht die Konfiguration der für virtuelle Server üblichen Vorgehensweise.

4.4.2 Zugriff auf Ressourcen ausschließlich über SSL

In unserem bisherigen Beispiel war der gesamte Ressourcenbaum des Servers sowohl über gewöhnliches HTTP als auch über HTTPS zugänglich. Es gibt aber die (oft sinnvolle) Möglichkeit, eine Web-Präsenz oder Teile davon ausschließlich über SSL zu ermöglichen. Hierzu dient die Direktive `SSLRequireSSL`. Wenn sie in einem `<Directory>`-Block oder einer `.htaccess`-Datei auftaucht, dann sind die Inhalte in und unter dem betreffenden Verzeichnis nur über HTTPS erreichbar, nicht mehr über HTTP.

Trotz `SSLRequireSSL` kann es immer noch möglich sein, ohne SSL Zugriff auf eine Ressource zu bekommen, nämlich wenn für die betreffende Ressource »`Satisfy Any`« in Kraft ist. Um das auszuschließen, müssen Sie außer `SSLRequireSSL` noch die SSL-Option `StrictRequire` setzen. Siehe hierzu auch die `mod_ssl`-Dokumentation.

4.4.3 SSL-Optionen

Die Apache-Direktive `SSLOptions` dient zur Konfiguration diverser SSL-Optionen, unter anderem:

`StdEnvVars` Sorgt dafür, dass die SSL-Umgebungsvariablen für CGI-Skripte und *server-side includes* erzeugt werden. Dies ist eine relativ aufwendige Operation und sollte daher nur für veritable CGI- oder SSI-Anfragen eingeschaltet werden (etwa in einem mit `ScriptAlias` angemeldeten CGI-Verzeichnis).

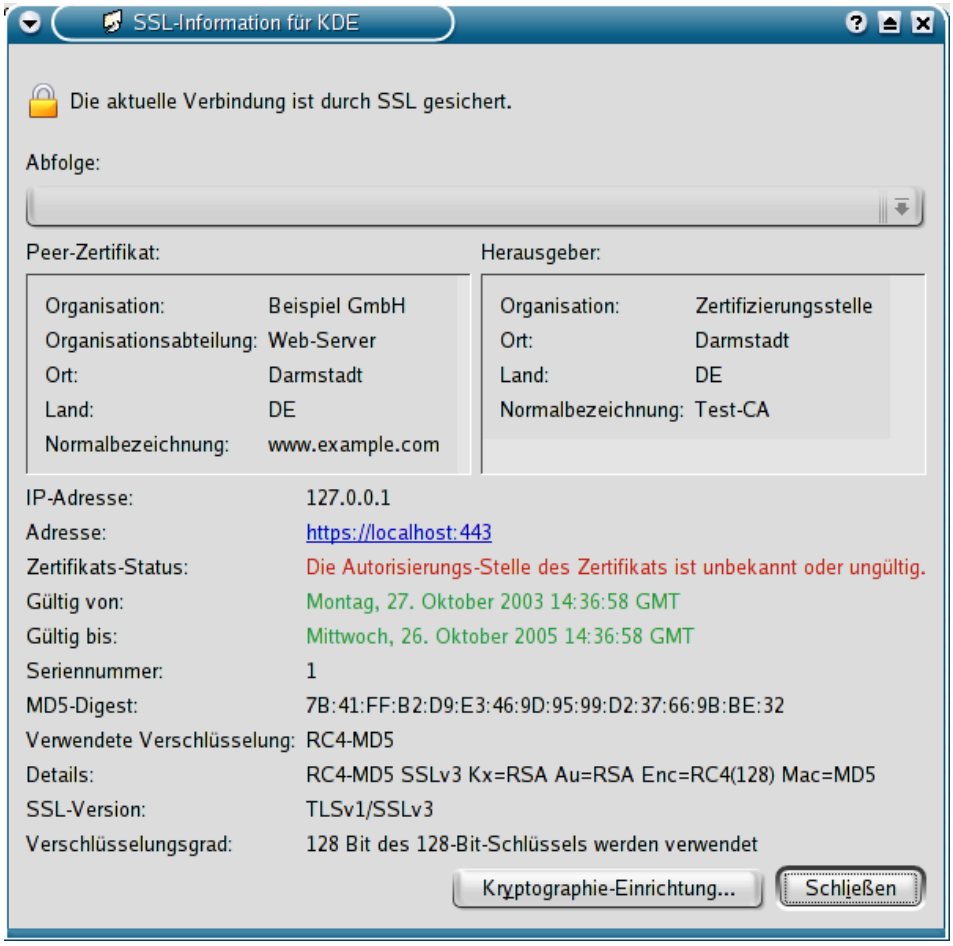


Bild 4.3: SSL-Informationen im Konqueror-Browser

`ExportCertData` Definiert zusätzliche Umgebungsvariablen mit den benutzten Zertifikaten (in PEM-Codierung). CGI-Skripte können diese Informationen benutzen, um die Zertifikate noch pingeliger zu prüfen, aber die Umgebung wird aufgebläht, weswegen Sie diese Funktionalität gesondert einschalten müssen.

`StrictRequire` Wie schon im vorigen Abschnitt erwähnt, können Sie mit dieser Option erzwingen, dass die Einschränkungen von `mod_ssl` selbst unter »`Satisfy any`« Bestand haben: Nicht-SSL-Zugriffe auf SSL-Ressourcen werden auf jeden Fall abgewiesen, und Zugriffe, die die Bedingungen von `SSLRequire` nicht erfüllen, auch.

Die `SSLOptions`-Direktiven werden behandelt wie die `Options`-Direktiven: Tauchen mehrere auf, so gilt immer diejenige, die am »nächsten« am betreffenden Verzeichnis ist. Eine Ausnahme gilt für `SSLOptions`-Direktiven, in denen *jede* Option ein »+« oder »-« hat: Die so benannten Optionen werden zu den gerade gültigen Optionen hinzugefügt oder aus dieser Menge entfernt.

4.5 Browser und CA-Zertifikate

Wie eben angedeutet, akzeptieren Browser anstandslos nur solche Zertifikate, die von Zertifizierungsstellen signiert wurden, deren Zertifikate dem Browser zur Verfügung stehen. Das ist auch ganz richtig so, denn durch eine geeignete Auswahl von Zertifizierungsstellen kann der Anwender des Browsers steuern, wem er (blind) vertrauen möchte und wem nicht. Ist an einem Zertifikat, das ein Web-

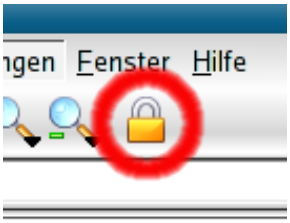


Bild 4.4: Abrufen von SSL-Parametern

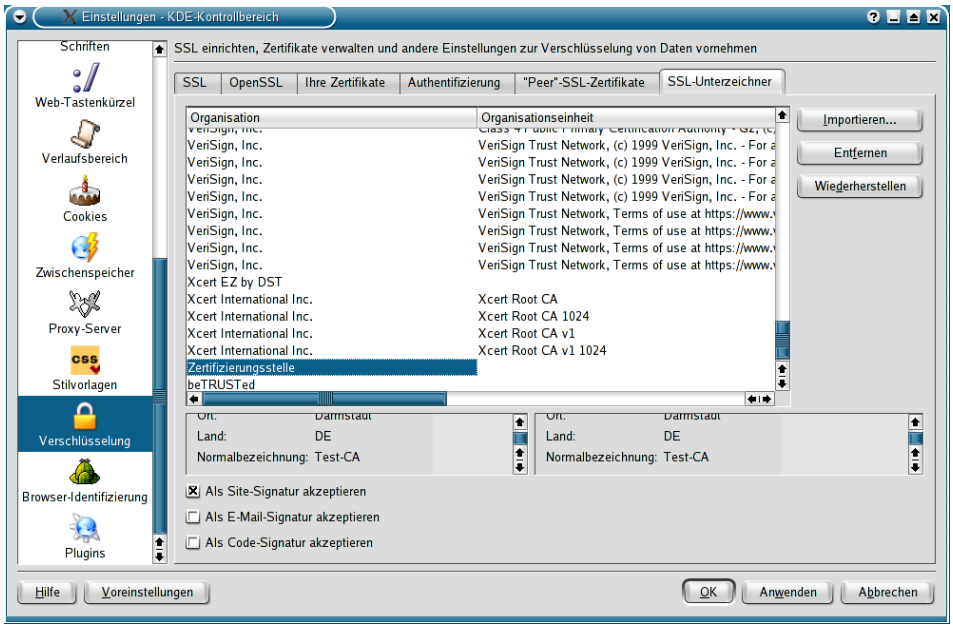


Bild 4.5: Konfiguration der Zertifikate von Zertifizierungsstellen im Konqueror-Browser

Server präsentiert, etwas auszusetzen, zeigt der Browser einen Informationsdialog an, der aussehen kann wie in Bild 4.3 (Konqueror): Neben dem Zertifikat des aktuellen Web-Servers wird auch das der Zertifizierungsstelle angezeigt; außerdem finden Sie die IP-Adresse und URL der Gegenstelle, die Gültigkeitsdauer des Zertifikats und eine Nachricht, die angibt, worin genau das Problem besteht (in Bild 4.3 ist, wie nicht anders zu erwarten, die Zertifizierungsstelle nicht bekannt). Ferner bekommen Sie Details der ausgehandelten SSL-Verschlüsselungsverfahren gezeigt. Aufgrund dieser Informationen können Sie sich entscheiden, ob Sie in diesem speziellen Fall eine Ausnahme machen möchten.

Sie können die SSL-Informationen auch aufrufen, wenn der Browser mit der aktuellen Situation kein Problem hat. Im Konqueror können Sie dazu auf das »Schloss«-Bildchen in der Werkzeugleiste klicken (Bild 4.4).

Gerade wenn Sie sich entschieden haben, selber als Zertifizierungsstelle aufzutreten, ist es aber sinnvoll bzw. nötig, Ihrem Browser das Zertifikat Ihrer Zertifizierungsstelle mitzuteilen. Hierzu sollten Sie den entsprechenden Konfigurationsdialog Ihres Browsers aufrufen (Bild 4.5 zeigt die »Kryptografie-Einrichtung« des Konqueror-Browsers; Zertifikate von Zertifizierungsstellen werden in der Karteikarte »SSL-Unterzeichner« konfiguriert).

Dort gibt es in der Regel eine Funktion namens »Importieren« (oder so ähnlich), mit der Sie ein in einer Datei vorliegendes Zertifikat in die Liste aufnehmen können. Daneben können Sie noch festlegen, wofür das Zertifikat zu gebrauchen sein soll (in Bild 4.5 unten links, neben der Auswahlleiste für die Konfigurationsbereiche) – zur Authentisierung der Zertifikate von Webpräsenzen, E-Mail-Absendern oder Code. Die beiden letzteren Anwendungen werden unter Linux nicht so oft eingesetzt, und überhaupt gibt es zunächst keinen Grund, warum Sie ausgerechnet diese Zertifizierungsstelle auch für Mail oder Codesignaturen akkreditieren sollten; am besten klicken Sie diese beiden Punkte weg.

Jetzt sollte Ihr Browser das Server-Zertifikat anstandslos akzeptieren.

Zusammenfassung

- Ein Apache-Server kann sich über SSL authentisieren, wenn er über ein geeignetes Server-Zertifikat verfügt.
- Sie können ein Server-Zertifikat erhalten, indem Sie mit OpenSSL einen X.509-*Certificate Signing Request* erzeugen und diesen entweder einer Zertifizierungsstelle vorlegen oder ihn selbst signieren.
- Für öffentliche Web-Server empfiehlt sich die Inanspruchnahme einer Zertifizierungsstelle; in geschlossenen Benutzergruppen ist die Selbstzertifizierung sinnvoller.
- Um einen SSL-Server zu konfigurieren, müssen Sie Zertifikat und öffentlichen Schlüssel installieren und (sinnvollerweise) einen IP-basierten virtuellen Server auf Port 443 einrichten.
- Ein Apache-Server kann mehrere Web-Präsenzen mit SSL verwalten, solange diese entweder als IP-basierte virtuelle Server implementiert sind, ein Zertifikat mit `subjectAltName`-Erweiterung verwendet wird oder Client und Server SNI unterstützen.
- Die Direktive `SSLRequireSSL` macht es möglich, Zugriffe auf Ressourcen nur über SSL zu erlauben.
- Mit `SSLOptions` können diverse Optionen gesetzt werden.
- Die meisten Browser erlauben die Abfrage der aktuellen SSL-Parameter und die Installation zusätzlicher Zertifikate für Zertifizierungsstellen.

Literaturverzeichnis

RFC2459 R. Housley, W. Ford, W. Polk, et al. »Internet X.509 Public Key Infrastructure Certificate and CRL Profile«, Januar 1999.
http://www.ietf.org/rfc/rfc2459.txt

RFC4366 S. Blake-Wilson, M. Nystrom, D. Hopwood, et al. »Transport Layer Security (TLS) Extensions«, April 2006. http://www.ietf.org/rfc/rfc4366.txt

VMC02 John Viega, Matt Messier, Pravir Chandra. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly & Associates, 2002. ISBN 0-596-00270-X.
http://www.oreilly.com/catalog/openssl/

LINUP FRONT

5

Client-Authentisierung über Client-Zertifikate

Inhalt

Lernziele

- Client-Zertifikate erstellen und installieren können
- Authentisierung und Autorisierung über Client-Zertifikate verstehen
- Einfache und komplexe Zugriffsrechte auf der Basis von Client-Zertifikaten vergeben können
- Zertifikats-Rückruflisten verwalten können

Vorkenntnisse

- Kenntnisse über OpenSSL und Zertifikate (Kapitel 3)
- Konfiguration und Betrieb eines Apache-Servers
- Erzeugung und Installation von Server-Zertifikaten (Kapitel 4)
- Benutzung eines WWW-Browsers

5.1 Erzeugung von Client-Zertifikaten

Client-Zertifikate werden im Prinzip erstellt wie Server-Zertifikate: Zuerst wird ein CSR erstellt und an die Zertifizierungsstelle übermittelt:

```
$ openssl req -newkey rsa:1024 -keyout hugo.key \
>   -out hugo.csr
Generating a 1024 bit RSA private key
.....................................................++++++
.............++++++
writing new private key to 'hugo.key'
Enter PEM pass phrase:x8o,Aq5!k
Verifying - Enter PEM pass phrase:x8o,Aq5!k
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:Darmstadt
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Beispiel GmbH
Organizational Unit Name (eg, section) []:Entwicklung
Common Name (eg, YOUR name) []:Hugo Schulz
Email Address []:hugo@example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:bli bla blubb
An optional company name []:Beispiel GmbH
$ _
```

Die Zertifizierungsstelle prüft die Authentizität des CSR und erzeugt daraus das von ihr signierte Zertifikat:

```
$ openssl ca -in hugo.csr
```

Das resultierende Zertifikat bekommt der betreffende Benutzer und kann es dann in seinem Browser installieren. Die meisten Browser erwarten hierfür Zertifikate
PKCS#12 im **PKCS#12**-Format, das Sie wie folgt erzeugen können:

```
$ openssl pkcs12 -export -in hugo.crt -inkey hugo.key -out hugo.p12
Enter pass phrase for hugo.key:x8o,Aq5!k
Enter Export Password:foo-bar
Verifying - Enter Export Password:foo-bar
$ _
```

Hierbei werden Zertifikat und privater Schlüssel zusammengefügt und gemein-
Exportkennwort sam mit einem »Exportkennwort« verschlüsselt. Das Exportkennwort können Sie frei wählen; es muss nichts mit dem schon für den privaten Schlüssel vergebenen Kennwort zu tun haben.

Der Browser braucht Zugriff nicht nur auf das Zertifikat, sondern auch auf den dazugehörigen privaten Schlüssel – das Zertifikat könnte ja jeder, dem es irgendwann mal in die Hände gefallen ist, einem Server präsentieren.

Der Server muss sich also vergewissern, dass am anderen Ende der Verbindung wirklich der rechtmäßige Inhaber des Zertifikats sitzt, und das geht über ein *challenge-response*-Verfahren, bei dem der Browser etwas mit dem öffentlichen Schlüssel Verschlüsseltes entschlüsseln muss. Und dafür ist bekanntlich der private Schlüssel nötig.

Im Konqueror-Browser können Sie zum »Importieren« eines Client-Zertifikats die Karteikarte »Ihre Zertifikate« in der »Kryptografie-Konfiguration« verwenden. Hierbei werden Sie nach dem Exportkennwort gefragt, das Sie beim Anlegen des PKCS#12-Pakets vergeben haben. In der Karteikarte »Authentifizierung« können Sie dann bestimmen, welche Server wann welches Client-Zertifikat präsentiert bekommen.

Übungen

5.1 [2!] Erzeugen Sie sich ein Client-Zertifikat und installieren Sie es in Ihrem Browser.

5.2 Server-Konfiguration

5.2.1 Grundlagen

Die Konfiguration des Apache-Servers zur Authentisierung von Clients über Client-Zertifikate besteht aus zwei Teilen:

Client-Zertifikate prüfen

1. Der Server muss in die Lage versetzt werden, Client-Zertifikate anzufordern und deren Gültigkeit zu prüfen (also die Signatur(en) der Zertifizierungsstelle(n) auf dem Zertifikat). Hierzu benötigt er das Zertifikat der Zertifizierungsstelle (bzw. die Zertifikate aller beteiligten Unter-Zertifizierungsstellen). Genau so, wie ein Browser das Zertifikat einer Zertifizierungsstelle aus »unabhängiger Quelle« haben muss – typischerweise indem es schon in den Browser integriert ist, wenn Sie ihn bekommen –, kann auch der Server sich nicht auf die Echtheit eines Zertifizierungsstellen-Zertifikats verlassen, das er von irgendwoher bekommt (am Ende gar vom zu authentisierenden Client selbst). Im typischen Fall, wo Sie Ihre eigene Zertifizierungsstelle sind, ist das natürlich kein Problem.

Zugriffsregeln

2. Für die zu schützenden Ressourcen müssen Zugriffsregeln aufgestellt werden, die sich auf Daten aus dem Zertifikat beziehen. Wenn Sie zum Beispiel eine Ressource nur den Mitgliedern der Entwicklungsabteilung zugänglich machen wollen, können Sie prüfen, ob der DN des Client-Zertifikats einen Eintrag wie »`OU=Entwicklung`« enthält. Apache unterstützt hierfür eine reichhaltige Auswahl von Möglichkeiten.

SSLVerifyClient

Mit der Direktive `SSLVerifyClient` können Sie bestimmen, ob und mit welchem Nachdruck Apache Client-Zertifikate anfordert. Die möglichen Werte sind:

`none` Es wird kein Client-Zertifikat verlangt

`optional` Ein Client-Zertifikat ist nicht nötig, wird aber angenommen, wenn der Browser eins schickt

`require` Der Client *muss* ein Zertifikat vorlegen

`optional_no_ca` Es ist kein Client-Zertifikat nötig, und wenn der Browser eins schickt, muss es nicht verifizierbar sein.

Wirklich sinnvoll sind nur `none` und `required`, ab und zu auch `optional`. Die `SSLVerifyClient`-Direktive kann in einem serverweiten Kontext (global oder in einem `<VirtualHost>`-Block) oder in der Konfiguration für ein Verzeichnis auftreten; im letzteren Fall

wird nach dem Lesen der HTTP-Anfrage eine Neuaushandlung der SSL-Kryptografieparameter erzwungen.

Client-Zertifikate können direkt von einer Zertifizierungsstelle ausgestellt worden sein, mittelbar über ein Zertifikat beglaubigt werden, das die Zertifizierungsstelle ausgestellt hat, und so weiter. Zwischen dem Client-Zertifikat und dem selbstsignierten Zertifikat der Zertifizierungsstelle existiert also eine »Kette« von Zertifikaten, in der jeweils die »nächsthöhere« Instanz (also die, die der eigentlichen Zertifizierungsstelle näher ist) die Echtheit jedes Zertifikats beglaubigt. Mit der Direktive `SSLVerifyDepth` können Sie bestimmen, wie lang eine solche Kette jenseits des zu prüfenden Client-Zertifikats maximal sein darf, damit Client-Zertifikate noch als authentisch gelten. Die Voreinstellung ist 1; Client-Zertifikate müssen also direkt von der Zertifizierungsstelle beglaubigt worden sein. Mit »`SSLVerifyDepth 0`« wären nur selbstsignierte Zertifikate zulässig (ziemlicher Unfug), während mit »`SSLVerifyDepth 3`« die Zertifizierungsstelle ein Zertifikat für eine Unter-Zertifizierungsstelle ausstellen könnte, diese Unter-Zertifizierungsstelle könnte wiederum eine Unter-Unter-Zertifizierungsstelle einrichten, und von dieser ausgestellte Client-Zertifikate würden noch als authentisch gelten.

Zertifizierungs-»Kette«

SSLVerifyDepth

Kürzer als verlangt darf die Zertifikatskette natürlich sein, nur nicht länger.

Zur Prüfung der Client-Zertifikate müssen dem Server die Zertifikate der Zertifizierungsstellen vorliegen, die die Client-Zertifikate beglaubigt haben (es können durchaus mehrere unabhängige Unter-Zertifizierungsstellen sein, die ihre Autorität jeweils über Zertifikatsketten von möglicherweise verschiedenen Zertifizierungsstellen ableiten). Es gibt zwei Möglichkeiten, dem Server diese Zertifikate bekanntzumachen.

Zunächst können Sie die Zertifikate aller beteiligten Zertifizierungsstellen (inklusive der Zertifikate von in den Ketten übergeordneten Zertifizierungsstellen) im PEM-Format aneinanderhängen und in eine große Datei schreiben. Diese Datei machen Sie Apache dann über die Direktive `SSLCACertificateFile` bekannt. Alternativ dazu können Sie die Zertifikate in einzelnen Dateien in ein Verzeichnis schreiben und Apache den Namen dieses Verzeichnisses über die Direktive `SSLCACertificatePath` mitteilen. Hierbei müssen Sie beachten, dass der Server in der Lage sein muss, bei Bedarf in diesem Verzeichnis das richtige Zertifikat zu finden; `mod_ssl` verwendet dazu *hash filenames*, also symbolische Links auf die tatsächlichen Zertifikatsdateien mit einem automatisch auf der Basis des Zertifikatinhaber-DNs vergebenen Namen. Diese *hash filenames* können Sie mit einem bei `mod_ssl` mitgelieferten `Makefile` anlegen, das Sie am besten in das betreffende Verzeichnis kopieren; anschließend müssen Sie nur noch in dieses Verzeichnis wechseln und `make` aufrufen.

SSLCACertificateFile

SSLCACertificatePath

hash filenames

Ausgerechnet werden die *hash filenames* innerhalb des `Makefiles` mit dem Kommando »`openssl x509 -hash`«. Aus dem Inhaber-DN »`CN=Hugo Schulz, L=Darmstadt, C=DE/emailAddress=hugo@example.com, O=Beispiel GmbH, OU=Entwicklung`« ergibt sich zum Beispiel der *hash filename* `8d047a10.N`.

Nur damit keine Verwirrung aufkommt: Sie müssen dem Apache natürlich nicht sämtliche Zertifikate von *Clients* zur Verfügung stellen, sondern nur die Zertifikate der Zertifizierungsstellen, die diese Zertifikate beglaubigt haben (und das Zertifikat der Zertifizierungsstelle, die die Zertifikate der beglaubigenden Zertifizierungsstellen beglaubigt hat und so weiter[1]).

Weder die `SSLCACertificateFile`- noch die `SSLCACertificatePath`-Methode hat klare Vorteile gegenüber der jeweils anderen. Das Sammeln aller Zertifikate in einer großen Datei macht die Konfiguration übersichtlicher, aber das Inkraftsetzen neuer Zertifikate oder das Streichen nicht mehr gebrauchter sind relativ mühselig. Beim Zertifikatsverzeichnis ist das wiederum einfacher – Sie müssen nur Dateien umkopieren oder löschen und `make` aufrufen –, aber dafür sorgen die *hash filenames* für eine gewisse Unordnung, und sie müssen ja auch aktuell sein. Im übrigen spricht nichts dagegen, beide Ansätze gleichzeitig zu benutzen und manche

[1] *It's turtles all the way down …*

Zertifikate in einer großen Datei und andere in einem Verzeichnis unterzubringen.

Beispiele

Hier noch ein paar Konfigurationsbeispiele. Zunächst sehen Sie hier, wie Sie alle Zugriffe auf Ihre Web-Präsenz auf diejenigen Anwender beschränken können, die über ein direkt von Ihrer Zertifizierungsstelle signiertes Client-Zertifikat verfügen:

```
SSLVerifyClient require
SSLVerifyDepth 1
SSLCACertificateFile /etc/apache/ssl.crt/cacert.pem
```

(der Pfadname des Zertifizierungsstellen-Zertifikats kann bei Ihrer Distribution geringfügig abweichen). Im folgenden Beispiel wird nur ein Teilbereich der Web-Präsenz ausschließlich Zertifikatsinhabern zugänglich gemacht:

```
SSLVerifyClient none
SSLCACertificateFile /etc/apache/ssl.crt/cacert.pem
<Location /members>
  SSLVerifyClient require
  SSLVerifyDepth 1
</Location>
```

Für den größten Teil der Web-Seiten ist kein Client-Zertifikat nötig (der Wert von `SSLVerifyClient` ist `none`). Nur bei Anfragen nach URLs wie `http://www.example.com/members/test.html` muss ein gültiges Client-Zertifikat vorgelegt werden.

Das folgende Beispiel aus [MODSSL-DOC] zeigt, wie Sie bestimmten Benutzern auf der Basis von Zertifikaten Zugriffsrechte auf einen bestimmten Teilbereich Ihrer Web-Präsenz einräumen, während der Rest der Präsenz für alle Benutzer zugänglich ist. Hierfür nutzen wir die SSL-Option `FakeBasicAuth`, mit der der DN des Zertifikatsinhabers als »Benutzername« für Basic-Authentisierung verwendet wird. Damit können Sie alle Authentisierungsmethoden verwenden, die Basic-Authentisierungsdaten benutzen, im einfachsten Fall also das Durchsuchen einer `.htpasswd`-Kennwortdatei, aber auch Module, die Benutzer in Datenbankdateien, relationalen Datenbanken, LDAP usw. ablegen. Da die Authentisierung bereits auf der Basis des Client-Zertifikats erfolgt, ist ein benutzerspezifisches Kennwort in der Datenbank für Basic-Authentisierung nicht mehr nötig; `mod_ssl` verwendet für *alle* diese »gefälschten« Benutzernamen das Kennwort »`password`«, das Sie in einer DES-basierten Kennwortdatenbank als »`xxj31ZMTZzkVA`« und in einer Kennwortdatenbank auf der Basis von MD5-Kennwörtern als »`$1$OXLyS...$Owx8s2/m9/gfkcRVXzgoE/`« eintragen können. Die Apache-Konfiguration könnte ungefähr so aussehen:

```
SSLVerifyClient none
<Directory /srv/www/htdocs/members>
  SSLVerifyClient      require
  SSLVerifyDepth       2
  SSLCACertificateFile /etc/httpd/ssl.crt/ca.crt
  SSLOptions           +FakeBasicAuth
  SSLRequireSSL
  AuthName             "Beispiel-Mitgliederseiten"
  AuthType             Basic
  AuthUserFile         /etc/httpd/httpd.passwd
  Require              valid-user
</Directory>
```

Hierbei müssen Sie natürlich dafür sorgen, dass alle Benutzer mit dem oben angegebenen Kennwort in der Kennwortdatenbank (hier `/etc/httpd/httpd.passwd`) eingetragen sind. In diesem Beispiel werden übrigens nicht nur Zertifikate akzeptiert, die unmittelbar von der Zertifizierungsstelle ausgestellt wurden, sondern auch solche von Unter-Zertifizierungsstellen.

⟨*Ausdruck*⟩ ← `true` | `false`
| `!` ⟨*Ausdruck*⟩ *Logische Negation*
| ⟨*Ausdruck*⟩ `&&` ⟨*Ausdruck*⟩ *Logische Und-Verknüpfung*
| ⟨*Ausdruck*⟩ `||` ⟨*Ausdruck*⟩ *Logische Oder-Verknüpfung*
| `(` ⟨*Ausdruck*⟩ `)`
| ⟨*Vergleich*⟩

⟨*Vergleich*⟩ ← ⟨*Wort*⟩ `==` ⟨*Wort*⟩ | ⟨*Wort*⟩ `eq` ⟨*Wort*⟩
| ⟨*Wort*⟩ `!=` ⟨*Wort*⟩ | ⟨*Wort*⟩ `ne` ⟨*Wort*⟩
| ⟨*Wort*⟩ `<` ⟨*Wort*⟩ | ⟨*Wort*⟩ `lt` ⟨*Wort*⟩
| ⟨*Wort*⟩ `<=` ⟨*Wort*⟩ | ⟨*Wort*⟩ `le` ⟨*Wort*⟩
| ⟨*Wort*⟩ `>` ⟨*Wort*⟩ | ⟨*Wort*⟩ `gt` ⟨*Wort*⟩
| ⟨*Wort*⟩ `>=` ⟨*Wort*⟩ | ⟨*Wort*⟩ `ge` ⟨*Wort*⟩
| ⟨*Wort*⟩ `in {` ⟨*Wortliste*⟩ `}`
| ⟨*Wort*⟩ `=~` ⟨*RegEx*⟩ | ⟨*Wort*⟩ `!~` ⟨*RegEx*⟩

⟨*Wortliste*⟩ ← ⟨*Wort*⟩ | ⟨*Wortliste*⟩ `,` ⟨*Wort*⟩

⟨*Wort*⟩ ← ⟨*Zahl*⟩ *Folge von `0...9`*
| ⟨*Zeichenkette*⟩ *Etwas in `"..."`*
| ⟨*Variable*⟩ | ⟨*Funktion*⟩

⟨*Variable*⟩ ← `%{` ⟨*VarName*⟩ `}`

⟨*Funktion*⟩ ← ⟨*FName*⟩ `(` ⟨*Argumente*⟩ `)`

Der ⟨*VarName*⟩ benennt eine der gültigen Umgebungsvariablen, die der Apache für Anfragen erzeugt oder eine der zusätzlichen SSL-Variablen in Tabelle 5.1. – Die einzige gültige Funktion, die Sie für ⟨*FName*⟩ angeben können, heißt `file`, übernimmt eine Zeichenkette als Argument und liefert den Inhalt der benannten Datei.

Bild 5.1: Syntax für `SSLRequire`-Ausdrücke

5.2.2 Zugriffsregeln

Wenn ein Browser ein gültiges Zertifikat geschickt hat und der Anwender so authentisiert wurde, kann der Server ihm Zugriffsrechte auf bestimmte Ressourcen einräumen oder versagen. Diese Zugriffsrechte werden – in Analogie zur `Require`-Direktive bei der Basic-Authentisierung – durch die `SSLRequire`-Direktive auf Verzeichnis- oder Dateiebene festgelegt. `SSLRequire` gestattet es, ausgefeilte Boolesche Ausdrücke anzugeben, die für jeden Zugriff ausgewertet werden. Die Syntax dieser Ausdrücke erinnert an eine Mischung der Programmiersprachen C und Perl und erlaubt den Rückgriff auf diverse SSL-Parameter sowie weitere Kriterien (Bild 5.1, Tabelle 5.1), die wir an ein paar Konfigurationsbeispielen illustrieren.

SSLRequire

Beispiele

Das folgende Beispiel erlaubt Benutzern den Zugriff zu einer Ressource, die entweder in der Marketing- oder der Entwicklungsabteilung der »Beispiel GmbH« arbeiten:

```
SSLRequire %{SSL_CLIENT_S_DN_O} eq "Beispiel GmbH" \
    && %{SSL_CLIENT_S_DN_OU} in { "Marketing", "Entwicklung" }
```

Das nächste Beispiel – wieder aus [MODSSL-DOC] – greift das Problem aus dem vorigen Abschnitt auf, wo bestimmte durch Zertifikate authentisierte Benutzer Zugriff auf einen besonderen Bereich der Web-Präsenz bekommen sollen, der Rest der Präsenz aber auch den anderen Benutzern zur Verfügung stehen soll:

```
SSLVerifyClient none
<Directory /srv/www/htdocs/members>
SSLVerifyClient require
SSLVerifyDepth  2
SSLCACertificateFile /etc/httpd/ssl.crt/ca.crt
```

Tabelle 5.1: Zusätzliche Umgebungsvariable für SSL-Anfragen

Name	Typ	Bedeutung
HTTPS	±	Wahr, wenn HTTPS-Anfrage, sonst falsch
SSL_PROTOCOL	$	Die verwendete Protokollversion (SSLv2, SSLv3, TLSv1
SSL_SESSION_ID	$	Die SSL-Sitzungs-ID (hexadezimal)
SSL_CIPHER	$	Die verwendeten Kryptoverfahren
SSL_CIPHER_EXPORT	$	Wahr, wenn die verwendeten Kryptoverfahren für den Export aus den USA geeignet sind (bzw. waren)
SSL_CIPHER_USEKEYSIZE	0	Anzahl von tatsächlich benutzten Bits im (symmetrischen) Schlüssel
SSL_CIPHER_ALGKEYSIZE	0	Anzahl von prinzipiell möglichen Bits im (symmetrischen) Schlüssel
SSL_VERSION_INTERFACE	$	Versionsnummer von mod_ssl
SSL_VERSION_LIBRARY	$	Versionsnummer von OpenSSL
SSL_CLIENT_M_VERSION	$	Versionsnummer des Client-Zertifikats
SSL_CLIENT_M_SERIAL	$	Seriennummer des Client-Zertifikats
SSL_CLIENT_S_DN	$	DN des Inhabers des Client-Zertifikats
SSL_CLIENT_S_DN_⟨*x509*⟩	$	Komponente des SSL_CLIENT_S_DN
SSL_CLIENT_I_DN	$	DN der Zertifizierungsstelle, die das Client-Zertifikat ausgestellt hat
SSL_CLIENT_I_DN_⟨*x509*⟩	$	Komponente des SSL_CLIENT_I_DN
SSL_CLIENT_V_START	$	Beginn des Gültigkeitszeitraums des Client-Zertifikats
SSL_CLIENT_V_END	$	Ende des Gültigkeitszeitraums des Client-Zertifikats
SSL_CLIENT_A_SIG	$	Zum Signieren des Client-Zertifikats benutzter Algorithmus
SSL_CLIENT_A_KEY	$	Für den öffentlichen Schlüssel des Client-Zertifikats benutzer Algorithmus
SSL_CLIENT_CERT	$	Das Client-Zertifikat (PEM-codiert)
SSL_CLIENT_CERT_CHAIN*r*	$	Das *r*-te Zertifikat in der Zertifizierungskette des Client-Zertifikats (PEM-codiert)
SSL_CLIENT_VERIFY	$	Ergebnis der Überprüfung des Client-Zertifikats: NONE, SUCCESS, GENEROUS oder FAILED:⟨*Grund*⟩

Datentypen: »±« = Boolescher Wert, »$« = Zeichenkette, »0« = Ganzzahl.
⟨*x509*⟩ ist der Name einer X.509-DN-Komponente (C, ST, L, O, OU, CN, T, I, G, S, D, UID, Email).
Die SSL_CLIENT_…-Variable existieren mit analoger Bedeutung auch als SSL_SERVER_…, ausgenommen SSL_CLIENT_CERT_CHAIN*r* und SSL_CLIENT_VERIFY.

```
SSLRequireSSL
SSLRequire %{SSL_CLIENT_S_DN_O} eq "Beispiel GmbH" \
    && %{SSL_CLIENT_S_DN_OU} in { "Marketing", "Entwicklung" }
```

Diese Vorgehensweise ist gegenüber der aus dem vorigen Abschnitt – mit der SSL-Option `FakeBasicAuth` und einer externen Benutzerdatenbank – vorzuziehen, wenn Sie es nicht mit *x*-beliebigen Benutzern zu tun haben, sondern mit solchen, deren Zertifikate eine gewisse gemeinsame Struktur aufweisen.

Intranet-Server

Zum Schluss ein komplizierteres Beispiel für den Zugriff auf einen Intranet-Server. Dieser Zugriff ist entweder über HTTP aus dem Intranet selbst oder – für einen bestimmten Bereich – über HTTPS mit starker Verschlüsselung und alternativ Client-Zertifikaten oder Basic-Authentisierung von beliebigen Rechnern auf dem Internet aus erlaubt. Wir nehmen an, dass das Intranet Adressen aus dem Netz `192.168.0.0/24` benutzt und dass der extern sichtbare Bereich auf dem Server URLs hat, die mit `/ext/` anfangen. Definieren Sie die folgende Konfiguration *außerhalb* Ihres virtuellen Servers für HTTPS, damit sie für HTTP *und* HTTPS gilt.

Außerhalb des `/ext/`-Bereichs auf dem Server ist Zugriff nur aus dem Intranet erlaubt:

```
SSLCACertificateFile /etc/httpd/ssl.crt/ca.crt

<Directory /srv/www/htdocs>
  Order deny,allow
  Deny  from all
  Allow from 192.168.0.0/24
</Directory>
```

Innerhalb von `/ext/` erlauben wir beliebige Zugriffe aus dem Intranet, aus dem Internet aber nur HTTPS mit starker Verschlüsselung und einem Kennwort oder einem Client-Zertifikat:

```
<Directory /srv/www/htdocs/ext>
  SSLVerifyClient optional
  SSLVerifyDepth  1
  SSLOptions      +FakeBasicAuth +StrictRequire
  SSLRequire      %{SSL_CIPHER_USEKEYSIZE} >= 128

  Satisfy any

  # Intranet ist OK
  Order deny,allow
  Deny  from all
  Allow from 192.168.0.0/24
```

Hier wird wieder die Authentisierungsmethode herangezogen, bei der die Zertifikatsinhaber-DNs als Basic-Authentisierungs-Benutzernamen interpretiert und in einer Kennwortdatei nachgeschlagen werden:

```
AuthType     Basic
AuthName     "Geschützter Intranet-Bereich"
AuthUserFile /etc/httpd/intra.passwd
Require      valid-user
```

mod_rewrite

Clients aus dem Internet werden gezwungen, HTTPS zu benutzen. `mod_rewrite` haben wir nicht weiter besprochen; die Idee ist, dass man mit `RewriteCond` eine Reihe von Bedingungen aufstellt, die zutreffen müssen, damit über die in `RewriteRule` beschriebene Umsetzung der gerade angefragte URL »umgeschrieben« wird. Das sehr simple Beispiel hier beantwortet jede Anfrage, auf die die `RewriteCond`-Bedingungen passen, mit dem HTTP-Rückgabewert `Forbidden`:

```
RewriteEngine on
RewriteCond   %{REMOTE_ADDR} !^192\.168\.0\.[0-9]+$
RewriteCond   %{HTTPS} != on
RewriteRule   .* - [F]
```

Die SSLRequireSSL-Direktive können wir hier nicht benutzen, da aus dem Intranet heraus auch normale HTTP-Zugriffe erlaubt sein sollen.

Übungen

5.2 [2!] Installieren Sie das Zertifikat Ihrer Zertifizierungsstelle in Ihrem Web-Server so, dass Sie damit die Authentizität von Client-Zertifikaten prüfen können, die (angeblich) von dieser Zertifizierungsstelle beglaubigt wurden.

5.3 [2!] Führen Sie auf Ihrem Web-Server einen Bereich ein, der nur nach erfolgreicher Authentisierung durch ein Client-Zertifikat angeschaut werden kann. Testen Sie dies.

5.4 [3] (Für Programmierer.) Schreiben Sie ein Shellskript, das, als CGI-Skript installiert, die SSL-Parameter – also die Umgebungsvariablen, die mit SSL_ anfangen – einer HTTP-Anfrage ausgibt.

5.5 [2] Geben Sie SSLRequire-Regeln an, die die folgenden Sachverhalte ausdrücken:

1. Der Inhaber des Zertifikats arbeitet in der Entwicklungsabteilung der »Beispiel GmbH«.
2. Das Zertifikat wurde von der Zertifizierungsstelle der »Beispiel GmbH« ausgestellt.
3. Der Zugriff erfolgt von einem Rechner im Netz 10.11.12.13/16 aus.
4. Der Zugriff erfolgt in der Adventszeit (1.–24.12.)

5.3 Zertifikats-Rückruflisten

5.3.1 Grundlagen

Zertifikat für ungültig erklären

Als Betreiber einer Zertifizierungsstelle kommen Sie möglicherweise an einen Punkt, wo Sie ein von Ihnen ausgestelltes Zertifikat für ungültig erklären wollen: Der Angestellte hat Ihre Firma verlassen, oder Sie haben beim Ausstellen des Zertifikats eine Unachtsamkeit begangen. Zertifikate laufen zwar früher oder später von selber ab, aber in der Regel können Sie darauf nicht warten; je kürzer die Gültigkeitsdauer Ihrer Zertifikate ist, desto mehr Arbeit haben Sie und Ihre Benutzer damit, ständig erneuerte Zertifikate zu erstellen, zu verteilen und zu installieren, und selbst wenn Ihre Zertifikate nur ein paar Wochen gälten, wären Sie in der Zwischenzeit potentiell angreifbar.

CRL

Die einzige Möglichkeit, ein existierendes Zertifikat vor Ablauf seiner Gültigkeitsdauer ungültig zu machen, besteht in der Veröffentlichung einer Zertifikats-Rückrufliste (engl. *certificate revocation list*, **CRL**), die das Zertifikat enthält. Institutionen wie Web-Server, die Authentisierung auf der Basis von Client-Zertifikaten vornehmen, können die aktuelle CRL konsultieren, um zu prüfen, ob das Zertifikat zwar formal gültig ist, aber auf der »schwarzen Liste« steht.

Theoretisch könnte man sich auch vorstellen, dass die Zertifizierungsstellen CRLs für *Server-Zertifikate* veröffentlichen. Das wird jedoch normalerweise nicht gemacht, und die gängigen Browser enthalten auch keine Infrastruktur dafür.

5.3.2 Erzeugung von Zertifikats-Rückruflisten

Zum Rückruf eines Zertifikats brauchen Sie eine Kopie des Zertifikats. Unsere Zertifizierungsstelle merkt sich alle ausgestellten Zertifikate im `certs`-Verzeichnis; das einzige Problem ist, dass wir die Seriennummer des Zertifikats kennen müssen, um die richtige Datei zu finden. Hierzu können Sie aber die Datei `index.txt` konsultieren, die unter anderem die DNs und Seriennummern der ausgestellten Zertifikate enthält.

Das eigentliche Zurückrufen geht sehr einfach mit dem OpenSSL-Werkzeug `ca`. Dazu müssen Sie natürlich die Konfiguration Ihrer Zertifizierungsstelle benutzen:

```
$ export OPENSSL_CONF=/usr/local/openssl/testCA/openssl.cnf
$ openssl ca -revoke certs/02.pem
Using configuration from /usr/local/openssl/testCA/openssl.cnf
Enter pass phrase for /usr/local/openssl/testCA/private/cakey.pem:blafasel
Revoking Certificate 02.
Data Base Updated
$ _
```

Für das Zurückrufen des Zertifikats ist die *passphrase* des privaten Schlüssels der Zertifizierungsstelle erforderlich. Der private Schlüssel wird zwar nicht zum Signieren gebraucht, aber zur Prüfung des zurückzurufenden Zertifikats und zur Authentisierung des Benutzers, der die Rückrufoperation vornehmen möchte. Anschließend ist in der Datenbank vermerkt, dass das Zertifikat zurückgerufen wurde und in künftigen CRLs erscheinen soll.

Das Zertifikat selbst wird wohlgemerkt nicht verändert. Das hätte auch gar keinen Zweck, da der Benutzer, dessen Zertifikat zurückgerufen werden soll, ja seine eigene Kopie davon hat, die wir nicht aus der Ferne ändern können. (Wenn das ginge, wäre die ganze Sache sowieso kein Problem …)

Die Rückrufliste selbst können Sie mit dem OpenSSL-Werkzeug `crl` erzeugen:

```
$ openssl ca -gencrl -out testca.crl
Using configuration from /usr/local/openssl/testCA/openssl.cnf
Enter pass phrase for /usr/local/openssl/testCA/private/cakey.pem:blafasel
$ _
```

Hier befindet die CRL sich anschließend in der Datei `testca.crl`. Bild 5.2 zeigt die CRL im Textformat. Beachten Sie, dass sowohl das Ausstellungsdatum der CRL als auch der vorgesehene Veröffentlichungszeitpunkt der nächsten Version der CRL angegeben ist; diesen haben wir mit dem `default_crl_days`-Parameter in der `openssl.cnf`-Datei unserer Zertifizierungsstelle angegeben.

5.3.3 Server-Konfiguration

CRL zum Server

Die CRL muss nun nur noch dem Apache-Server zur Verfügung gestellt werden, damit er sie bei der Authentisierung von Client-Zertifikaten zu Rate ziehen kann.

SSLCARevocationFile
SSLCARevocationPath

Die relevanten Konfigurationsdirektiven von `mod_ssl` sind hier `SSLCARevocationFile` und `SSLCARevocationPath`. Sie funktionieren im wesentlichen wie `SSLCACertificateFile` und `SSLCACertificatePath` – die erstere verweist auf eine Datei mit aneinandergehängten CRLs, die letztere auf ein Verzeichnis mit CRLs in einzelnen Dateien, die über *hash filenames* angesprochen werden. Auch hier gibt es ein `Makefile` zur Erstellung der *hash filenames*; sie hören zur Unterscheidung von denen für Zertifikate mit `.rN` auf. Die beiden Verfahren können wie zuvor alternativ oder gleichzeitig eingesetzt werden.

Während bei den CA-Zertifikaten beide Verfahren – Datei und Verzeichnis – ziemlich gleichwertig waren, spricht bei den CRLs einiges für die Verzeichnis-Methode, jedenfalls sobald Sie es mit CRLs verschiedener Zertifizierungsstellen zu tun haben: Da die verschiedenen Zertifizierungsstellen

```
$ openssl crl -in testca.crl -text -noout
Certificate Revocation List (CRL):
        Version 1 (0x0)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: /CN=Test-CA/L=Darmstadt/C=DE/emailAddress=ca@example.com/
          O=Zertifizierungsstelle
        Last Update: Nov 18 09:06:36 2003 GMT
        Next Update: Nov 25 09:06:36 2003 GMT
Revoked Certificates:
    Serial Number: 02
        Revocation Date: Nov 18 09:06:08 2003 GMT
    Signature Algorithm: md5WithRSAEncryption
        47:34:84:50:27:bf:ab:c3:cb:ec:28:8c:5a:2d:76:07:02:79:
        ◁◁◁◁◁
        56:b0:14:51:39:16:83:6d:07:9f:6a:d8:d0:e1:1a:26:31:27:
        14:a3:3b:b1
```

Bild 5.2: Eine CRL (Beispiel)

ihre CRLs vermutlich nicht alle gleichzeitig veröffentlichen werden, ist es viel einfacher, sie einzeln zu besorgen und zu installieren, wenn sie individuell in Dateien stehen, als wenn eine große Datei ständig geändert werden muss.

5.3.4 Verteilung von Zertifikats-Rückruflisten

OpenSSL hilft zwar beim Erstellen der Zertifikats-Rückruflisten, aber nicht bei ihrer Verteilung. Es ist Ihre Sache, dafür zu sorgen, dass Ihre CRLs auf allen relevanten Servern zur Verfügung stehen. Glücklicherweise ist das leicht zu erreichen, solange Sie nur einen oder wenige Server betreuen; Sie übertragen einfach die CRL-Datei von Ihrem CA-Rechner (möglicherweise per Diskette) auf den Web-Server und kopieren sie an den richtigen Platz.

Schwieriger wird es da, wo eine CRL wirklich weit verfügbar gemacht werden muss, etwa weil es sich um eine CRL für Server-Zertifikate einer öffentlichen Zertifizierungsstelle handelt. In [RFC2459] wird zwar ein Mechanismus zur Veröffentlichung von CRLs beschrieben, aber in der Praxis halten die wesentlichen Zertifizierungsstellen (allen voran VeriSign) sich nicht daran. Ansonsten gilt die Tanenbaumsche Maxime »Das Schöne an Standards ist, dass man unter so vielen wählen kann« [Tan02]. Ein weiteres Problem ist, dass CRLs nicht wirklich helfen, wenn das selbstsignierte Zertifikat einer Zertifizierungsstelle kompromittiert wurde.

Im Interim sollten Sie CRLs wohl am besten über HTTPS zugänglich machen – wir haben Ihnen ja erklärt, wie das geht. Die Benutzer der CRLs können dann automatisierte Mechanismen installieren, um sich immer die aktuellen CRLs zu besorgen.

Übungen

5.6 [2!] Erzeugen Sie ein neues Client-Zertifikat und rufen Sie es zurück. Erzeugen Sie anschliessend eine CRL und vergewissern Sie sich, dass das Zertifikat darin auftaucht.

5.7 [2!] Installieren Sie die CRL in Ihrem Web-Server und prüfen Sie, ob das zurückgerufene Zertifikat noch von ihm akzeptiert wird oder nicht.

Zusammenfassung

- Client-Zertifikate werden im wesentlichen erzeugt wie Server-Zertifikate. Für den Import in Browser wandelt man sie ins PKCS#12-Format um.
- Zur Authentisierung von Zugriffen über Client-Zertifikate prüft Apache zunächst die Authentizität des Zertifikats und ordnet dem Zugriff dann anhand der Identität des Zertifikatsinhabers und anderer Kriterien Rechte zu.
- Eine Zertifizierungsstelle kann einmal erstellte Zertifikate über Zertifikats-Rückruflisten (CRLs) für ungültig erklären.

Literaturverzeichnis

MODSSL-DOC Ralf S. Engelschall. »Apache SSL/TLS Encryption«.
`http://httpd.apache.org/docs/2.2/ssl/`

RFC2459 R. Housley, W. Ford, W. Polk, et al. »Internet X.509 Public Key Infrastructure Certificate and CRL Profile«, Januar 1999.
`http://www.ietf.org/rfc/rfc2459.txt`

Tan02 Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 2002, 3. Auflage. Unbedingt auf englisch lesen, die deutsche Übersetzung ist eine Katastrophe.

VMC02 John Viega, Matt Messier, Pravir Chandra. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly & Associates, 2002. ISBN 0-596-00270-X.
`http://www.oreilly.com/catalog/openssl/`

LINUP FRONT

A

Apache-Direktiven

Hier sind die in dieser Schulungsunterlage besprochenen Apache-Direktiven kurz zusammengefasst. Für jede Direktive gibt es eine Informationszeile und eine kurze Erklärung. Die Informationszeile gibt die Syntax der Direktive und (in Klammern) einen etwaigen Standardwert wieder; außerdem führt sie das Modul auf, das die Direktive definiert (sofern sie nicht Standardbestandteil des Apache ist) und gibt den Kontext an, in dem die Direktive auftauchen darf. Hierbei steht »G« für die »globale« Ebene der Konfigurationsdatei, »V« für `<VirtualHost>`-Blöcke, »B« für `<Directory>`-, `<Files>`- und `<Location>`-Blöcke und »D« für `.htaccess`-Dateien. Am Ende der Erklärung steht ein Verweis auf die Seite in der Schulungsunterlage, die die ausführliche Erläuterung der Direktive enthält.

`SSLCACertificateFile` ⟨*Dateiname*⟩ `mod_ssl` *GV*
Die benannte Datei enthält die Zertifikate von Zertifizierungsstellen zur Prüfung von Client-Zertifikaten. 68

`SSLCACertificatePath` ⟨*Verzeichnisname*⟩ `mod_ssl` *GV*
Das benannte Verzeichnis enthält die Zertifikate von Zertifizierungsstellen zur Prüfung von Client-Zertifikaten. Die Zertifikate werden über *hash filenames* mit der Endung `.N` gefunden. 68

`SSLCARevocationFile` ⟨*Dateiname*⟩ `mod_ssl` *GV*
Die benannte Datei enthält Zertifikats-Rückruflisten, die bei der Prüfung von Client-Zertifikaten beachtet werden. 74

`SSLCARevocationPath` ⟨*Verzeichnisname*⟩ `mod_ssl` *GV*
Das benannte Verzeichnis enthält Zertifikats-Rückruflisten, die bei der Prüfung von Client-Zertifikaten beachtet werden. Die Rückruflisten werden über *hash filenames* mit der Endung `.rN` gefunden. 74

`SSLCertificateChainFile` ⟨*Datei*⟩ `mod_ssl` *GV*
In dieser Datei steht die komplette »Kette« des Zertifikats des Servers.
56

`SSLCertificateFile` ⟨*Datei*⟩ `mod_ssl` *GV*
In dieser Datei steht das Zertifikat des Servers.
56

`SSLCertificateKeyFile` ⟨*Datei*⟩ `mod_ssl` *GV*
In dieser Datei steht der private Schlüssel des Servers.
56

`SSLCipherSuite` ⟨*Schlüsselliste*⟩ (`Siehe unten`) `mod_ssl` *GVBD*
Gibt die akzeptablen SSL-Verschlüsselungsverfahren an. Details stehen in der Dokumentation zu `mod_ssl`. Die Voreinstellung (die oben nicht passt) ist `ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP`.
34

`SSLEngine on|off|optional` `off` `mod_ssl` *GV*
Schaltet SSL für den betreffenden (globalen oder virtuellen) Server ein oder aus.
33

`SSLOptions` [+|-]⟨*Option*⟩ … `mod_ssl` *GVBD*
Erlaubt die Konfiguration diverser SSL-Optionen. 59

`SSLProtocol` [+|-]⟨*Protokoll*⟩ … (`all`) `mod_ssl` *GV*
Gibt an, welche SSL-Protokollversionen akzeptabel sind.
34

`SSLRandomSeed` ⟨*Kontext*⟩ ⟨*Quelle*⟩ [⟨*Bytes*⟩] `mod_ssl` *G*
Gibt die Quelle(n) für Zufallszahlen an.
34

`SSLRequire` ⟨*Bedingung*⟩ `mod_ssl` *BD*
Stellt Bedingungen für Client-Zertifikate auf. 69

`SSLRequireSSL` `mod_ssl` *BD*
Lehnt Zugriffe ab, die nicht über SSL erfolgen. 59

`SSLVerifyClient` ⟨*Stufe*⟩ `none` `mod_ssl` *GVBD*
Gibt an, ob und mit welchem Nachdruck Apache ein Client-Zertifikat anfordert. 67

`SSLVerifyDepth` ⟨*Tiefe*⟩ `1` `mod_ssl` *GVBD*
Gibt die maximale akzeptable Tiefe der Zertifikatskettenprüfung für Client-Zertifikate an. 67

B

Musterlösungen

Dieser Anhang enthält Musterlösungen für ausgewählte Aufgaben.

1.2 Ein simpler Ansatz:

```
#!/bin/sh

msg="$*"
for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
do
    echo $msg | tr A-Z $i-ZA-$i
done
```

(»Überstehende« Buchstaben im zweiten Argument von `tr` werden ignoriert.)

1.3 Bevor Sie jetzt damit loslegen, 58216819 durch 3, 5 (OK, das vielleicht nicht) und so weiter zu dividieren und auf ein glattes Ergebnis zu warten, fangen Sie lieber oben an, nämlich bei $\sqrt{58216819} = 7629{,}994692\ldots$. Das sieht verdächtig glatt aus; wenn Sie clever sind und uns keine allzugroße Gemeinheit unterstellen wollen, dann berechnen Sie mal interessehalber $7630^2 = 58216900$ und stellen fest, dass das genau $9^2 = 81$ mehr ist als unsere Zahl. Nach der dritten binomischen Formel ist $58216819 = 7630^2 - 9^2 = (7630 + 9)(7630 - 9) = 7639 \cdot 7621$, und – bingo! – Sie haben Ihre Primfaktoren.

Oder seien Sie faul, gehen Sie zu `www.wolframalpha.com` und geben Sie

```
factors of 58216819
```

ein.

1.4 Verwenden Sie etwas wie

```
$ T="Das Pferd frisst keinen Gurkensalat"
$ for i in md5 sha1 sha256; do echo $T | ${i}sum; done
```

Die Prüfsummen sollten wie folgt aussehen:

MD5	`078cc11a7e61b4fb04415ebaa1bda154`
SHA-1	`a37c684ee6e69cec58cb33af61c49a7e7c44acfd`
SHA-256	`3854d66566d9edfa0e6ab47a0f077da232608fd4` `6c8d1f8845290dc5e7f90bae`

1.5 Wenn Sie nur $H(g, N)$ bestimmen, kann ein Angreifer mit $H(g, N)$ als »Zwischenergebnis« einen für die neue Nachricht stimmenden MAC $H(g, N, N')$ ausrechnen (er muss ja nur mit seinen eigenen Daten weitermachen). Bei $H(g, H(g, N))$ als MAC ist das nicht möglich, weil der MAC nicht mehr als »Zwischenergebnis« dienen kann.

1.6 $H(g, N)$ ist `8543f9f9dcc985df280729aba41963a9` und $H(g, H(g, N))$ ist `bc6bc19da4c8b6bd5fc64abdaa0b`

1.7 POP3-Server, die das `APOP`-Kommando unterstützen, liefern in ihrer Begrüßungsnachricht eine eindeutige Zeichenkette, typischerweise in der Form

<⟨*Prozess-ID*⟩.⟨*Uhrzeit*⟩`@pop.example.com>`

Der Client nimmt diese Zeichenkette und hängt ein geheimes Kennwort an, das nur er und der Server kennen. Das Resultat wird mit MD5 bearbeitet und das Ergebnis dieser Operation an den Server geschickt. Der Server vollzieht dieselbe Operation nach (er weiß, was er dem Client als Begrüßung geschickt hat, und kennt auch das geheime Kennwort); stimmt seine Prüfsumme mit der vom Client geschickten überein, wird der Client akzeptiert.

Der Hauptvorteil des Ansatzes ist natürlich, dass im Gegensatz zum gewöhnlichen POP keine Kennwörter im Klartext über das Netz geschickt werden müssen. Der wesentliche Nachteil ist, dass das Kennwort des Clients im Klartext auf dem Server bekannt sein muss. Dies ist ein Problem, wenn der Server kompromittiert wird.

2.3 HTTP ist so konstruiert, dass auf eine Anfrage immer genau eine Antwort kommt. Für ein `STARTTLS`-Kommando ist in diesem Konzept nicht wirklich Platz. (Eigentlich stimmt das nicht: HTTP sieht durchaus die Möglichkeit vor, etwas wie `STARTTLS` zu integrieren. Allerdings würde das bedeuten, dass zum Abrufen von Ressourcen von einem HTTPS-Server für jede Ressource zwei »Rundreisen« vom Client zum Server gemacht werden müssten und nicht eine – unabhängig vom Überhang der SSL-Aushandlung. Das ist natürlich ineffizient.) (Tatsächlich unterstützen gute Web-Server wie zum Beispiel Apache einen »SSL-Sitzungscache«, um die Wiederverwendung von mühsam ausgehandelten SSL-Verbindungen zu ermöglichen. Das hat aber mit dem Protokoll HTTP als solchem nur am Rande zu tun.)

2.6 Die Verfügbarkeit von OpenSSL macht es möglich, die Software von unabhängigen Experten prüfen zu lassen. So lässt sich leichter sicherstellen, dass die kryptografischen Verfahren und ihre Implementierung korrekt sind. Fehler lassen sich schneller finden und reparieren. Außerdem ist SSL ein wichtiger Bestandteil der Infrastruktur und sollte nicht von einer einzigen Firma kontrolliert werden. Die Freiheit der Software garantiert ihr Fortbestehen, solange sie für die Nutzergemeinde interessant ist.

3.5 Das Zertifikat bekommen Sie normalerweise mit Ihrem Browser geliefert. Sie können die Authentizität aber prüfen, indem Sie den »Fingerabdruck« (engl. *fingerprint*) des Zertifikats bilden und mit dem »offiziellen« vergleichen. Den Fingerabdruck eines Zertifikats erhalten Sie mit

```
$ openssl x509 -in cacert.pem -noout -fingerprint
```

Nach dem offiziellen Fingerabdruck müssen Sie die Zertifizierungsstelle fragen – am besten per Telefon.

4.2 Der Browser beschwert sich, dass die IP-Adresse des Servers nicht zu der des Zertifikats passt. Das ist nicht weiter überraschend, denn das Zertifikat ist auf `www.example.com` ausgestellt, und der Server ist `localhost`, mithin `127.0.0.1`, was aller Wahrscheinlichkeit nach nicht auch die Adresse von `www.example.com` ist. Theoretisch bräuchten Sie dafür ein weiteres Zertifikat, aber da die wenigsten Web-Server HTTPS-Verbindungen zu sich selbst unterstützen müssen, macht man sich die Mühe in der Praxis nicht.

5.2 Benutzen Sie die Optionen `SSLCACertificateFile` oder `SSLCACertificatePath`. Im letzteren Fall müssen Sie noch den korrekten *hash filename* anlegen.

5.4 Im einfachsten Fall etwas wie

```
#!/bin/sh
echo 'Content-Type: text/plain'
echo ''
printenv | grep '^SSL_' | sort
```

Achten Sie darauf, dass dieses Skript keine Programmaufrufe von Sachen abhängig macht, die der Client kontrolliert. Ein Aufruf wie

```
echo 'SSL_CLIENT_I_DN:' $SSL_CLIENT_I_DN
```

ist potentiell gefährlich.

5.5 Zum Beispiel:

1. Der Inhaber des Zertifikats arbeitet in der Entwicklungsabteilung der »Beispiel GmbH«:

   ```
   SSLRequire %{SSL_CLIENT_S_DN_O} eq "Beispiel GmbH" \
              && %{SSL_CLIENT_S_DN_OU} eq "Entwicklung"
   ```

2. Das Zertifikat wurde von der Zertifizierungsstelle der »Beispiel GmbH« ausgestellt:

   ```
   SSLRequire %{SSL_CLIENT_I_DN_O} eq "Beispiel GmbH" \
              && %{SSL_CLIENT_I_DN_OU} eq "Zertifizierungsstelle"
   ```

3. Der Zugriff erfolgt von einem Rechner im Netz 10.11.12.0/24 aus:

   ```
   SSLRequire %{REMOTE_ADDR} =~ m/^10\.11\.12\.[0-9]+$/
   ```

4. Der Zugriff erfolgt in der Adventszeit (1.–24.12.):

   ```
   SSLRequire %{TIME_DAY} <= 24 && %{TIME_MONTH} == 12
   ```

Index

Dieser Index verweist auf die wichtigsten Stichwörter in der Schulungsunterlage. Besonders wichtige Stellen für die einzelnen Stichwörter sind durch **fette** Seitenzahlen gekennzeichnet. Sortiert wird nur nach den Buchstaben im Indexeintrag; „`~/.bashrc`“ wird also unter „B“ eingeordnet.