TLS/SSL Kochbuch

Rezepte für die Verwendung von OpenSSL, HTTP Strict Transport Security (HSTS) und HTTP Public Key Pinning (HPKP)

Jörg Kastning

9. August 2016

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 Akteure . . . 2
2.2 Verschlüsselung . . . 2
2.2.1 Symmetrische Verschlüsselung . . . 3
2.2.2 Asymmetrische Verschlüsselung . . . 4
2.3 TLS/SSL . . . 5
2.4 PKI (Public Key Infrastructure) . . . 7
2.4.1 Zertifikate . . . 8
2.4.2 Zertifikatsketten . . . 9
2.5 Schwachstelle im Design . . . 10
2.6 OpenSSL . . . 11
2.7 HTTP Strict Transport Security (HSTS) . . . 13
2.8 Public Key Pinning Extension for HTTP (HPKP) . . . 14

3 Implementierung von TLS/SSL **16**
3.1 Generierung eines CSR mit OpenSSL . . . 16
3.1.1 Generierung eines privaten Schlüssels . . . 16
3.1.2 Erstellung eines CSR . . . 18
3.2 Skriptgesteuerte Generierung von Private Key und CSR . . . 20
3.3 Certificate Signing Request (CSR) für mehrere Hostnamen erstellen . . . 22
3.3.1 Erstellung eines CSR für mehrere Hostnamen unter Verwendung von OpenSSL . . . 24
3.3.2 Erstellung eines CSR für mehrere Hostnamen unter Verwendung von make-cert.sh . . . 24
3.4 Zertifizierungsstellen . . . 25
3.4.1 Let's Encrypt . . . 25
3.4.2 StartSSL . . . 26
3.4.3 Thawte . . . 27
3.4.4 CAcert . . . 27
3.4.5 SSL-Zertifikate von Hosting-Providern . . . 28
3.5 Implementierung der Zertifikate im Zielsystem . . . 28
3.5.1 Apache . . . 29
3.5.2 NGINX . . . 30
3.5.3 lighttpd . . . 31
3.5.4 Microsoft IIS . . . 31
3.5.5 Postfix . . . 31
3.5.6 Dovecot . . . 32
3.6 Implementierung von Let's Encrypt . . . 32
3.6.1 Ausschließliche Nutzung von Let's Encrypt . . . 32

3.6.2 Nutzung von Let's Encrypt mit einem eigenen CSR . . 34
3.7 Erneuerung von Zertifikaten 35

4 Implementierung von HPKP 38
4.1 Vorüberlegungen 38
4.2 Backupstrategie 39
4.3 Die PINs berechnen 39
4.4 Konfiguration von NGINX 40
4.5 Validierung der Backupstrategie 40
4.5.1 Laborumgebung 40
4.5.2 Test der Backupstrategie 41
4.5.3 Gegenprobe 42
4.6 Fazit 43

5 Fallbeispiel: Let's Encrypt und HPKP 45
5.1 Konfiguration von NGINX für die Test-Domain 45
5.2 Erstellung des CSR mit OpenSSL 46
5.3 Beantragung und Implementierung des TLS-Zertifikats mit HSTS und HPKP 48
5.4 Verlängerung des TLS-Zertifikats mittels SmartRenew 50

6 Schlussworte 52

Abbildungsverzeichnis III

Listings IV

Literatur V

1 Einleitung

Heutzutage werden immer mehr Kommunikationsverbindungen im Internet mit TLS/SSL-Verschlüsselung geschützt. Die Verschlüsselung hilft, die Vertraulichkeit der zwischen Sender und Empfänger übertragenen Daten zu schützen und sollte daher standardmäßig aktiviert sein. Doch bereitet der Einsatz von TLS/SSL-Verschlüsselung immer noch vielen Betreibern von Webseiten, Web-, Mail- und Chat-Servern Kopfschmerzen.

Zu undurchsichtig scheint der Dschungel aus Zertifizierungsstellen, Zertifikaten, Zertifikatsanfragen, öffentlichen und privaten Schlüsseln. Verschiedenste Validierungsverfahren und Dateiformate für Zertifikate tragen nicht gerade dazu bei, den Durchblick zu behalten. Bereits die Erstellung einer Zertifikatsanfrage gerät dabei häufig genug zu einem Problem. Die richtige Konfiguration der zu sichernden Server bzw. Dienste erscheint kompliziert und Fehler in der Konfiguration führen nicht selten zur Nichterreichbarkeit einer Webseite. Die Folge: Immer noch wird viel zu häufig auf den Einsatz von TLS/SSL-Verschlüsselung verzichtet.

Diese Rezeptsammlung möchte dazu beitragen, etwas Licht ins Dunkel zu bringen und praktische Tipps für die Verwendung von TLS/SSL-Verschlüsselung mittels OpenSSL, HTTP Strict Transport Security (HSTS) und HTTP Public Key Pinning (HPKP) geben.

Kapitel 2 führt in das Thema ein und definiert die für diesen Text wesentlichen Begriffe. Es bildet die Grundlage für das Verständnis der daran anschießenden Kapitel.

In Kapitel 3 geht es um die Implementierung von TLS/SSL. Hier werden verschiedene Methoden zur Generierung von privaten Schlüsseln und Certificate Signing Requests vorgestellt. Darüber hinaus werden einige Zertifizierungsstellen kurz vorgestellt, bei denen Zertifikate beantragt werden können. Im Folgenden wird auf die Implementierung von TLS/SSL-Zertifikaten in verschiedenen Diensten eingegangen. Hier wird insbesondere die Implementierung von Zertifikaten der recht jungen Zertifizierungsstelle „Let's Encrypt“ erläutert, mit deren Hilfe sich der Prozess der Zertifikatserneuerung automatisieren lässt.

Der Implementierung des HTTP Public Key Pinning (HPKP) widmet sich Kapitel 4. Das Pinning-Verfahren wird am Beispiel des Webservers NGINX erläutert und getestet.

Zum Schluss werden die in den einzelnen Kapiteln vorgestellten Techniken und Methoden nochmals zusammenfassend an einem konkreten Beispiel in Kapitel 5 verdeutlicht.

2 Grundlagen

Das Leben ist voller Missverständnisse. Nur weil verschiedene Personen die gleichen Begriffe verwenden, bedeutet dies nicht, dass sie damit auch das Gleiche meinen.

> When I use a word, it means just what I choose it to mean – neither more nor less.

– Humpty Dumpty (in *Through the Looking Glass*)

Leider werden viele der in diesem Text verwendeten Begriffe in der Literatur und im Alltag teilweise mit unterschiedlicher Bedeutung benutzt. Um Missverständnisse zu vermeiden, werden diese Begriffe kurz erläutert und ihre Verwendung im folgenden Text definiert.

2.1 Akteure

Alice und *Bob*[1] werden als Synonyme für Sender und Empfänger einer Nachricht verwendet: Alice sendet eine Nachricht an Bob. Neben Alice und Bob werden noch folgende Akteure in diesem Text eine Rolle spielen:

Eve, von engl. *eavesdropper* (zu deutsch Lauscher/Lauscherin), ist passiver Angreifer. Sie versucht, ausgetauschte Nachrichten mitzuhören (und auch zu verstehen). Sie kann die übertragenen Nachrichten jedoch nicht verändern oder in die Systeme von Sender und Empfänger eindringen.

Marvin bezeichnet einen aktiven Angreifer. Dieser kann aktiv in das Geschehen eingreifen, indem er z. B. Nachrichten fälscht oder zu übertragende Nachrichten verändert.

Trudy, von engl. intruder (Eindringling), wird in ähnlicher Rolle wie Marvin gebraucht, im Unterschied zu diesem jedoch mit Fokus auf das Eindringen in ein bestehendes System.

2.2 Verschlüsselung

Durch die in diesem Text beschriebenen Verschlüsselungsverfahren sollen in Bezug auf eine Kommunikationsverbindung im Wesentlichen die drei folgenden Ziele erreicht werden:

[1] `https://de.wikipedia.org/wiki/Alice_und_Bob` [letzter Abruf: 14.05.2016]

Vertraulichkeit Nur direkt an der Kommunikation beteiligte Sender und Empfänger haben Zugriff auf den Inhalt einer Nachricht. Unbeteiligte Dritte können den Inhalt der Nachricht hingegen nicht lesen.

Integrität Durch entsprechende Verfahren kann überprüft werden, ob eine Nachricht auf dem Übertragungsweg verändert wurde. Dadurch wird die Nachricht vor Fälschung durch Dritte auf dem Übertragungsweg geschützt.

Authentizität Der Absender einer Nachricht kann überprüft werden. Es soll damit verhindert werden, dass Marvin Nachrichten im Namen von Alice versendet.

Dabei werden im Allgemeinen symmetrische und asymmetrische Verschlüsselungsverfahren unterschieden.

2.2.1 Symmetrische Verschlüsselung

Symmetrische Verschlüsselung beschreibt ein Verfahren, bei dem zur Ver- und Entschlüsselung von Daten der gleiche Schlüssel verwendet wird, vgl. hierzu [13, S. 47] sowie [20, S. 5]. Mit einem solchen Schlüssel kann Alice bspw. ihre Daten verschlüsseln, um sie vor den neugierigen Augen Eves zu schützen. Damit nun niemand außer Alice die Daten entschlüsseln und lesen kann, muss Alice den Schlüssel geheim halten. Daher wird dieses Verfahren auch als *secret key* Verschlüsselung bezeichnet.

Möchte Alice nach diesem Verfahren Nachrichten mit Bob austauschen, müssen beide über den gleichen Schlüssel verfügen, um Nachrichten ver- und entschlüsseln zu können. Die Kenntnis des geheimen Schlüssels stellt ein gemeinsames Geheimnis von Alice und Bob dar. Dieses Geheimnis wird auch als *shared secret* bezeichnet.

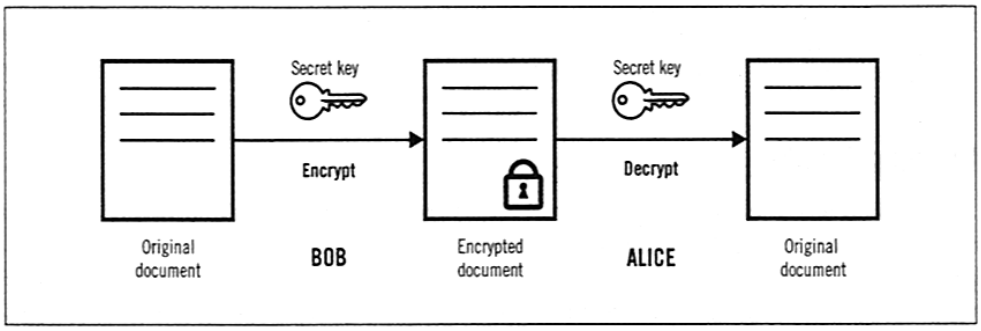

Abbildung 2.1: Symmetrische Verschlüsselung [20, S. 6]

In Abb. 2.1 verwendet Bob einen geheimen Schlüssel, um eine Nachricht an Alice zu verschlüsseln. Alice verwendet den gleichen geheimen Schlüssel, um die Nachricht wieder zu entschlüsseln. Hört Eve die Nachricht während der Übertragung ab, sieht sie nur den verschlüsselten Text und nicht die ursprüngliche Nachricht. Die Kommunikation von Alice und Bob bleibt so lange vertraulich, wie der verwendete Schlüssel geheim bleibt [siehe 20, S. 5].

Die Schwierigkeit bei der symmetrischen Verschlüsselung besteht darin, dass Bob vor Beginn des Nachrichtenaustauschs Alice den geheimen Schlüssel mitteilen

muss. Hierbei besteht das Risiko, dass der geheime Schlüssel bei der Übertragung von Eve abgehört werden kann. Eve könnte damit fortan die Kommunikation zwischen Alice und Bob mithören, mit Hilfe des geheimen Schlüssels entschlüsseln und damit auf den ursprünglichen Nachrichtentext zugreifen.

2.2.2 Asymmetrische Verschlüsselung

Symmetrische Verschlüsselung leistet gute Arbeit bei der Verarbeitung großer Datenmengen mit hoher Geschwindigkeit. Doch bleibt das Problem des sicheren Schlüsselaustauschs. Wie in [20, S. 12] ausgeführt wird, wächst dieses Problem mit der Menge der Kommunikationsteilnehmer. Denn alle Teilnehmer müssen über den gleichen geheimen Schlüssel verfügen. Je mehr Teilnehmer Kenntnis über den geheimen Schlüssel haben, desto größer ist das Risiko der Kompromittierung.

Dieses Problem versucht die asymmetrische Verschlüsselung (welche auch als *public key* Verschlüsselung bezeichnet wird) zu lösen, indem für die Ver- und Entschlüsselung unterschiedliche Schlüssel verwendet werden. Es existiert ein öffentlicher Schlüssel, welcher zur Verschlüsselung genutzt und frei verteilt werden kann. Die mit diesem öffentlichen Schlüssel verschlüsselten Daten können, wie in Abb. 2.2 dargestellt, ausschließlich mit dem dazugehörigen privaten Schlüssel wieder entschlüsselt werden [vgl. 20, S. 12-13]. Darüber hinaus kann mit dem privaten Schlüssel eine Signatur erstellt werden, mit deren Hilfe die Authentizität einer Nachricht überprüft werden kann. Dieses Verfahren wird in Abb. 2.3 dargestellt, vgl. auch [20, S. 13-14] sowie [13, S. 51].

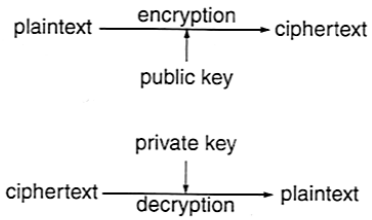

Abbildung 2.2: Asymmetrische Verschlüsselung [13, S. 51]

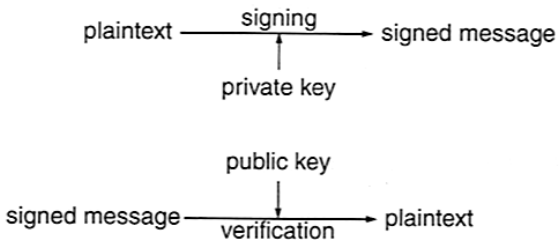

Abbildung 2.3: Digitale Signatur [13, S. 51]

Die asymmetrische Verschlüsselung findet z. B. Anwendung in OpenPGP [1] und S/MIME [19].

Da der zur Verschlüsselung benötigte Schlüssel nicht geheim gehalten werden muss, kann dieser auf vielfältige Weise z. B. auf Webseiten, im E-Mail-Anhang oder über Schlüsselserver veröffentlicht werden.

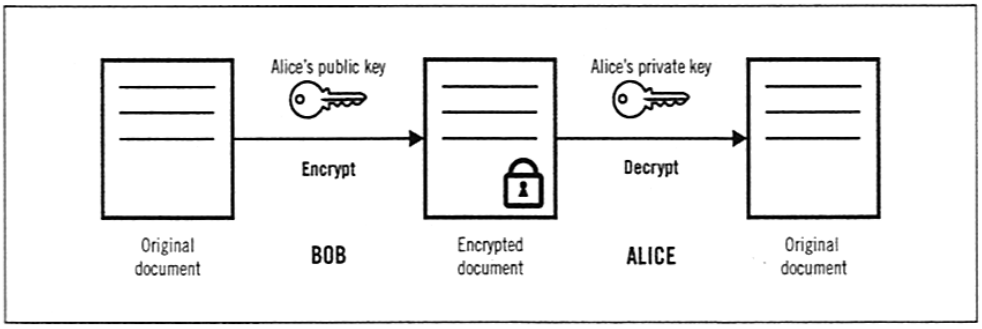

Abbildung 2.4: Asymmetrische Verschlüsselung beim Nachrichtenversand von Bob an Alice [20, S. 13]

Abbildung 2.4 zeigt, wie Bob eine Nachricht mit dem öffentlichen Schlüssel von Alice verschlüsselt. Alice kann diese Nachricht ausschließlich mit ihrem dazugehörigen privaten Schlüssel wieder entschlüsseln. Hört Eve die Kommunikation ab, kann sie die Nachricht nicht entschlüsseln, da sie den zugehörigen privaten Schlüssel nicht besitzt.

Wenn Bob die Nachricht an Alice zusätzlich noch mit seinem eigenen privaten Schlüssel signiert hat, kann Alice mit Bobs öffentlichen Schlüssel überprüfen, ob die Nachricht tatsächlich von Bob stammt oder auf dem Übertragungsweg evtl. von Marvin verändert wurde.

Wie in [13, Abschnitt 2.5.1] beschrieben wird, kann asymmetrische Verschlüsselung für die gleichen Zwecke wie symmetrische Verschlüsselung verwendet werden. Das Verfahren ist jedoch wesentlich langsamer als symmetrische Verschlüsselungsverfahren. Daher kommen in der Praxis häufig hybride Verfahren zum Einsatz. Dabei verwendet Alice z. B. den öffentlichen Schlüssel von Bob, um einen symmetrischen Schlüssel sicher an Bob zu übertragen. Damit Bob verifizieren kann, dass ihm der symmetrische Schlüssel auch wirklich von Alice gesendet wurde, kann Alice diesen zusätzlich mit ihrem privaten Schlüssel signieren. Bob kann den symmetrischen Schlüssel mit seinem privaten Schlüssel entschlüsseln. Anschließend nutzen Alice und Bob für die weitere Kommunikation den symmetrischen Sitzungsschlüssel.

2.3 TLS/SSL

SSL (*secure socket layer*) und TLS (*transport socket layer*) werden häufig synonym verwendet. Um Missverständnisse so weit wie möglich zu vermeiden, bin ich bemüht, diese Begriffe wie folgt zu verwenden:

- SSL wird verwendet, wenn sich eine Aussage auf SSLv3 [8] oder älter bezieht,
- TLS wenn von TLS 1.1 oder 1.2 [3] die Rede ist und
- TLS/SSL wenn eine Aussage im Wesentlichen auf beide Protokolle zutrifft.

TLS/SSL-Verschlüsselung wird verwendet, um eine vertrauliche Kommunikation über ein unsicheres Medium zu ermöglichen. So kann Alice mit Bob über das Internet kommunizieren, während hinreichend sichergestellt ist, dass sie tatsächlich

mit Bob kommuniziert und die Kommunikation nicht von Eve mitgelesen werden kann [20, S. 1]. Zu Beginn der Verbindung zwischen Client und Server wird ein Handshake wie in Abb. 2.5 durchgeführt.

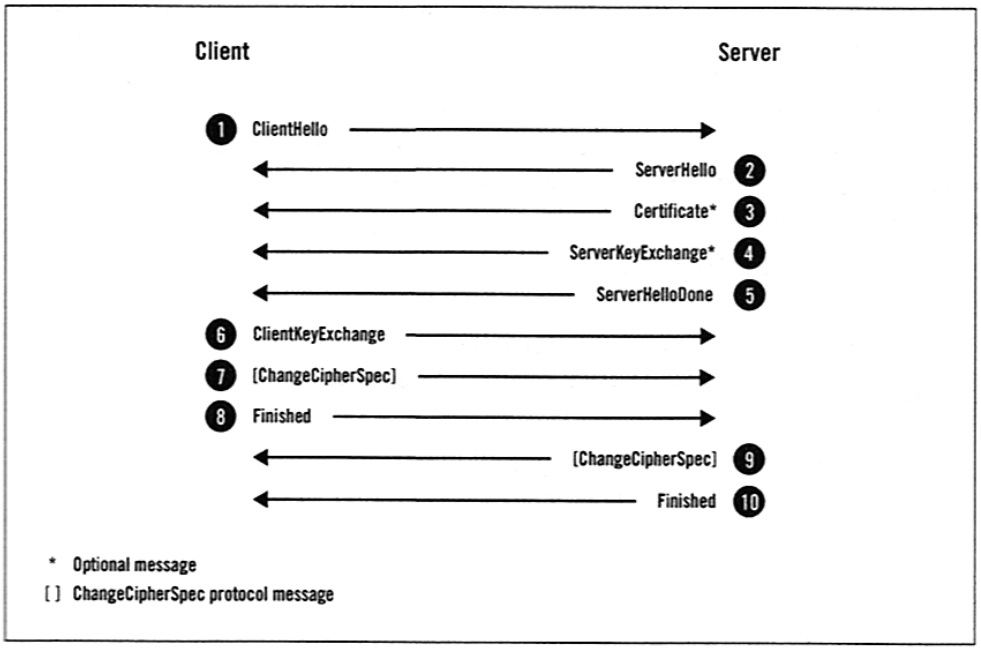

Abbildung 2.5: TLS Handshake mit Server-Authentifizierung [20, S. 27]

1. Der Client beginnt den Handshake und übermittelt die von ihm unterstützten Ciphers an den Server.
2. Der Server wählt die Verbindungsparameter aus.
3. Der Server übermittelt zur Authentifizierung sein Zertifikat, welches den öffentlichen Schlüssel enthält.
4. Abhängig vom gewählten Schlüsselaustausch-Verfahren übermittelt der Server weitere Informationen zur Berechnung des Sitzungsschlüssels.
5. Der Server signalisiert, dass alle notwendigen Informationen übermittelt wurden.
6. Der Client übermittelt weitere zur Berechnung des Sitzungs-Schlüssels benötigte Informationen.
7. Der Client informiert den Server, dass er mit der verschlüsselten Kommunikation beginnt.
8. Der Client signalisiert, dass der Handshake erfolgreich durchgeführt wurde.
9. Der Server informiert den Client, dass die Kommunikation nun verschlüsselt erfolgt.
10. Der Server signalisiert, dass der Handshake erfolgreich durchgeführt wurde.

Das in Schritt 3 übermittelte Server-Zertifikat enthält neben dem öffentlichen Schlüssel Informationen, anhand derer der Client die Authentizität des Servers überprüfen kann. Dazu wurde das Server-Zertifikat von einer sogenannten „vertrauenswürdigen Zertifizierungsstelle“ (engl. *trusted Certificate Authority* (CA)) signiert. Verfügt der Client über einen öffentlichen Schlüssel der Zertifizierungsstelle, welche das Server-Zertifikat signiert hat, kann er die Authentizität des Servers überprüfen. Andernfalls wird eine Warnung ausgegeben. Die Meldung lautet dabei in der Regel, dass es sich um ein nicht vertrauenswürdiges Zertifikat handelt, da die Authentizität nicht überprüft werden kann. Der Anwender hat dann meist die Wahl, ob er den Verbindungsvorgang dennoch fortsetzen oder abbrechen möchte.

2.4 PKI (Public Key Infrastructure)

Eine PKI (*public key infrastructure*) bietet alle notwendigen Komponenten, um öffentliche Schlüssel zu verteilen und zu verwalten. Idealerweise setzt sich eine PKI aus Zertifikaten, einem Verzeichnis zum Download von Zertifikaten sowie Methoden zum Validieren und Widerrufen von Zertifikaten zusammen. Eine allgemeine Beschreibung von PKI findet sich in [13, Kap. 15].

Die in diesem Text beschriebenen Verfahren und Methoden basieren auf der Internet X.509 PKI, wie sie in [2] spezifiziert ist und wie sie Ivan Ristić in [20, Kap. 3] beschreibt. In dieser existieren die folgenden Rollen:

Subscriber Entität, welche einen gesicherten Service anbieten möchte und hierfür ein Zertifikat benötigt.

Certification authority (CA) Die Zertifizierungsstelle, welche die Zertifikate ausstellt, die Identität des Subscriber bestätigt und denen wir vertrauen. Sie stellt außerdem Informationen über widerrufene Zertifikate zur Verfügung, so dass *relying parties* die Gültigkeit von Zertifikaten überprüfen können.

Registration authority (RA) Übernimmt die Aufgabe, die Identität eines Subscribers zu validieren. Diese Aufgabe wird in der Praxis häufig von den CAs übernommen.

Relying party Die *relying party* bezeichnet z. B. Webbrowser oder andere Programme, welche Zertifikate nutzen und versuchen deren, Echtheit zu validieren. Dies geschieht mit Hilfe von sogenannten *trust anchors*, also Zertifikaten, denen ultimativ vertraut wird und die in den Webbrowsern gespeichert sind.

Wenn ich in diesem Text von Vertrauen spreche, ist dies im technischen Sinne gemeint. So vertraut ein Webbrowser den Zertifikaten, welche von einer CA in seinem Zertifikatsspeicher signiert wurden. Dies hat jedoch nichts mit dem wahren Vertrauen zu tun, das ich als Person in ein Zertifikat bzw. eine Zertifizierungsstelle habe oder auch nicht.

Doch wie kommt man eigentlich zu einem Zertifikat? Dazu sind im Wesentlichen drei Schritte notwendig:

1. Generierung eines starken privaten Schlüssels,
2. Erstellung eines *Certificate Signing Requests (CSR)* (deutsch Zertifikatsanfrage), Übermittlung des CSR an eine CA und
3. Installation des von der CA ausgestellten Zertifikats im Webserver.

2.4.1 Zertifikate

Ein Zertifikat ist eine Datei, welche den öffentlichen Schlüssel, Informationen zum Besitzer des Zertifikats und eine digitale Signatur einer CA enthält [20, S. 66]. Es soll dazu dienen, die Authentizität und Integrität einer Domain, Person oder Organisation zu bestätigen. Dazu umfasst ein Zertifikat u. a. folgende Felder (vgl. [18] und [20, S. 67f.]):

Issuer Informationen über die CA, welche das Zertifikat signiert hat.

Validity Zeitraum für den das Zertifikat gültig ist. Es enthält Start- und Enddatum.

Subject Name des Hosts[2], welcher mit dem öffentlichen Schlüssel verknüpft ist, für den das Zertifikat ausgestellt werden soll.

Public key Öffentlicher Schlüssel des Zertifikats.

Welche Prozesse durchlaufen werden, um die Identität eines Subscribers zu überprüfen, hängt zum einen von der ausstellenden CA ab und zum anderen davon, welchem Typ das beantragte Zertifikat entspricht [20, S. 74-75]:

Domain validated (DV) Zertifikate werden ausgestellt, wenn die Kontrolle über die Domain nachgewiesen werden kann. Dies geschieht häufig, indem eine Bestätigungs-E-Mail an eine E-Mail-Adresse der Domain gesendet wird. Darüber hinaus existieren noch weitere Methoden, wie z. B. das Setzen eines speziellen DNS-TXT-Records oder die Bereitstellung einer Datei in einem bestimmten Pfad des Webservers.

Organisation validated (OV) Zertifikate erfordern eine Identitäts- und Authentizitäts-Verifizierung der beantragenden Organisation. Die dazu notwendigen Prozesse unterscheiden sich teilweise deutlich von CA zu CA.

Extended validation (EV) Zertifikate sollen durch einen besonders aufwendigen Validierungsprozess ein entsprechend hohes Vertrauen in ein Zertifikat erzeugen. Die Adresszeile eines Webbrowsers wird bei Verwendung eines EV-Zertifikats grün dargestellt und zeigt meist zusätzlich den Namen des Subscribers an. Für diesen Aufwand muss der Subscriber allerdings auch deutlich tiefer in die Tasche greifen.

Es ist jedoch umstritten, ob EV-Zertifikate tatsächlich ein höheres Sicherheitsniveau bieten oder eher unter kommerziellen Gesichtspunkten eingeführt wurden.

[2] z. B. www.example.com

So sieht der DFN-Verein (Deutsches Forschungsnetz) aktuell von der Ausstellung von EV-Zertifikaten ab, bis deren sicherheitstechnischer Mehrwert beurteilt werden kann.[3]

Nach erfolgreicher Validierung wird das Zertifikat ausgestellt. Neben dem Zertifikat selbst werden dem Subscriber zumeist auch noch benötigte Zwischenzertifikate (engl. *intermediate certificates*) zur Verfügung gestellt, um eine vollständige Zertifikatskette bereitstellen zu können.

2.4.2 Zertifikatsketten

In der Praxis verwenden CAs meist ein Zwischenzertifikat, um Zertifikate von Antragstellern zu signieren. Die Gründe dafür erläutert Ristić in [20, S. 71f.]. Für den Dienstbetreiber bedeutet dies, dass er nicht nur sein Zertifikat, sondern die komplette Zertifikatskette ausliefern muss, die zum Root-Zertifikat der CA führt, welchem die Webbrowser der Clients vertrauen. Abbildung 2.6 soll dies verdeutlichen.

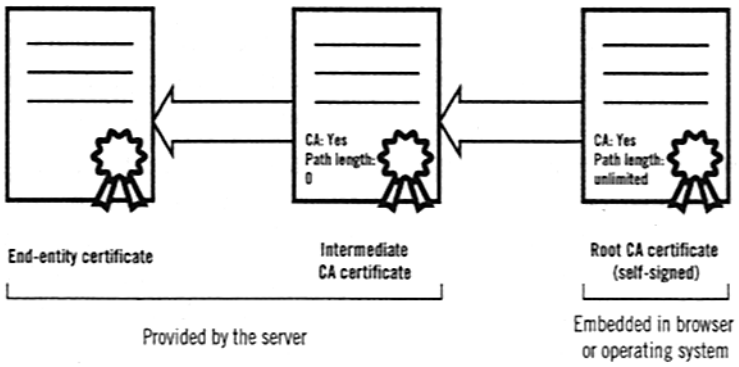

Abbildung 2.6: Struktur einer Zertifikatskette [20, S. 71]

Es wird vorausgesetzt, dass Server die komplette Zertifikatskette bereitstellen. Dies ist jedoch durch menschliche Konfigurationsfehler häufig nicht der Fall, weshalb die Bereitstellung häufig eines der größten Probleme in der Implementierung von TLS/SSL darstellt. Um Fehler im späteren Produktivbetrieb zu vermeiden, ist diesem Punkt besondere Aufmerksamkeit zu widmen und die Dokumentation des verwendeten Servers zu Rate zu ziehen.

Ganz allgemein ist eine Zertifikatskette eine Textdatei, welche die codierten Zertifikate beinhaltet, die in ihrer Gesamtheit die Zertifikatskette bilden. Listing 2.1 auf der nächsten Seite[4] soll dies verdeutlichen. Es zeigt die komplette Zertifikatskette der HRZ Uni Bielefeld CA. Die Zertifikate werden zur Verbesserung der Übersichtlichkeit gekürzt ausgegeben.

[3] `https://www.pki.dfn.de/faqpki/faqpki-allgemein/#c15073` [Letzter Abruf am 15.05.2016]

[4] `https://pki.pca.dfn.de/uni-bielefeld-ca/pub/cacert/chain.txt`

Listing 2.1: Zertifikatskette der HRZ Uni Bielefeld CA

```
SHA-2 chain, PCA Jul 14
subject= /C=DE/O=Universitaet Bielefeld/CN=CA der Universitaet Bielefeld - G02/
    emailAddress=hrz-ra@uni-bielefeld.de
-----BEGIN CERTIFICATE-----
MIIFZjCCBE6gAwIBAgIHF5Bg0ljWAzANBgkqhkiG9w0BAQsFADBaMQswCQYDVQQG
EwJERTETMBEGA1UEChMKREZOLVZlcmVpbjEQMA4GA1UECxMHREZOLVBLSTEkMCIG
A1UEAxMbREZOLVZlcmVpbiBQQ0EgR2xvYmFsIC0gRzAxMB4XDTE0MDUxMjE1MDUz
4ymLtIYO4fadaIvhizyzOan3AkpbC/CT2uGmNhetvTz00/HF+6V1qQuJ
-----END CERTIFICATE-----
subject= /C=DE/O=DFN-Verein/OU=DFN-PKI/CN=DFN-Verein PCA Global - G01
-----BEGIN CERTIFICATE-----
MIIE1TCCA72gAwIBAgIIUE7G9T0RtGQwDQYJKoZIhvcNAQELBQAwcTELMAkGA1UE
BhMCREUxHDAaBgNVBAoTE0RldXRzY2hlIFRlbGVrb20gQUcxHzAdBgNVBAsTFlQt
VGVsZVNlYyBUcnVzdCBDZW50ZXIxIzAhBgNVBAMTGkRldXRzY2hlIFRlbGVrb20g
fbV7pQLxJMUkYxE0zFqTICp5iDolQpCpZTt8htMSFSMp/CzazDlbVBc=
-----END CERTIFICATE-----
subject= /C=DE/O=Deutsche Telekom AG/OU=T-TeleSec Trust Center/CN=Deutsche
    Telekom Root CA 2
-----BEGIN CERTIFICATE-----
MIIDnzCCAoegAwIBAgIBJjANBgkqhkiG9w0BAQUFADBxMQswCQYDVQQGEwJERTEc
MBoGA1UEChMTRGV1dHNjaGUgVGVsZWtvbSBBRzEfMB0GA1UECxMWVC1UZWxlU2Vj
IFRydXN0IENlbnRlcjEjMCEGA1UEAxMaRGV1dHNjaGUgVGVsZWtvbSBSb290IENB
Cm26OWMohpLzGITY+9HPBVZkVw==
-----END CERTIFICATE-----
```

Die Textdatei enthält alle Zertifikate bis zum Wurzelzertifikat der Telekom, beginnend mit der HRZ Uni Bielefeld CA. Um welches Zertifikat es sich handelt, wird mit dem Schlüsselwort subject angegeben. Es können jedoch nicht alle Anwendungen damit umgehen, wenn diese Zeilen in der Datei mit der Zertifikatskette enthalten sind. In diesen Fällen dürfen einzig und allein die Blöcke mit den Zertifikaten in der Datei enthalten sein. An dieser Stelle muss auf die Dokumentation der jeweiligen Anwendung verwiesen werden.

2.5 Schwachstelle im Design

TLS/SSL-Verbindungen sollen die vertrauliche Kommunikation zwischen zwei Kommunikationspartnern sicherstellen. Bei den Kommunikationspartnern handelt es sich typischerweise um einen Client und einen Server. Die Sicherheit der Vertraulichkeit beruht darauf, dass der Client auch tatsächlich mit dem richtigen Server verbunden ist. Dazu weist sich der Server mit einem Zertifikat aus, welches von einer Zertifizierungsstelle (Certificate Authority, CA) ausgestellt wurde. Die Zertifizierungsstelle prüft die Identität des Dienstbetreibers und beglaubigt mit ihrer digitalen Signatur das Zertifikat.

Die c't schreibt in ihrem Artikel [23, S. 118, Spalte 2-3]: "Das Problem dabei: Es gibt weit über hundert solcher Zertifizierungsstellen, denen die gängigen Internet-Programme wie Browser und E-Mail-Client vertrauen. Als wäre das nicht unübersichtlich genug, haben die dann auch noch zahllose Unter-CAs, die berechtigt und in der Lage sind, im Namen dieser CAs zu unterschreiben."

Das Problem besteht nun nicht in der Anzahl der Zertifizierungsstellen, sondern darin, dass jede dieser Zertifizierungsstellen Zertifikate für beliebige Domains ausstellen darf. So können sich Dritte ein Zertifikat auf eine bereits existierende Domain ausstellen lassen und dieses zum Beispiel für einen Man-In-The-Middle-Angriff (MITM-Angriff)[5] verwenden.

[5] `https://de.wikipedia.org/wiki/Man-in-the-middle` [Letzter Abruf: 16.05.2016]

Das genannte Angriffsszenario existiert nicht nur in der Theorie. Wie die c't in [23, S. 119, Spalte 1] berichtet, nutzte die chinesische Regierung gefälschte Google-Zertifikate auf ihrer großen Firewall, um den Google-Mail-Verkehr ihrer Bevölkerung überwachen zu können. Darüber hinaus werden weitere Fälle wie der Hack der niederländischen Zertifizierungsstelle DigiNotar und TrustWave aufgeführt. Im ersten Fall wurden gefälschte Zertifikate für GMail und Facebook ausgestellt. Im zweiten Fall wurden Zertifikate für Firmen ausgestellt, welche diese Zertifikate nutzten, um den verschlüsselten Internet-Verkehr ihrer Mitarbeiter zu überwachen. In [20, Kap. 4, „Attacks against PKI"] finden sich weitere Beispiele für erfolgreiche Angriffe auf Internet PKIs.

Die Man-In-The-Middle-Angriffe funktionieren, da die Zertifizierungsstellen, welche die gefälschten Zertifikate ausgestellt haben, in der Liste der vertrauenswürdigen Zertifizierungsstellen der gängigen Browser geführt werden. Für den Browser ist damit auch das gefälschte Zertifikat gültig. Der Benutzer kann den MITM-Angriff nicht erkennen und nimmt fälschlicherweise an, direkt mit dem gewünschten Server zu kommunizieren.

Um dieser Schwäche im Design zu begegnen kann die *Public Key Pinning Extension for HTTP* (HPKP) genutzt werden, welche in diesem Text in Abschnitt 2.8 auf Seite 14 beschrieben wird.

2.6 OpenSSL

OpenSSL ist ein Open Source Projekt und das Schweizer Taschenmesser der TLS/SSL-Verschlüsselung. Von der Projekt-Webseite[6]:

> OpenSSL is an open source project that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library.

Kurz gesagt, wer TLS/SSL-Verschlüsselung einsetzen möchte, kommt an OpenSSL nur schwer vorbei. Leider ist das Projekt nicht besonders gut dokumentiert. Deswegen werde ich hier etwas ausführlicher auf die grundlegenden Funktionen eingehen.

Die Einsatzmöglichkeiten des OpenSSL-Toolkit werden in [21] umfassend beschrieben. Das E-Book „OpenSSL Cookbook" kann nach einer kostenlosen Registrierung auf der Webseite[7] in verschiedenen Formaten heruntergeladen oder online gelesen werden.

Dieser Abschnitt erläutert, wie man OpenSSL für sein Betriebssystem beziehen bzw. die Version einer installierten Version abrufen kann. Denn eine funktionsfähige OpenSSL-Installation ist Voraussetzung, um die beschriebenen Schritte nachvollziehen und ausführen zu können.

Wer ein Linux- bzw. ein Unix-System nutzt, hat in vielen Fällen Glück. Denn hier ist OpenSSL bereits installiert. Sollte dies wider Erwarten nicht der Fall sein, finden Benutzer dieser Betriebssysteme die aktuelle Version zum Download auf der

[6] `https://www.openssl.org/` [Letzter Abruf am 15.05.2016]

[7] `https://www.feistyduck.com/library/openssl-cookbook/` [Letzter Abruf am 15.05.2015

Projekt-Webseite.[8] Benutzer von Microsoft Windows finden Installationsdateien für ihr Betriebssystem auf den Webseiten von Shining Light Productions[9].

Um sich die installierte Version von OpenSSL anzeigen zu lassen, führt man in einer Kommandozeile den Befehl `openssl version` aus. Dies liefert auf einem Ubuntu GNOME 14.04 LTS, welches ich für die Beispiele in diesem Text verwende, folgende Ausgabe:

```
$ openssl version
OpenSSL 1.0.1f 6 Jan 2014
```

Zum Zeitpunkt der Erstellung dieses Textes trägt die aktuelle OpenSSL Version die Nummer 1.0.1t. Jedoch ist die in Ubuntu 14.04 LTS enthaltene Version nicht hoffnungslos veraltet. Etliche Betriebssysteme verändern den Code eines Pakets, um z. B. Sicherheitslücken zu schließen, ohne dessen Bezeichnung zu ändern. Zu erkennen ist dies, wenn man sich die Versionshistorie des Pakets anzeigen lässt:

Listing 2.2: OpenSSL Changelog

```
1  $ apt-get changelog openssl
2  openssl (1.0.1f-1ubuntu2.19) trusty-security; urgency=medium
3
4    * SECURITY UPDATE: EVP_EncodeUpdate overflow
5      - debian/patches/CVE-2016-2105.patch: properly check lengths
6        in crypto/evp/encode.c, add documentation to
7        doc/crypto/EVP_EncodeInit.pod, doc/crypto/evp.pod.
8      - CVE-2016-2105
9    * SECURITY UPDATE: EVP_EncryptUpdate overflow
10     - debian/patches/CVE-2016-2106.patch: fix overflow in
11       crypto/evp/evp_enc.c.
12     - CVE-2016-2106
13   * SECURITY UPDATE: Padding oracle in AES-NI CBC MAC check
14     - debian/patches/CVE-2016-2107.patch: check that there are
15       enough padding characters in
16       crypto/evp/e_aes_cbc_hmac_sha1.c.
17     - CVE-2016-2107
18   * SECURITY UPDATE: Memory corruption in the ASN.1 encoder
19     - debian/patches/CVE-2016-2108-1.patch: don't mishandle zero
20       if it is marked as negative in crypto/asn1/a_int.c.
21     - debian/patches/CVE-2016-2108-2.patch: fix ASN1_INTEGER
22       handling in crypto/asn1/a_type.c, crypto/asn1/asn1.h,
23       crypto/asn1/tasn_dec.c, crypto/asn1/tasn_enc.c.
24     - CVE-2016-2108
25   * SECURITY UPDATE: ASN.1 BIO excessive memory allocation
26     - debian/patches/CVE-2016-2109.patch: properly handle large
27       amounts of data in crypto/asn1/a_d2i_fp.c.
28     - CVE-2016-2109
29   * debian/patches/min_1024_dh_size.patch: change minimum DH size
30     from 768 to 1024.
31
32  -- Marc Deslauriers <marc.deslauriers@ubuntu.com>  Thu, 28 Apr 2016 11:22:20
       -0400
```

Die Ausgabe in Listing 2.2 lässt erkennen, dass zuletzt am 28.04.2016 Sicherheitsupdates eingepflegt wurden. Es wird dringend empfohlen, die OpenSSL-Installation stets aktuell zu halten. Nur so kann sichergestellt werden, dass die verwendete Version nicht mehr anfällig gegen bereits bekannte Schwachstellen ist.

[8] `https://www.openssl.org/source/` [Letzter Abruf am 15.05.2016]
[9] `http://slproweb.com/products/Win32OpenSSL.html` [Letzter Abruf am 15.05.2016]

Um nun mit Hilfe von OpenSSL zu einem TLS/SSL-Zertifikat zu kommen, sind im Wesentlichen drei Schritte notwendig:

1. Generierung eines starken privaten Schlüssels,
2. Erstellung eines *Certificate Signing Requests (CSR)*, dessen Übermittlung an eine CA und
3. Installation des von der CA ausgestellten Zertifikats im Webserver.

In den Abschnitten 3.1.1 auf Seite 16 und 3.1.2 auf Seite 18 wird die grundlegende Bedienung von OpenSSL beschrieben. Es sei dabei erwähnt, dass OpenSSL unter Linux, Unix und Windows die gleiche Syntax verwendet. Die gezeigten Beispiele können so gleichermaßen auf den genannten Betriebssystemen ausgeführt werden.

2.7 HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security (HSTS) ist in RFC 6797 [11] spezifiziert und beinhaltet einen Mechanismus, mit dem Webseiten ihre ausschließliche Erreichbarkeit über sichere Verbindungen deklarieren können. Dadurch soll sichergestellt werden, dass User-Agents (Webbrowser) nur über gesicherte Verbindungen mit einer Webseite kommunizieren.

HTTP sieht keinen Mechanismus vor, dem Webbrowser mitzuteilen, ob TLS auf einer Webseite implementiert ist. Wird nur der Hostname (ohne das Protokoll) angegeben, baut der Webbrowser von Alice standardmäßig eine ungesicherte Verbindung zur Webseite von Bob auf. Diese Verbindung kann leicht von Eve mitgehört werden.

Leider sind es Benutzer heute immer noch gewohnt, dass sie in ihrem Webbrowser Warnmeldungen zu sehen bekommen, die auf nicht vertrauenswürdige Zertifikate hinweisen. Die Webbrowser erlauben es jedoch, die Warnung zu ignorieren und die Webseite trotzdem aufzurufen. Ignoriert Alice die Warnung, setzt sie sich damit dem Risiko aus, Opfer einer Man-In-The-Middle-Attacke von Marvin zu werden.

HSTS adressiert diese Schwachstelle durch zwei Mechanismen (siehe [11, Abschnitt 2.2] und [20, S. 286]):

1. HTTP-URLs werden für Alice transparent umgeschrieben, so dass HTTPS-URLs verwendet werden und
2. alle Zertifikatsfehler werden als schwerwiegend eingestuft und können von Alice nicht mehr ignoriert werden.

Der RFC 6797 beschreibt darüber hinaus weitere Bedrohungsszenarien in [11, Abschnitt 2.3].

Implementiert wird HSTS über ein zusätzliches Feld im HTTP-Header, welches den `Strict-Transport-Security`-Header enthält. Es folgt ein Beispiel für einen HSTS-Header, welcher einen Webbrowser anweist, HSTS für einen Zeitraum von 365 Tagen für den Hostnamen und sämtliche Sub-Domains zu aktivieren:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Versucht Marvin nun die Verbindung von Alice zu Bob auf eine unverschlüsselte Verbindung umzulenken oder präsentiert Alice ein selbstsigniertes Zertifikat, blockiert der Webbrowser von Alice den Verbindungsaufbau.

Dabei ist zu beachten, dass der HSTS-Header nur über HTTPS-Verbindungen ausgeliefert wird. Andernfalls könnte der Header für Denial-of-Service-Angriffe auf Webseiten missbraucht werden, welche nur über HTTP-Verbindungen erreichbar sind.

Um dieses Szenario zu verdeutlichen, nehmen wir an, Alice kommuniziert via HTTP mit der Webseite von Bob. Diese hat kein TLS implementiert. Nun beginnt Marvin mit einer Man-In-The-Middle-Attacke und verändert die Antwort von Bob so, dass sie einen HSTS-Header enthält. Alice würde damit angewiesen, Bobs Webseite nur noch über HTTPS-URLs aufzurufen. Ihr Browser schreibt die URL ja für Alice transparent auf HTTPS um. Da Bob jedoch gar kein TLS implementiert hat, erhält Alice nun statt der erhofften Antwort nur eine Fehlermeldung und kann nicht mehr auf Bobs Webseite zugreifen.

Hat Bob TLS auf seiner Webseite implementiert und möchte sicherstellen, dass Alice die HSTS-Informationen erhält, sollte Bob sämtlichen Traffic auf Port 80 an Port 443 umleiten.

HSTS trägt folglich dazu bei, die Angriffsfläche zu verkleinern, indem Schwachstellen in TLS abgeschwächt werden.

2.8 Public Key Pinning Extension for HTTP (HPKP)

Public Key Pinning Extension for HTTP (HPKP) alias *Certificate Pinning* wurde im RFC 7469 [7] spezifiziert. Es handelt sich dabei um einen Mechanismus, der es ermöglicht, bestimmte Eigenschaften eines Zertifikats „festzunageln“. Der Browser erhält dabei mit dem Zertifikat zusätzliche Informationen, mit denen er die Authentizität des Zertifikats überprüfen kann.

In der Praxis wird dem Browser ein Hash-Wert übermittelt, welcher über den öffentlichen Schlüssel des Zertifikats gebildet wurde. Dieser Hash-Wert wird in einem HTTP-Header-Feld an den Browser übertragen und von diesem gespeichert. Wie dieses Verfahren im Detail funktioniert, beschreibt [7, Abschnitt 2.1.1]. Da nur der Dienstbetreiber über den dazugehörigen privaten Schlüssel verfügt, ist ein Missbrauch durch Dritte ausgeschlossen. Übermittelt nun nämlich ein Angreifer bei einem MITM-Angriff den zu seinem Zertifikat gehörenden öffentlichen Schlüssel, kann der Browser mit Hilfe des PINs erkennen, dass dieser Schlüssel nicht dem Dienstbetreiber gehört und den Angriff damit erkennen.

Eine Schwachstelle bleibt jedoch. Damit das beschriebene Verfahren funktioniert, muss die erste Verbindung, die ein Client zum Server aufbaut, integer und vertrauenswürdig sein. Findet bereits beim ersten Verbindungsaufbau ein MITM-Angriff statt, kann der Angreifer einen zu seinem öffentlichen Schlüssel passenden Hash übermitteln und das beschriebene Verfahren damit aushebeln [vgl. 23, S. 121, Kasten „TOFU: Trust On First Use“].

Es existiert also ein Henne-Ei-Problem, da man zuerst eine sichere Verbindung benötigt, um die erforderlichen Informationen zu erhalten, mit denen zukünftige Verbindungen gesichert werden können. Dies betrifft sowohl HSTS als auch HPKP. Denn beide Verfahren sind darauf angewiesen, dass bei der ersten Verbindung die korrekten Informationen zur Sicherung zukünftiger Verbindungen übermittelt werden können.

Mittlerweile sollte jedoch bekannt sein, dass es keine hundertprozentige Sicherheit gibt. Damit ist der beschriebene Schutz besser als gar keiner. In der Praxis können HSTS und HPKP daher in sehr vielen Fällen helfen, die Sicherheit vertraulicher Kommunikation zu steigern.

3 Implementierung von TLS/SSL

Das vorliegende Kapitel widmet sich der Implementierung von TLS/SSL. Es beschreibt verschiedene Verfahren, mit denen Certificate Signing Requests erstellt werden können. Im Anschluss daran gibt Abschnitt 3.4 auf Seite 25 eine kurze Einführung in die Welt der CAs und stellt einige zufällig ausgewählte Zertifizierungsstellen kurz vor.

Abschnitt 3.5 auf Seite 28 zeigt einige Beispiele für die Implementierung von TLS für verschiedene Dienste. Bei den beschriebenen Diensten befindet sich jeweils ein Verweis auf die offizielle Dokumentation mit weiteren Hinweisen zur Implementierung und Konfiguration des jeweiligen Dienstes.

Der Implementierung von *Let's Encrypt*-Zertifikaten (siehe 3.4.1 auf Seite 25) ist mit 3.6 auf Seite 32 ein eigener Abschnitt gewidmet, um die Besonderheiten dieser noch recht jungen CA besser herausstellen zu können.

3.1 Generierung eines CSR mit OpenSSL

In diesem Abschnitt werden die erforderlichen Schritte beschrieben, um einen CSR mit Hilfe von OpenSSL zu erstellen. Dabei wird zuerst erläutert, wie ein CSR für einen Hostnamen erstellt wird. Am Ende des Abschnitts wird dann erklärt, wie man einen CSR für ein Zertifikat generiert, welches mehrere Hostnamen beinhalten kann.

Die in diesem Abschnitt beschriebenen Methoden lassen sich gleichermaßen unter Windows, Linux und Unix ausführen. Voraussetzung ist lediglich eine funktionsfähige OpenSSL-Installation (vgl. 2.6 auf Seite 11).

3.1.1 Generierung eines privaten Schlüssels

Ristić weist in [21, S. 8] darauf hin, dass vor der Erstellung eines privaten Schlüssels Entscheidungen hinsichtlich Schlüssel-Algorithmus, Schlüssellänge und Passphrase getroffen werden müssen.

OpenSSL unterstützt RSA-, DSA- und ECDA-Schlüssel. Doch nicht alle Algorithmen sind für jedes Szenario gleich gut geeignet. Für Webserver werden zum Zeitpunkt der Erstellung dieses Textes hauptsächlich RSA-Schlüssel verwendet. Diese bieten eine Schlüssellänge von bis zu 4096 Bits. Zudem werden diese Schlüssel von den meisten CAs unterstützt.

Da die von OpenSSL verwendete Standard-Schlüssellänge unter Umständen zu kurz ist, sollten Sie diese immer explizit angeben. Heutzutage gilt eine Schlüssellänge von 2048 Bits für einen RSA-Schlüssel als sicher. Diese Schlüssellänge sollte

demnach mindestens verwendet werden. Eine Schlüssellänge von 4096 Bits bietet ein noch höheres Maß an Sicherheit, erhöht jedoch auch die Zeit, welche die CPU für die Berechnungen mit Schlüsseln dieser Länge benötigt.

Die Verwendung einer Passphrase ist optional. Sie wird verwendet, um den privaten Schlüssel vor unberechtigtem Zugriff zu schützen. Wird eine Passphrase verwendet, muss diese eingegeben werden, bevor auf den privaten Schlüssel zugegriffen werden kann. Dies ist jedoch ein zweischneidiges Schwert. Während eine Passphrase zum Schutz des privaten Schlüssels beiträgt, ist die Verwendung solcher Schlüssel auf Webservern in den meisten Fällen eher unpraktisch. Denn der Webserver würde bei jedem Start die Passphrase abfragen, bevor er den privaten Schlüssel lesen kann. Auch trägt die Passphrase in diesem Fall nicht zur Steigerung der Sicherheit bei. Eine im Klartext hinterlegte oder im Arbeitsspeicher befindliche Passphrase kann durch einen aktiven Angreifer wie Trudy gestohlen werden. Trudy wäre so in der Lage, sowohl den privaten Schlüssel als auch die Passphrase zu entwenden. Das Risiko ist damit genauso hoch, als würde Trudy einen ungeschützten privaten Schlüssel erbeuten.

In anderen Worten: Eine Passphrase ist gut und nützlich, um einen privaten Schlüssel zu schützen, wenn er nicht gerade auf einem Produktiv-System installiert ist. Für den Einsatz auf einem Produktiv-System empfiehlt es sich, die Passphrase vom privaten Schlüssel zu entfernen. Die einzige Ausnahme bildet der Einsatz spezieller Hardware-Appliances, welche über spezielle Techniken verfügen, um private Schlüssel im Speicher zu schützen.

Die Generierung eines privaten Schlüssels inkl. Passphrase gelingt mit dem Kommando `genrsa`:

Listing 3.1: Generierung des privaten Schlüssels

```
/tmp$ openssl genrsa -aes256 -out test.key 2048
Generating RSA private key, 2048 bit long modulus
..............................+++
.............................................+++
e is 65537 (0x10001)
Enter pass phrase for test.key:
Verifying - Enter pass phrase for test.key:
```

Hier wurde ein RSA-Schlüssel mit 2048 Bits Länge erstellt, welcher mit einer AES-256-Passphrase geschützt ist.

Private Schlüssel werden im sogenannten PEM-Format gespeichert, welches mit einem gewöhnlichen Text-Editor betrachtet werden kann:

Listing 3.2: Ausgabe des privaten Schlüssels (verschlüsselt)

```
/tmp$ cat test.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,EF34D969278DD4E052701D59FFDC614B

/+U4g7TkBiwJVLoPuCk6t4rU1V/jtBxLmq3csY1KSjCPeTsSygVDdUIAe/qcyqBh
FA7hW2J1tNJoYbiuA0WSqcxTVJ/eb3kzh90dPd0B+coBD+Fmu9t1cUmHe6jhavpa
ANYDDbDacm27KVBgWPicrHfu2X212NIH+Aze7VBgodw+h6NWQ2COgjODF1iQI4Sp
7nxcqaEt1hSzdJL8YrApO6aNBULtzcaxoGHdtxbT/buJs3Q/i9s3N0Fg5K2I1ewH
8w6pGHnRXBBd6K27N2dMi2o3PQNWOnIoZGCvJJvS/MXbPt3GvCyoR5i5GijXysNs
RMMIMg3MhE+8MB6flNFh4yosNQMjdTppnmUyxoa6YPNrniWwUPqy/ka/iXCll5IM
5O/17AB4SzKOyQ8PSXGP8bM/hbtKar0MQmxUSgbfwa3vigQq6Bnvgy+NxIXQbga2
eeEXWRd9JW1V+d1qZ5OTyBD8SlpyTHk+OEntsJEjYV6PkXg5J9bKOnhP9Z1/gmX0
yko+PbIZLO80CsibQj0EkygMB9nE999mNN8cn6GU0eg018m4KB8fQA8AAywCmhhb
9x3CO4BrsWZcTtoEmLLfEMB2EkpX7I11Q6+TurAayBh6RIrANRweIFRa3v2DoU3l
xFhqpW2kdoMxP4U4xvTyigYxJeqmPeIT02kugeVlfGB63gyU2bhoF2uqnsgScs8F
glKA3YCULll6sPRgoIOLgOcgLp6lNe8zle+bB/3dX+BT3COg0ZMLusopOarnk1CJ
```

```
qePEASzpFuVn516Sn15qyfG4Cdp9teVy7MQW/C5dCzhzblRRBfrSrPr2UUvlN6vN
RTaaYWtPzFHWYfFh2QhQYLPIXSrCoJ72roNgTat0uXwhDxus9WPrkNR3vggD911o
khCeeww1WSBf3mH5U4G3GFVZdRK55vWbz3WQdomkpetN7H5tu0L+CmSlYEebUGKX
00bOXcI5YH6nylbEtxPqrgpE/VO6apjrNKZZFi4ntmiPjdslsYmyTTevfIWsMX/a
MIMjltoUimY95ovwwtka7vJSDtfiSGhU41XnMFtKyHOnRqsZNI7wO/W/YOJ+XZWs
++kasjqZHtk+jU4xocj9X79uEVVL1vHaeJaRkgkG6vpRH8GO9hOV9/fIanWaR40v
1A/ker0IObT5wBJEkWC9VqrIbJwcV5WfiKeyYyJ1bLcaH1rukow98RXycbFMm3VC
AviobUd9UcinToHx1NAT4SwfLqj1ajmkE5n78FUbEl/1sUjKZyhX9AvBl+jObMKs
H1MiVbxhU0JOosZ+DWVl8gMnpftUythlBiYRV3Y62vL/fUkpgrYDm74c34xDK+Ai
KhdgakN0KYeUK+xHbXtFyco/Rkd4zAug3DY8uhuGeIHPdXEyibGd+nolBo9j/7TS
/Gua2Hbu0p7jr+k2yM5dd5ebt1cIIBa46qKGdxbSd5wJO3gi+4vHsBwSF19eMWD0
DDsdn0p5guM7YX71ImhTs8i6I47iDL5Dg6ZL74yT5cojBKnxLXGHjvCqySPklZZR
mq/xoOKyzxzpQQ4ka5GfTpxlJ6mtTOrNu2fFViY06494XtXQf6ZgEQyUiQQW6c8d
-----END RSA PRIVATE KEY-----
```

Möchte man die Passphrase vom privaten Schlüssel entfernen, gelingt dies ganz einfach mit dem Kommando:

Listing 3.3: Passphrase entfernen

```
$ openssl rsa -in /path/to/originalkeywithpass.key -out /path/to/
    newkeywithnopass.key
```

3.1.2 Erstellung eines CSR

Hat man den privaten Schlüssel erstellt, geht es mit der Erstellung des *Certificate Signing Request* (CSR) weiter. Es werden an dieser Stelle nur die Komponenten des CSR beschrieben, welche für die Erlangung eines Zertifikats für Webserver am gebräuchlichsten sind. Die vollständige Spezifikation der Syntax kann in [16] nachgelesen werden.

Die Erstellung des CSR ist für gewöhnlich ein interaktiver Prozess, in dem die benötigten Angaben für das Zertifikat abgefragt werden. Als Beispiel soll an dieser Stelle die folgende Ausgabe dienen:

Listing 3.4: Interaktive CSR-Erstellung

```
/tmp$ openssl req -new -key test.key -out test.csr
Enter pass phrase for test.key:
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) []:Musterstadt
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Musterfirma
Organizational Unit Name (eg, section) []:Musterabteilung
Common Name (e.g. server FQDN or YOUR name) []:foo.example.com
Email Address []:foo@example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:.
An optional company name []:.
```

Das optionale Feld *challenge password* wird in [17, Abschnitt 5.4.1] spezifiziert. Danach soll es genutzt werden, um den ursprünglichen Antragsteller eines Zertifikats bei einem Zertifikatswiderruf identifizieren zu können. Ristić schreibt dazu in [21, S. 13], dass dieses Feld in der Praxis von so gut wie keiner CA ausgewertet wird und rät dazu, es unausgefüllt zu lassen.

Neben dem interaktiven Modus gibt es auch die Möglichkeit, die benötigten Informationen als Parameter beim Aufruf von `openssl` zu übergeben. Der CSR aus obigen Beispiel kann damit auch wie folgt erstellt werden. Es wird nun lediglich nach der Eingabe der Passphrase gefragt:

Listing 3.5: CSR-Erstellung mit Parametern

```
1         openssl req -batch -sha256 -new -key test.key -out test.csr -subj "/C=DE
              /L=Musterstadt/O=Musterfirma/OU=Musterabteilung/CN=foo.example.com/
              emailAddress=foo@example.com"
```

Der Parameter `-batch` schaltet den interaktiven Modus aus. Mit `-sha256` wird angegeben, dass dieser CSR mit SHA-256 signiert werden soll. Dieser Wert kann jedoch von der CA überschrieben werden.

Möchte man den Inhalt des CSR noch einmal überprüfen, bevor man ihn an die CA sendet, kann man dies mit dem folgenden Befehl tun:

Listing 3.6: CSR-Ausgabe im Klartext

```
1  tmp$ openssl req -text -verify -in test.csr -noout
2  verify OK
3  Certificate Request:
4      Data:
5          Version: 0 (0x0)
6          Subject: C=DE, L=Musterstadt, O=Musterfirma, OU=Musterabteilung, CN=foo.
               example.com/emailAddress=foo@example.com
7          Subject Public Key Info:
8              Public Key Algorithm: rsaEncryption
9                  Public-Key: (2048 bit)
10                 Modulus:
11                     00:bb:97:62:57:ca:47:2f:d2:88:1b:80:c2:42:fc:
12                     d3:9a:ca:1d:6a:16:49:f7:6d:48:7f:7f:b5:a4:73:
13                     62:6e:37:01:06:66:31:d7:0a:5e:22:d1:99:2b:44:
14                     46:80:10:eb:a6:0f:32:91:de:b8:1f:43:29:55:13:
15                     83:f7:d8:1a:2a:6d:2e:73:3f:17:d9:41:b3:32:83:
16                     f8:81:52:81:57:8f:7d:2f:bd:05:5a:e7:10:d9:2a:
17                     de:fe:88:a0:4e:2b:1e:6f:c1:ee:2f:f1:7a:be:fb:
18                     48:77:1e:a0:67:02:bd:7d:a8:bf:ff:76:9b:c7:26:
19                     d3:25:1e:57:cd:42:8d:09:92:54:3d:2f:da:85:ca:
20                     e1:21:2b:54:c0:20:a0:42:96:42:89:a5:e7:e8:9b:
21                     b8:23:71:41:01:8e:2e:1e:4d:53:99:80:50:32:4a:
22                     8d:86:f3:69:a9:c2:2c:0c:2d:10:fb:50:0e:bc:26:
23                     8e:fa:23:52:13:d0:fa:6f:1a:a1:9f:a8:b5:dd:46:
24                     a7:b4:06:b5:e5:aa:a2:ec:43:69:8c:92:98:b3:0a:
25                     d6:a8:34:81:24:8d:70:bf:ea:1c:63:4e:89:64:ce:
26                     49:f0:96:30:41:72:3c:ce:2a:da:b5:8a:36:ff:10:
27                     2e:3b:4b:5a:d4:4b:cf:52:62:d6:ff:1b:26:f6:27:
28                     ef:e3
29                 Exponent: 65537 (0x10001)
30         Attributes:
31             a0:00
32     Signature Algorithm: sha256WithRSAEncryption
33          b8:ca:d8:00:85:88:fc:73:ce:6a:41:39:57:4f:a0:f0:df:d2:
34          28:2c:27:d6:33:93:4f:74:09:b9:0b:02:5d:7a:2b:46:f3:e6:
35          6b:f1:8c:d7:87:1f:31:e8:b4:ee:d5:d2:fb:a5:a4:04:07:8b:
36          3f:32:4c:78:d9:62:47:27:76:ba:2a:2b:47:04:45:29:e7:4c:
37          f1:b2:a9:9a:e8:fe:a3:42:26:5d:3e:17:44:47:0a:32:90:84:
38          de:17:08:74:14:63:a1:24:56:57:0a:98:15:30:ee:cc:2e:e5:
39          13:62:cf:a2:10:77:b1:a4:6d:4f:b0:06:29:0b:bb:7a:aa:b0:
40          25:ee:14:4c:b0:e0:84:5c:17:ea:99:97:4e:69:15:29:1e:33:
41          41:9d:4e:ca:ff:3f:c2:b4:b8:a6:d2:7a:d6:5b:16:97:9d:eb:
42          d2:e8:e1:ba:c4:30:7f:32:41:11:a2:28:f9:bd:8d:ec:f5:b8:
43          0d:e3:72:23:49:d4:78:79:04:68:ac:f9:02:68:9e:73:15:b2:
44          20:34:4c:d2:a4:4c:b5:99:a4:6d:8f:1d:37:74:cc:43:5b:de:
45          04:11:d4:e1:07:7e:fb:05:c7:84:82:e0:c8:50:58:8c:eb:c7:
46          3f:37:8e:d0:41:2a:a1:ae:9c:e8:49:3d:4a:f5:b6:fb:61:e7:
47          87:f7:3c:49
```

Der generierte CSR kann nun an eine CA gesendet werden, um von dieser signiert zu werden. Nur mit der Signatur einer vertrauenswürdigen CA (vgl. Kapitel 2.4) vertraut ein Webbrowser diesem Zertifikat.

3.2 Skriptgesteuerte Generierung von Private Key und CSR

Die in Abschnitt 3.1 beschriebene Generierung des privaten Schlüssels und des CSR kann mit dem Skript *make-cert.sh*[1] automatisiert werden, was die Erstellung und Verwaltung vereinfacht. Das folgende Skript kann auf jedem System ausgeführt werden, auf dem eine Bash[2] zur Ausführung zur Verfügung steht.

Listing 3.7: make-cert.sh

```
#!/bin/bash
# Shell-Skript zur Erstellung von TLS/SSL-Zertifikaten  mit der Bash
# Originalversion von JP, Bash-Anpassung f r Cygwin von MS 2016-05-09,
# Erweiterung von JKA 2016-05-22

# Variablen ###########################################################
OPENSSL=/usr/bin/openssl

# Funktionen ##########################################################
usage() {
        cat << EOF
        usage: $0 OPTIONS
        Shell-Skript zur Erstellung von TLS/SSL-Zertifikaten  mit der Bash
        Originalversion von JP, Bash-Anpassung f r Cygwin von MS 2016-05-09,
        Erweiterung von JKA 2016-05-22

        OPTIONS:
        -f Gibt den FQDN an. Beispiel: "foo.example.com"
        -h Zeigt diese Nachricht an.
        -p Gibt den Dateipraefix an. Beispiel: "foo"
EOF
}

# Hauptteil ###########################################################
while getopts .f:hp:. OPTION
do
        case $OPTION in
                p)
                        ZERTPREFIX="${OPTARG}"
                        ;;
                h)
                        usage
                        exit;;
                f)
                        ZERTCN="${OPTARG}"
                        ;;
                ?)
                        usage
                        exit;;
        esac
done

if [[ -z $ZERTPREFIX ]]; then
        read -p "Filepraefix:␣" ZERTPREFIX
fi

if [[ -z $ZERTCN ]]; then
        read -p "FQDN:␣" ZERTCN
fi
```

[1] Download unter: http://www.my-it-brain.de/wordpress/wp-content/uploads/2016/08/make-cert.txt [Letzter Abruf: 09.08.2016]

[2] https://de.wikipedia.org/wiki/Bash_(Shell) [Letzter Abruf: 26.07.2016

```
50
51 ZERTDN_PREF="/C=DE/ST=Bundesland/L=Stadt/O=Firma/OU=Abteilung"
52 S_DN="${ZERTDN_PREF}/CN=${ZERTCN}"
53 # Fuer den Fall, dass man mal was anderes braucht S_DN explizit setzen: S_DN="/C
       =DE/O=Firma/OU=Abteilung/CN=GRP:abuse"
54 YEAR=`date +"%Y"`
55 DIR=${ZERTPREFIX}.${YEAR}
56 PRIV=${ZERTPREFIX}_priv.pem
57 REQ=${ZERTPREFIX}_request.csr
58
59 mkdir -p ${DIR} || exit 73
60 chmod 700 ${DIR}
61 cd ${DIR} || exit 73
62 # falls nicht vorhanden, den privaten Schl ssel anlegen
63 # Da mkpasswd unter Cygwin/Windows anders arbeitet, nutzen wir hier openssl mit
       rand-Operator
64 umask 077; openssl rand -base64 12 > ${ZERTPREFIX}_priv.passwd
65 umask 077; openssl rand -base64 12 > ${ZERTPREFIX}_revocation.passwd
66
67 if [ ! -f $PRIV ]; then
68   touch ${PRIV}
69   chmod 600 ${PRIV}
70         ${OPENSSL} genrsa  -passout file:${ZERTPREFIX}_priv.passwd -aes256 -out
              ${PRIV} 2048
71 fi
72 # Request generieren
73 ${OPENSSL} req -batch -sha256 -new -key ${PRIV} -passin file:${ZERTPREFIX}_priv.
       passwd -out ${REQ} -subj "${S_DN}"
74 chmod 600 ${REQ}
75
76 # Request in Textform exportieren (nur zur manuellen Kontrolle)
77 ${OPENSSL} req -text -verify -in ${REQ} > ${REQ}.txt
78
79 cat <<EOF > ${ZERTPREFIX}.TODO
80 # Request /.csr an CA   bermitteln
81 #
82 # Signiertes File als ${ZERTPREFIX}.pem abspeichern bzw. verlinken ln -s <cert-
       xyz.pem> ${ZERTPREFIX}.pem
83 # Sicherungs-/Transportformat ist pkcs12
84 ${OPENSSL} pkcs12 -export -passin file:${ZERTPREFIX}_priv.passwd -inkey $PRIV -
       in ${ZERTPREFIX}.pem -out ${ZERTPREFIX}.p12
85 #
86 # Laufzeit ermitteln
87 ${OPENSSL} x509 -in ${ZERTPREFIX}.pem -enddate -noout
88 #
89 # Beispiele f r weitere Verarbeitung
90 # ${OPENSSL} rsa -passin file:${ZERTPREFIX}_priv.passwd -in $PRIV -out $PRIV.
       unverschluesselt
91
92 EOF
```

Bevor das Skript in Listing 3.7 auf der vorherigen Seite ausgeführt werden kann, sind die Optionen in Zeile 51 an die eigenen Bedürfnisse anzupassen.

Um einen privaten Schlüssel und den davon abgeleiteten CSR zu erstellen, übergibt man einen Prefix und den FQDN, welcher als CN in das Zertifikat aufgenommen werden soll, als Parameter an das Skript. Folgendes Beispiel soll die Anwendung verdeutlichen.

Es sollen ein privater Schlüssel und ein CSR für den FQDN *test.my-it-brain.de* generiert werden. Als Prefix wird der Hostname *test* verwendet. Das Skript wird mit den folgenden Parametern aufgerufen:

```
1         make-cert.sh -p test -f test.my-it-brain.de
```

Das Skript legt ein Verzeichnis nach dem Muster `<Prefix>.<Jahreszahl>` an. In diesem Beispiel ist es das Verzeichnis test.2016. Dieses Verzeichnis enthält Dateien, welche dem Listing 3.8 auf der nächsten Seite zu entnehmen sind.

Listing 3.8: Durch make-cert.sh erstellte Dateien

```
1 ll test.2016/
2 total 32
3 -rw------- 1 jkastning  jkastning    17 Jun 18 21:42 test_priv.passwd
4 -rw------- 1 jkastning  jkastning  1766 Jun 18 21:42 test_priv.pem
5 -rw------- 1 jkastning  jkastning  1062 Jun 18 21:42 test_request.csr
6 -rw------- 1 jkastning  jkastning  3576 Jun 18 21:42 test_request.csr.txt
7 -rw------- 1 jkastning  jkastning    17 Jun 18 21:42 test_revocation.passwd
8 -rw------- 1 jkastning  jkastning   540 Jun 18 21:42 test.TODO
```

test_priv.passwd enthält die zufällig generierte Passphrase, mit welcher der private Schlüssel verschlüsselt wurde. Diese Passphrase wird benötigt, um den privaten Schlüssel nutzen zu können.

test_priv.pem enthält den privaten Schlüssel in verschlüsselter Form. Um ihn zu entschlüsseln wird die Passphrase aus der zuvor genannten Datei benötigt.

test_request.csr enthält den CSR.

test_request.csr.txt enthält den CSR zur Kontrolle im Klartext.

test_revocation.passwd enthält das Passwort, welches benötigt wird, um ein Zertifikat beim DFN zu widerrufen und sperren zu lassen.

test.TODO ist eine Datei, welche weitere Hinweise zur Verwaltung des Zertifikats beinhaltet. Die beschriebenen Schritte sind auszuführen, wenn man das signierte Zertifikat erhalten hat.

Die Datei `test_request.csr` kann nun an eine CA (vgl. 2.4 auf Seite 7) übermittelt werden.

Bei Verwendung dieser Methode werden alle Dateien, welche zur Implementierung des TLS/SSL-Zertifikats benötigt werden, in einem gemeinsamen Verzeichnis abgelegt. Die Verwendung eines Prefix, aus dem sich das Zertifikat ableiten lässt und die Angabe des Jahres, in dem der CSR erstellt wurde, ermöglicht einheitliche und übersichtliche Verwaltung mehrerer Zertifikate auf einem Host.

3.3 Certificate Signing Request (CSR) für mehrere Hostnamen erstellen

Die in den vorangegangenen Abschnitten erstellten Certificate Signing Requests enthalten einen einzigen CN und sind daher nur für einen einzigen Hostnamen gültig. Dies ist nicht immer vorteilhaft. Betreibt man z. B. mehrere Webpräsenzen auf einem Host, benötigt man für jede Webseite ein separates Zertifikat. In diesem Fall wäre es viel einfacher, ein einzelnes Zertifikat verwenden zu können, welches alle benötigten Domain-Namen beinhaltet. Des Weiteren gibt es meist mehr als einen Weg, über den ein Dienst erreichbar ist. So sind z. B. viele Webseiten sowohl über die URL *https://example.com* als auch über *https://www.example.com* erreichbar. Damit existieren bereits zwei Namen für eine Webseite, die in einem Zertifikat untergebracht werden sollten. Um nun mehrere Hostnamen in einem Zertifikat unterzubringen, existieren zwei Möglichkeiten.

So können zum einen Wildcard-Zertifikate (z. B. **.example.com*) verwendet werden. Ein Wildcard-Zertifikat umfasst dabei alle Subdomains und kann daher für

alle FQDNs unterhalb von *example.com* verwendet werden. Um einen CSR für ein Wildcard-Zertifikat zu beantragen, können die in Abschnitt 3.1 auf Seite 16 und den folgenden Abschnitten verwendeten Methoden verwendet werden. Hierbei wird als CN statt eines Hostnamens lediglich ein '*' eingetragen.

Die zweite Möglichkeit ist die Nutzung der X.509 Zertifikats-Erweiterung *Subject Alternative Name* [2, S. 35 ff]. Diese ermöglicht es, mehrere Identitäten an das Feld *Subject* (vgl. 2.4.1 auf Seite 8) zu binden. Neben der Angabe mehrerer DNS-Namen erlaubt die Erweiterung auch die Angabe von IP-Adressen, Uniform Resource Identifiers (URI) und E-Mail-Adressen.

In der Praxis werden beide Möglichkeiten häufig miteinander kombiniert [21, S. 15]. In diesem Fall enthält das Zertifikat sowohl die Basisdomain als auch eine Wildcard für sämtliche Subdomains (z. B. *example.com* und **.example.com*). Um die Erweiterung *Subject Alternative Name* nutzen zu können, wird eine Konfigurationsdatei benötigt [21, S. 16].

Listing 3.9 zeigt ein Beispiel einer Konfigurationsdatei[3], welche als Vorlage für die eigene Nutzung dienen kann.

Listing 3.9: Beispiel einer Konfigurationsdatei

```
1  [ req ]
2  default_bits = 2048
3  default_keyfile = test.my-it-brain.de_privatekey.pem
4  distinguished_name = req_distinguished_name
5  encrypt_key = no
6  prompt = no
7  string_mask = nombstr
8  req_extensions = v3_req
9
10 [ v3_req ]
11 basicConstraints = CA:FALSE
12 keyUsage = digitalSignature, keyEncipherment, dataEncipherment
13 extendedKeyUsage = serverAuth, clientAuth
14 subjectAltName = DNS:test.my-it-brain.de, DNS:*.test.my-it-brain.de
15
16 [ req_distinguished_name ]
17 countryName = DE
18 stateOrProvinceName = Nordrhein-Westfalen
19 localityName = MeinOrt
20 0.organizationName = MeineFirma
21 organizationalUnitName = MeineAbteilung
22 commonName = test.my-it-brain.de
```

Im ersten Block der Konfigurationsdatei werden allgemeine Angaben für den CSR vorgenommen. Hier wird angegeben, dass eine Schlüssellänge von 2048 bit verwendet wird und der private Schlüssel nicht durch eine Passphrase geschützt wird. Darüber hinaus wird angegeben, dass für den *distinguished_name* die Informationen aus dem Block `[ req_distinguished_name ]` berücksichtigt werden sollen und die Erweiterungen aus dem Block `[ v3_req ]` verwendet werden.

Der Block `[ v3_req ]` enthält u. a. die Erweiterung "subjectAltName„. Hier können durch Kommata getrennt die FQDNs angegeben werden, die in das Zertifikat mit aufgenommen werden sollen. Weitere Hinweise und Beispiele zur Angabe von DNS-Namen, IP-, E-Mail-Adressen u. a. in diesem Feld finden sich in der

[3]Download unter: `http://www.my-it-brain.de/wordpress/wp-content/uploads/2016/08/Beispiel_einer_OpenSSL-Konfigurationsdatei.txt` [Letzter Abruf: 09.08.2016]

OpenSSL-Dokumentation.[4] Die Werte der übrigen Felder können aus dem Beispiel übernommen werden. Für eine detaillierte Beschreibung dieser Erweiterungen siehe [2, Abs. 4.2.1].

Man beachte, dass der angegebene *commonName* ebenfalls unter dem Feld *subjectAltName* mit angegeben wurde. Dies ist erforderlich, da bei Verwendung von *subjectAltName* das Feld *commonName* ignoriert wird [21, S. 16]. Es wird jedoch empfohlen, das Feld *commonName* dennoch zu belegen, da etliche CAs einen CSR ablehnen, wenn das Feld nicht belegt ist. Dies gilt aktuell (Stand 31.05.2016) z. B. auch für die PKI des DFN.

Die Verwendung der Konfigurationsdatei aus Listing 3.9 auf der vorherigen Seite wird in Abschnitt 3.3.1 und 3.3.2 erläutert.

3.3.1 Erstellung eines CSR für mehrere Hostnamen unter Verwendung von OpenSSL

Um mit Hilfe von OpenSSL einen CSR für mehrere Hostnamen zu erstellen, wird das Listing 3.9 auf der vorherigen Seite als Datei `csr.cfg` gespeichert und an die eigenen Bedürfnisse angepasst. Der CSR kann nun mit folgendem Kommando erstellt werden:

Listing 3.10: OpenSSL-Kommando zur Erstellung des CSR

```
1   openssl req -new -out test.csr -verify -config csr.cfg
```

Bei dieser Variante entfällt die separate Generierung des privaten Schlüssels, da dessen Erstellung in der Konfigurationsdatei bereits mit angegeben wurde. In dem Verzeichnis, in dem der Befehl ausgeführt wurde, befinden sich nun die folgenden Dateien:

```
1 -rw-rw-r-- 1 jkastning jkastning  668 Jun 25 11:07 csr.cfg
2 -rw-rw-r-- 1 jkastning jkastning 1269 Jun 25 11:10 test.csr
3 -rw-rw-r-- 1 jkastning jkastning 1704 Jun 25 11:10 test.my-it-brain.
      de_privatekey.pem
```

Die PEM-Datei enthält den privaten Schlüssel. Die CSR-Datei enthält den damit erzeugten Certificate Signing Request, welcher nun an eine CA übermittelt werden kann. Selbstverständlich kann man den CSR zuvor noch mit dem Kommando aus Listing 3.6 auf Seite 19 überprüfen.

3.3.2 Erstellung eines CSR für mehrere Hostnamen unter Verwendung von make-cert.sh

Um mit Hilfe des Skripts *make-cert.sh* einen CSR für mehrere Hostnamen zu erstellen, wird das Listing 3.9 auf der vorherigen Seite als Datei `csr.cfg` gespeichert und an die eigenen Bedürfnisse angepasst. Vorausgesetzt das Skript und die Konfigurationsdatei liegen im selben Verzeichnis, kann der CSR mit dem Kommando aus Listing 3.11 auf der nächsten Seite erstellt werden. Liegen das Skript und die Konfigurationsdatei in unterschiedlichen Verzeichnissen, muss der vollständige Pfad zur Konfigurationsdatei angegeben werden.

[4] `https://www.openssl.org/docs/manmaster/apps/x509v3_config.html#Subject-Alternative-Name`

Listing 3.11: make-cert-Aufruf zur Erstellung des CSR

```
1 make-cert.sh -p test -f test.my-it-brain.de -c csr.cfg
```

Achtung: Mit dem Parameter -f muss der FQDN übergeben werden, welcher später im CN des Zertifikats enthalten sein soll.

Durch die Ausführung des Skripts wird ein Verzeichnis erstellt, welches die vom Skript erzeugten Dateien enthält. Die Struktur entspricht der aus Listing 3.8 auf Seite 22. Die CSR-Datei enthält den Certificate Signing Request, welcher nun an eine CA übermittelt werden kann.

3.4 Zertifizierungsstellen

In den vorangegangenen Abschnitten wurden verschiedene Verfahren beschrieben, um Certificate Signing Requests für ein oder mehrere Hostnamen zu erstellen. Dieser Abschnitt widmet sich der Beschreibung einiger zufällig ausgewählter CAs, bei denen ein CSR zur Validierung eingereicht werden kann.

Da sich die Verfahren der einzelnen Zertifizierungsstellen zur Registrierung, Übermittlung des CSR und Validierung von Zeit zu Zeit ändern, werden diese Verfahren hier nicht im Detail beschrieben. Informationen dazu sind den Webseiten und der Dokumentation der einzelnen Zertifizierungsstellen zu entnehmen.

Warnung: Einige CAs bieten an, den privaten Schlüssel für das SSL-Zertifikat online im Webbrowser zu generieren. Dies ist in jedem Fall zu vermeiden. Durch die Generierung des privaten Schlüssels im Browser hat man keine Kontrolle darüber, wer alles Zugriff auf diesen Schlüssel hat. Eine Kopie des persönlichen Schlüssels könnte auf dem Server des Anbieters gespeichert bleiben oder durch einen Angriff auf die Webseite des Anbieters abgegriffen werden. Der private Schlüssel sollte daher immer auf einem vertrauenswürdigen Rechner erstellt und niemals an eine CA oder einen sonstigen Dritten übermittelt werden.

Nachdem der CSR von einer CA validiert wurde, stellt diese daraufhin ein signiertes Zertifikat aus, welches zusammen mit dem dazugehörigen privaten Schlüssel im Zielsystem implementiert werden kann.

3.4.1 Let's Encrypt

Let's Encrypt[5] ist eine gemeinnützige Zertifizierungsstelle. Getragen von der „Internet Security Research Group (ISRG)“[6] stellt sie kostenlose DV-Zertifikate für jedermann bereit. Let's Encrypt vertritt dabei laut eigener Aussage im Wesentlichen folgende Prinzipien:

- **Frei:** Jeder Domainbesitzer kann Let's Encrypt nutzen, um ein kostenloses SSL-Zertifikat zu generieren, dem die gängigen Webbrowser vertrauen.

[5] `https://letsencrypt.org` [Letzter Abruf: 26.07.2016]
[6] `https://letsencrypt.org/isrg/` [Letzter Abruf: 26.07.2016]

- **Automatisierung:** Eine auf dem Webserver laufende Software kümmert sich um die Generierung, Beantragung, Signierung, Ausstellung und Erneuerung der SSL-Zertifikate.
- **Sicherheit:** Einsatz aktueller TLS-Sicherheit und Best Practices, um Administratoren dabei zu helfen, ihre Server zu sichern.
- **Transparenz:** Alle Zertifikate, sowohl ausgestellte als auch widerrufene, werden protokolliert und können von jedermann nachgeprüft werden.
- **Offen:** Das Protokoll zur automatisierten Erstellung und Erneuerung von Zertifikaten wird als offener Standard veröffentlicht.
- **Kooperativ:** Let's Encrypt ist eine Kooperation aus verschiedenen Organisationen und möchte einen Mehrwert für die Gemeinschaft bieten, frei von der Kontrolle eines einzigen Unternehmens.

Let's Encrypt möchte mit seinem Dienst Administratoren Arbeit abnehmen und die Verwaltung von SSL-Zertifikaten weitgehend automatisieren. Dazu wird auf dem zu sichernden Server eine Client-Software installiert, welche

- die Erstellung der Schlüssel,
- die Erstellung des Certificate Signing Request (CSR),
- Übermittlung des CSR an die Zertifizierungsstelle (CA) und
- die Domain-Validierung

übernimmt.

Ist die Domain-Validierung erfolgreich, wird das Zertifikat von der Zertifizierungsstelle signiert und an den Client übermittelt. Je nach Konfiguration des Clients kann dieser das Zertifikat auf dem lokalen Server speichern oder direkt im Webserver installieren. Anschließend kann der Client dazu genutzt werden, mit ihm verwaltete Zertifikate automatisch zu erneuern.

3.4.2 StartSSL

StartSSL wurde im Februar 2005 als Zertifizierungsstelle vom israelischen Unternehmen StartCom gegründet. Die CA bietet verschiedene Klassen von Zertifikaten an (vgl. 2.4.1 auf Seite 8). Darunter mit StartSSL Free auch ein kostenloses Klasse 1 DV Zertifikat. Wer sich registriert, kann bis zu 5 Domains und eine E-Mail-Adresse mit diesem kostenlosen Zertifikat validieren lassen. Laut Aussage der CA-Webseite, ist die Anzahl der kostenlosen Zertifikate unbegrenzt.[7] Jedoch wird für den Widerruf eines Zertifikats eine Gebühr erhoben.

StartPKI[8] ist ein weiteres Produkt von StartCom, welches sich vor allem an Unternehmen und Organisationen richtet, um diesen den Betrieb einer CA mit ihrem Markennamen zu ermöglichen. Der Markenname wird dabei im Feld *Issuer* aufgeführt. In diesem Modell zahlt der Kunde nicht für die Zertifikate, sondern eine Gebühr zur Validierung der Identität.

[7] `https://www.startssl.com/` [Letzter Abruf: 26.07.2016]
[8] `https://www.startssl.com/StartPKI` [Letzter Abruf: 26.07.2016]

Mit StartAPI[9] und StartEncrypt[10] stehen noch zwei Produkte zur Verfügung, welche grundsätzlich auf der Architektur von Let's Encrypt (siehe Abschnitt 3.4.1 auf Seite 25) basieren. Sie sollen die Verwaltung und Installation von Zertifikaten automatisieren helfen.

Die StartCom-Zertifikate werden von allen gängigen modernen Webbrowsern und Betriebssystemen akzeptiert. Informationen zu *Certification Policies, Practice Statements & Resources* finden sich unter der URL `https://www.startssl.com/policy`.

In [12] wird im Abschnitt „SSL/TLS Zertifikat“ beschrieben, wie der Registrierungsprozess abläuft, ein CSR eingereicht und ein Zertifikat ausgestellt wird.

3.4.3 Thawte

Thawte ist eine Zertifizierungsstelle, welche 1995 von Mark Shuttleworth in Südafrika gegründet wurde. Das Unternehmen wurde 1999 von Verisign gekauft und ging 2010 in den Besitz von Symantec über.[11]

Thawte bietet neben verschiedenen TLS/SSL-Zertifikaten[12] auch verschiedene Codesigning-Zertifikate[13] an. Hierbei handelt sich ausschließlich um kostenpflichtige Angebote.

Zertifikate dieser CA werden ebenfalls von allen gängigen modernen Browsern und Betriebssystemen akzeptiert. Die Nutzungsbedingungen und Richtlinien der CA finden sich unter der URL: `https://www.thawte.de/repository/` [Letzter Abruf: 26.07.2016]

3.4.4 CAcert

CAcert.org[14] ist eine von einer Gemeinschaft betriebene Zertifizierungsstelle, die kostenfreie Zertifikate für die Mitglieder der Gemeinschaft ausstellt.

Diese CA basiert nicht wie die meisten CAs auf einer hierarchischen PKI, sondern auf einem dezentralen Vertrauensnetzwerk (Web of Trust[15]). Hierbei übernehmen erfahrene Mitglieder der Gemeinschaft die Funktion der RA und validieren die Identität von anderen Mitgliedern, wofür letztere sogenannte Vertrauenspunkte erhalten. Für eine allgemeine Beschreibung dieser CA wird auf den entsprechenden Wikipedia-Artikel[16] verwiesen. Eine Besonderheit dieser CA ist, dass neben der anonymen Bestätigung einer Domain oder E-Mail-Adresse auch der Name des Mitglieds in ein Zertifikat mit aufgenommen wird, sobald dieses die dafür notwendige Anzahl Vertrauenspunkte erreicht hat.

[9] `https://www.startssl.com/StartAPI` [Letzter Abruf: 26.07.2016]
[10] `https://www.startssl.com/StartEncrypt` [Letzter Abruf: 26.07.2016]
[11] `https://en.wikipedia.org/wiki/Thawte` [Letzter Abruf: 26.07.2016]
[12] `https://www.thawte.de/ssl/` [Letzter Abruf: 26.07.2016]
[13] `https://www.thawte.de/code-signing/` [Letzter Abruf: 26.07.2016]
[14] `https://www.cacert.org/index.php?id=0&lang=de` [Letzter Abruf: 26.07.2016]
[15] `https://de.wikipedia.org/wiki/Web_of_Trust` [Letzter Abruf: 26.07.2016]
[16] `https://de.wikipedia.org/wiki/CAcert` [Letzter Abruf: 26.07.2016]

Die CA stellt den Mitgliedern ihrer Gemeinschaft kostenlose Zertifikate in unbegrenzter Anzahl zur Verfügung. Die Wurzelzertifikate der CA fehlen jedoch in den meisten gängigen Browsern, E-Mail-Clients und Betriebssystemen. Dies bedeutet, dass CAcert.org von diesen nicht als vertrauenswürdige Zertifizierungsstelle geführt wird. Anwender, welche einen mit einem CAcert-Zertifikat gesicherten Dienst nutzen, erhalten von ihrem Client-Programm eine Meldung, dass die Herkunft des Zertifikats nicht überprüft werden kann. Um diese Meldungen zu vermeiden, muss das Wurzelzertifikat manuell auf den Client-Systemen installiert werden.

Um Mitglied der Gemeinschaft werden zu können, ist eine Registrierung und Zustimmung zum *CAcert Community Agreement* (CCA) erforderlich. Informationen zum CCA und den Richtlinien der Gemeinschaft finden sich unter:

- CCA: `https://www.cacert.org/policy/CAcertCommunityAgreement.html`
- Richtlinien: `https://www.cacert.org/policy/`

3.4.5 SSL-Zertifikate von Hosting-Providern

Anstatt sich direkt an eine der verschiedenen Zertifizierungsstellen zu wenden, kann man auch bei etlichen Hosting-Providern ein TLS/SSL-Zertifikat als Option zu den verfügbaren Webhosting-Produkten erhalten. Jedoch kann man diese Zertifikate meist nicht separat erwerben. Im Folgenden werden einige Angebote bekannter deutscher Hosting-Provider aufgezählt. Unter den genannten URLs befinden sich weitere Informationen zu den einzelnen Angeboten:

- Strato: `https://www.strato.de/ssl-zertifikat/`
- Hosteurope: `https://www.hosteurope.de/SSL-Zertifikate/`
- 1&1: `https://hosting.1und1.de/ssl-zertifikate?__lf=Order-Product`

3.5 Implementierung der Zertifikate im Zielsystem

In diesem Abschnitt wird beschrieben, wie ein Zertifikat, welches von einer CA validiert und signiert wurde, zusammen mit dem dazugehörigen privaten Schlüssel im Zielsystem implementiert wird. In Abhängigkeit davon, ob die ausstellende CA ihr Wurzel- oder ein Zwischen-Zertifikat zur Signatur des CSR verwendet hat, sind noch weitere Zertifikate nötig, um die komplette Zertifikatskette (siehe Abschnitt 2.4.2 auf Seite 9) bereitstellen zu können. Im Folgenden werden die URLs aufgeführt, unter denen die Wurzel- und Zwischen-Zertifikate, der unter Abschnitt 3.4 auf Seite 25 aufgeführten Zertifizierungsstellen abgerufen werden können:

Let's Encrypt `https://letsencrypt.org/2015/06/04/isrg-ca-certs.html`

HRZ Uni Bielefeld CA `https://info.pca.dfn.de/uni-bielefeld-ca/index.html`

StartSSL CA `https://www.startssl.com/root`

Thawte Root `https://www.thawte.com/roots/`

Thawte Intermediate `http://tinyurl.com/z6b2cek`

CAcert `https://www.cacert.org/index.php?id=3`

Mit Hilfe dieser Wurzel- und Zwischen-Zertifikate kann die für eine Implementierung evtl. benötigte Zertifikatskette erstellt werden.

Hinweis: Es sei an dieser Stelle darauf hingewiesen, dass die Dateiendung einer Zertifikatsdatei in der Regel frei gewählt werden kann. Eine Datei, welche ein TLS-Zertifikat enthält kann z. B. auf *.cer, *.crt, *.pem oder *.Zertifikat enden.

An dieser Stelle wird angenommen, dass alle benötigten Dateien für eine Implementierung vorliegen. Für die folgenden Beispiele nehmen wir an, dass die Dateinamen wie folgt lauten:

test_priv.key enthält den unverschlüsselten privaten Schlüssel.

test.csr enthält den CSR.

test.pem enthält das von einer CA signierte Zertifikat.

test-chain.pem enthält die Zertifikatskette bis zum Wurzelzertifikat.

Vom CSR abgesehen sind die genannten Dateien auf dem Zielsystem zu platzieren. Sie müssen in einem Verzeichnis gespeichert werden, welches der Dienst, für den TLS implementiert werden soll, lesen darf. Die Dateien sollten jedoch keinesfalls im Wurzelverzeichnis (DocumentRoot) des jeweiligen Dienstes gespeichert werden. Hier haben in aller Regel auch die Benutzer eines Dienstes Zugriff. In diesem Fall könnten Zertifikat und privater Schlüssel leicht kompromittiert bzw. entwendet werden.

Die folgenden Abschnitte beschreiben im Ansatz, wie TLS für verschiedene Dienste implementiert wird. Für weiterführende Informationen wird auf die Dokumentation des jeweiligen Dienstes verwiesen. Für alle Dienste gilt, dass diese neu geladen werden müssen, wenn die Dateien für das Zertifikat und den privaten Schlüssel ausgetauscht wurden.

3.5.1 Apache

Der *Apache*-Webserver ist ein beliebter und weit verbreiteter Webserver. Er wird vor allem unter Linux und Unix eingesetzt, steht jedoch auch für Microsoft Windows zur Verfügung. Dieser Abschnitt setzt voraus, dass der Apache-Webserver bereits auf dem Zielsystem installiert und so konfiguriert ist, dass Inhalte unverschlüsselt über das HTTP-Protokoll ausgeliefert werden können. Der Verweis auf die Dokumentation zum Apache 2.4 findet sich im Literaturverzeichnis unter [5].

Um TLS/SSL implementieren zu können, muss der Apache auf Port 443 lauschen und das Apache-Modul *mod_ssl* geladen sein. Die Konfiguration muss mindestens folgende Angaben aus Listing 3.12 auf der nächsten Seite enthalten [vgl. 5, Abschnitt: SSL/TLS-Verschlüsselung].

Listing 3.12: Beispielkonfiguration eines VirtualHost

```
1 LoadModule ssl_module modules/mod_ssl.so
2
3 Listen 443
4 <VirtualHost *:443>
5     ServerName test.my-it-brain.de
6     SSLEngine on
7     SSLCertificateFile "/Pfad/zu/test.pem"
8     SSLCertificateKeyFile "/Pfad/zu/test_priv.key"
9 </VirtualHost>
```

Das Beispiel aus Listing 3.12 zeigt, wie die Pfade zum Zertifikat und zur Datei mit dem privaten Schlüssel angegeben werden. Wird darüber hinaus auch noch die Angabe des Wurzelzertifikats bzw. der Zertifikatskette benötigt, geschieht dies mittels des Schlüsselworts SSLCertificateFile [siehe 5, Abschnitt: SSL/TLS-Verschlüsselung/mod_ssl reference].

3.5.2 NGINX

NGINX ist eine unter BSD-Lizenz veröffentlichte Webserver-Software, welche unter Linux, OpenBSD und auch Windows zur Verfügung steht. Die Dokumentation zu NGINX ist unter [15] abrufbar.

Um einen HTTPS-Server zu konfigurieren, muss die Konfiguration folgende Angaben umfassen [vgl. 15, Configuring HTTPS servers]:

Listing 3.13: Beispielkonfiguration eines HTTPS-Servers in NGINX

```
1 server {
2     listen              443 ssl;
3     server_name         test.my-it-brain.de;
4     ssl_certificate     /Pfad/zu/test.pem;
5     ssl_certificate_key /Pfad/zu/test_priv.key;
6     ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
7     ...
8 }
```

In Listing 3.13 wird gezeigt, wie die Pfade zum Zertifikat und privaten Schlüssel angegeben werden. Das Schlüsselwort ssl_protocols definiert, welche TLS/SSL-Protokolle verwendet werden. In diesem Beispiel sind ausschließlich die TLS-Protokolle erlaubt. Frühere SSL-Protokolle werden dagegen nicht mehr verwendet, da diese heute als nicht mehr sicher gelten.

In dem Fall, dass die Zertifikatskette (vgl. 2.4.2 auf Seite 9) vom Webserver mit ausgeliefert werden soll, sind das Serverzertifikat (test.pem) mit der Zertifikatskette (test-chain.pem) zu verknüpfen. Dabei muss das Serverzertifikat am Anfang der neuen Datei stehen, gefolgt von der Zertifikatskette, die bis zum Wurzelzertifikat führt. Unter Linux und Unix können die beiden Dateien wie folgt zusammengeführt werden:

Listing 3.14: Erstellung der Zertifikatskette

```
1 cat test.pem test-chain.pem > test-cert-chain.pem
```

In der TLS/SSL-Konfiguration wird nun mit dem Schlüsselwort ssl_certificate der Pfad zur Datei test-cert-chain.pem angegeben. Wurden die Zertifikate in der falschen Reihenfolge zusammengeführt, bricht NGINX den Start mit einer Fehlermeldung ab. In diesem Fall ist die Zusammenführung der beiden genannten Dateien zu wiederholen.

3.5.3 lighttpd

lighttpd steht ebenfalls unter BSD-Lizenz und steht für Linux, Unix und Windows zur Verfügung. Die Dokumentation des Projekts wird in einem Wiki gepflegt [siehe 4].

Im Gegensatz zu NGINX und Apache ist die Implementierung von TLS/SSL im lighttpd nicht so einfach möglich. Für die Implementierung muss lighttpd mit den Bibliotheken von OpenSSL kompiliert werden. Da eine detaillierte Beschreibung über den Rahmen dieses Werks hinausgeht, sei an dieser Stelle auf die Dokumentation [siehe 4, Abschnitt SSL Documentation] verwiesen.

Hinweis: In OpenSSL werden über die Zeit immer wieder Schwachstellen gefunden und geschlossen. Damit lighttpd davon profitieren kann, muss der Webserver jedes Mal erneut gegen die neuen OpenSSL-Bibliotheken kompiliert werden. Dieser Aufwand ist vor dem Hintergrund, dass lighttpd laut Dokumentation [siehe 4, Abschnitt SSL Documentation] in der aktuellen Version ausschließlich das SSLv3-Protokoll unterstützt, nicht verhältnismäßig. Daher rate ich an dieser Stelle von der Verwendung des lighttpd für die Implementierung von TLS/SSL ab. Statt dessen empfehle ich einen der übrigen in diesem Dokument erwähnten Webserver.

3.5.4 Microsoft IIS

Während der Erstellung dieser Arbeit stand kein Microsoft IIS für eine Implementierung zur Verfügung. Daher kann an dieser Stelle ausschließlich auf folgende Artikel verwiesen werden:

- SSL Certificate Installation in Microsoft IIS 7 [26]
- SSL Certificate Installation in Microsoft IIS 8 and IIS 8.5 [27]

3.5.5 Postfix

Postfix ist ein beliebter Mail-Transfer-Agent (MTA) für diverse Linux- und Unix-Derivate. Für die erfolgreiche Implementierung von TLS muss das Serverzertifikat zuvor mit dem Zwischenzertifikat der ausstellenden CA und dem Wurzelzertifikat zusammengeführt werden (siehe Listing 3.14 auf der vorherigen Seite), um die vollständige Zertifikatskette zu bilden (vgl. 2.4.2 auf Seite 9).

Die erstellte Zertifikatsdatei und der private Schlüssel werden mit Hilfe der beiden folgenden Schlüsselwörter in der Konfigurationsdatei `main.cf` angegeben:

```
1 smtpd_tls_cert_file = /Pfad/zu/test.pem
2 smtpd_tls_key_file = /Pfad/zu/test_priv.key
```

Weitere Hinweise zur TLS-Konfiguration siehe [25]. Eine ausführliche Beschreibung mit Beispielkonfiguration findet sich in [12].

3.5.6 Dovecot

Bei Dovecot handelt es sich um einen *Mail Delivery Agent* (MDA), welcher Zugriff auf E-Mail-Postfächer via POP3 und IMAP bereitstellt. TLS/SSL kann in Dovecot auf verschiedenen Wegen implementiert werden. Eine Darstellung an dieser Stelle wäre unvollständig. Daher wird auf die offizielle Dokumentation verwiesen. *Dovecot SSL configuration* [6] erläutert die verschiedenen Wege im Detail.

Eine erläuterte Beispielkonfiguration findet sich in [12, Grundlegende Konfiguration von Dovecot].

3.6 Implementierung von Let's Encrypt

Ein Fokus von Let's Encrypt liegt auf der einfachen Verwaltung und Verlängerung von TLS/SSL-Zertifikaten. Die CA setzt dabei auf die weitgehende Automation von Schlüsselerzeugung, CSR-Erstellung, Validierung und Zertifikatsausstellung (siehe hierzu auch 3.4.1 auf Seite 25 und [14]). Im Folgenden werden zwei ausgewählte Verfahren beschrieben, wie sich Let's Encrypt auf einem Zielsystem implementieren lässt.

Beide Verfahren beziehen sich auf eine Implementierung auf einem Ubuntu 14.04 LTS Server. Sie besitzen generelle Gültigkeit für alle äquivalenten Linux-Systeme, müssen jedoch evtl. an die eigene Distribution angepasst werden.

3.6.1 Ausschließliche Nutzung von Let's Encrypt

Mit Let's Encrypt kann der gesamte Prozess der Zertifikatsverwaltung automatisiert werden. Dazu wird der Let's Encrypt Client `certbot` auf dem Zielsystem benötigt. Auf der Webseite `https://certbot.eff.org/` kann der verwendete Webserver und das verwendete Betriebssystem ausgewählt werden. An dieser Stelle finden NGINX und Ubuntu 14.04 LTS Verwendung. Die folgenden Schritte werden mit einem unprivilegierten Benutzeraccount ausgeführt.

Im ersten Schritt wird ein Verzeichnis für den `certbot` erstellt, der Client heruntergeladen, ausführbar gemacht und installiert:

```
1  $ mkdir -p certbot
2  certbot$ cd certbot
3  certbot$ wget https://dl.eff.org/certbot-auto
4  certbot$ chmod a+x certbot-auto
5  certbot$ ./cerbot-auto
6  Creating virtual environment...
7  Installing Python packages...
8  Installation succeeded.
9  Requesting root privileges to run certbot...
10   /home/test/.local/share/letsencrypt/bin/letsencrypt
11 No installers seem to be present and working on your system; fix that or try
       running certbot with the "certonly" command
```

Die Warnung in der letzten Zeile kann ignoriert werden, da cerbot im Folgenden ausschließlich mit dem Modul certonly verwendet wird.

Das folgende Listing 3.15 zeigt den Aufruf, mit dem ein DV-Zertifikat für die Domain test.my-it-brain.de erzeugt wird. Dabei wird das Staging-Environment[17] verwendet. Möchte man Zertifikate für produktive Systeme zu erstellen, ruft man das Kommando ohne die Option `--staging` auf.

Listing 3.15: Erstellung eines DV-Zertifikats für test.my-it-brain.de

```
certbot$ ./certbot-auto certonly --staging --webroot -w /var/www/sites/test.my-
    it-brain.de/public/ -d test.my-it-brain.de
Requesting root privileges to run certbot...
  /home/benutzername/.local/share/letsencrypt/bin/letsencrypt certonly --staging
       --webroot -w /var/www/sites/test.my-it-brain.de/public/ -d test.my-it-
      brain.de

IMPORTANT NOTES:
 - Congratulations! Your certificate and chain have been saved at
   /etc/letsencrypt/live/test.my-it-brain.de/fullchain.pem. Your cert
   will expire on 2016-10-01. To obtain a new or tweaked version of
   this certificate in the future, simply run certbot-auto again. To
   non-interactively renew *all* of your certificates, run
   "certbot-auto␣renew"
```

Das Skript erstellt das Verzeichnis /etc/letsencrypt/live/test.my-it-brain.de/, in dem sich der private Schlüssel und die Zertifikatsdateien für die Domain test.my-it-brain.de befinden:

```
ertbot$ sudo ls -l /etc/letsencrypt/live/test.my-it-brain.de/
total 0
lrwxrwxrwx 1 root root 43 Jul  3 15:16 cert.pem -> ../../archive/
    test.my-it-brain.de/cert2.pem
lrwxrwxrwx 1 root root 44 Jul  3 15:16 chain.pem -> ../../archive/
    test.my-it-brain.de/chain2.pem
lrwxrwxrwx 1 root root 48 Jul  3 15:16 fullchain.pem -> ../../
    archive/test.my-it-brain.de/fullchain2.pem
lrwxrwxrwx 1 root root 46 Jul  3 15:16 privkey.pem -> ../../archive/
    test.my-it-brain.de/privkey2.pem
```

cert.pem enthält das ausgestellte Zertifikat,

chain.pem enthält die Zertifikatskette (hier das Intermediate der CA),

fullchain.pem enthält sowohl das Server- als auch das Intermediate-Zertifikat und

privkey.pem enthält den privaten Schlüssel.

Wurde das Zertifikat korrekt im Webserver implementiert, stellt es sich im Webbrowser wie in den Abbildungen 3.1 auf Seite 37 und 3.2 auf Seite 37 dar.

Möchte man mehrere FQDNs in das Zertifikat mit aufnehmen, wird der Parameter '-d' einfach mehrfach verwendet:

```
certbot$ ./certbot-auto certonly --staging --webroot -w /var/www/
    sites/test.my-it-brain.de/public/ -d test.my-it-brain.de
Requesting root privileges to run certbot...
  /home/benutzername/.local/share/letsencrypt/bin/letsencrypt
      certonly --staging --webroot -w /var/www/sites/test.my-it-brain
      .de/public/ -d test.my-it-brain.de -d test2.my-it-brain.de
```

[17] https://letsencrypt.org/docs/staging-environment/ [Letzter Abruf: 26.07.2016]

Alle angegebenen FQDNs müssen via DNS auflösbar und erreichbar sein. Die Ausstellung von Wildcard-Zertifikaten ist nicht möglich.

Die mit certbot ausgestellten Zertifikate sind 90 Tage lang gültig. Um sie zu verlängern, werden die oben genannten Befehle einfach noch einmal ausgeführt.

3.6.2 Nutzung von Let's Encrypt mit einem eigenen CSR

Wie Scott Helme [9] auf seiner Webseite schreibt, lässt sich Let's Encrypt auch mit einem eigenen CSR nutzen. Dazu wird der Client acme-tiny von Daniel Roesler [22] verwendet. Die Verwendung eines eigenen CSR kann z. B. erforderlich sein, wenn man neben TLS/SSL auch noch HPKP (siehe 2.8 auf Seite 14) implementieren möchte.

Die Installation und Verwendung von acme−tiny soll in diesem Abschnitt beispielhaft beschrieben werden. Dazu wird eine Laborumgebung auf einem Ubuntu 14.04 LTS Server verwendet. Als Webserver kommt weiterhin NGINX zum Einsatz. Für die gezeigte Konfiguration wird ein Benutzer namens *johndoe* verwendet. Dieser Benutzer verfügt über das Recht, Kommandos via sudo mit Root-Rechten ausführen zu dürfen. Der NGINX-Prozess wird unter dem Systembenutzer www-data ausgeführt. Für das folgende Beispiel sind folgende Verzeichnisse auf dem Server relevant:

/var/www/sites/test/public DocumentRoot des VirtualHost test.my-it-brain.de

/var/www/sites/test/ssl Verzeichnis, welche die zum Zertifikat gehörenden Dateien aufnimmt

/home/johndoe/acme-tiny Verzeichnis in welchem sich die Client-Software befindet (nachdem das Git-Repository geklont wurde)

Im ersten Schritt wird das benötigte GitHub-Repository auf den Server geklont:

Listing 3.16: Installation des Let's Encrypt Clients acme-tiny

```
1 :~$ git clone https://github.com/diafygi/acme-tiny.git
2 Cloning into 'acme-tiny'...
3 remote: Counting objects: 232, done.
4 remote: Total 232 (delta 0), reused 0 (delta 0), pack-reused 232
5 Receiving objects: 100% (232/232), 47.01 KiB | 0 bytes/s, done.
6 Resolving deltas: 100% (124/124), done.
7 Checking connectivity... done.
```

Im zweiten Schritt wird ein Account-Key für Let's Encrypt erstellt. *Wichtig:* Der Account-Key ist nicht mit dem privaten Schlüssel zu verwechseln, welcher für die Erstellung des CSR benötigt wird. Es handelt sich dabei um zwei unterschiedliche Schlüssel.

Listing 3.17: Generierung eines Account-Keys

```
1 $ openssl genrsa 4096 > /var/www/sites/test.my-it-brain.de/ssl/account.key
2 Generating RSA private key, 4096 bit long modulus
3 ..............................................++
4 ....................................++
5 e is 65537 (0x10001)
```

Ein privater Schlüssel und ein CSR können nach einer der in diesem Kapitel vorgestellten Methoden erzeugt werden. Beide Dateien sollten anschließend ebenfalls unterhalb des Verzeichnisses /var/www/sites/test.my-it-brain.de/ssl/ liegen. Für dieses Beispiel wurde ein CSR für den FQDN test.my-it-brain.de mit der Methode aus Abschnitt 3.2 auf Seite 20 erzeugt. Aus Sicherheitsgründen wurden die Zugriffsrechte auf den privaten Schlüssels so gesetzt, dass ausschließlich der Benutzer www-data diese Datei lesen darf.

Im nächsten Schritt wird das Verzeichnis für die Acme-Challenge [14] im Document-Root des VirtualHost erzeugt. Die Besitzrechte werden so angepasst, dass dieses Verzeichnis vom Benutzer *johndoe* geschrieben und von www-data gelesen werden kann. Dazu werden die folgenden Kommandos ausgeführt:

Listing 3.18: Verzeichnis für die Acme-Challenge

```
1 $ sudo mkdir -p /var/www/sites/test.my-it-brain.de/public/.well-known/acme-
      challenge/
2 $ sudo chmod -R 755 /var/www/sites/test.my-it-brain.de/public/.well-known
3 $ sudo chown -R johndoe:www-data /var/www/sites/test.my-it-brain.de/public/.well
      -known
```

Nun kann mit dem folgenden Befehl ein Zertifikat bei Let's Encrypt angefordert werden:

```
1 $ python acme-tiny/acme_tiny.py --account-key /var/www/sites/test.my-it-brain.de
      /ssl/account.key --csr /var/www/sites/test.my-it-brain.de/ssl/test.2016/
      test_request.csr --acme-dir /var/www/sites/test.my-it-brain.de/public/.well-
      known/acme-challenge/ > /var/www/sites/test.my-it-brain.de/ssl/test
      .2016/2016-07-07_test.crt
2 Parsing account key...
3 Parsing CSR...
4 Registering account...
5 Already registered!
6 Verifying test.my-it-brain.de...
7 test.my-it-brain.de verified!
8 Signing certificate...
9 Certificate signed!
```

Der private Schlüssel und das ausgestellte Zertifikat können nun wie unter 3.5.2 auf Seite 30 beschrieben eingebunden werden. Anschließend stellt sich das Zertifikat im Browser wie in Abbildung 3.1 auf Seite 37 und 3.2 auf Seite 37 dar.

3.7 Erneuerung von Zertifikaten

TLS/SSL-Zertifikate sind nur für einen begrenzten Zeitraum gültig. Die Gültigkeit kann je nach CA und Zertifikat schwanken.

Zertifizierungsstelle	Gültigkeit der Zertifikate
CAcert	2 Jahre
StartSSL	1 Jahr
Let's Encrypt	90 Tage

Um ein Zertifikat zu verlängern, muss der CSR erneut zur Validierung bei einer Zertifizierungsstelle eingereicht werden. Was bei den meisten CA ein manueller Prozess ist, lässt sich bei Let's Encrypt automatisieren.

Das Listing 3.19 gibt ein Beispiel, wie das unter 3.6.2 auf Seite 34 erzeugte Zertifikat verlängert werden kann. Dazu wird ein kurzes Bash-Skript erstellt, welches das Zertifikat erneuert und anschließend den NGINX-Dienst neu lädt, damit dieser das neue Zertifikat einlesen kann. Dieses Skript kann dann durch den Cron-Dienst zeitgesteuert automatisiert ausgeführt werden.

Listing 3.19: Verlängerung eines Zertifikats mit acme-tiny

```
1 #!/bin/bash
2 /usr/bin/python /home/johndoe/acme-tiny/acme_tiny.py --account-key /var/www/
      sites/test.my-it-brain.de/ssl/account.key --csr /var/www/sites/test.my-it-
      brain.de/ssl/test.2016/test_request.csr --acme-dir /var/www/sites/test.my-it
      -brain.de/public/.well-known/acme-challenge/ > /var/www/sites/test.my-it-
      brain.de/ssl/test.2016/2016-07-07_test.crt || exit
3 service nginx reload
```

Mit dem in Listing 3.19 dargestellten Skript wird jedoch nur das Domain-Zertifikat erneuert. Möchte man dieses zusammen mit NGINX verwenden, muss auch die Zertifikatskette neu erstellt werden. Dies kann mit den beiden Skripts realisiert werden, welche in Abschnitt 5.4 auf Seite 50 beschrieben werden.

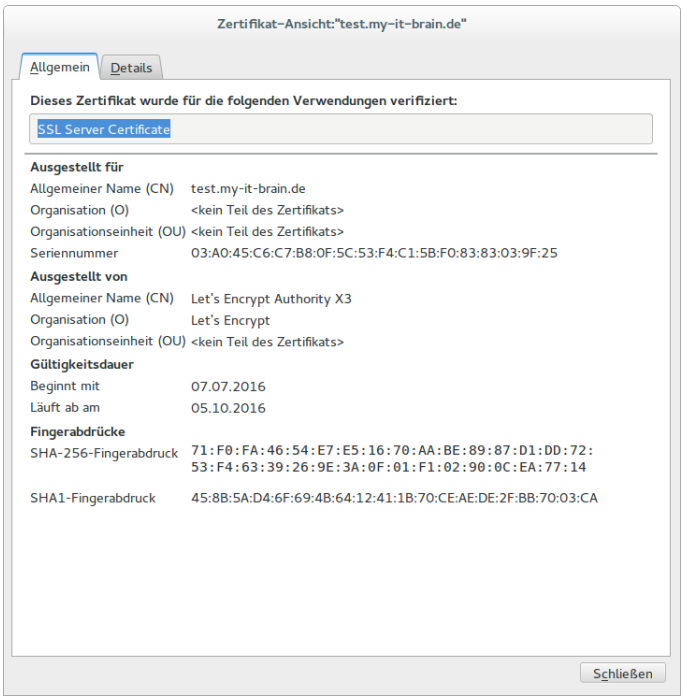

Abbildung 3.1: 2016-07-07_test.cert

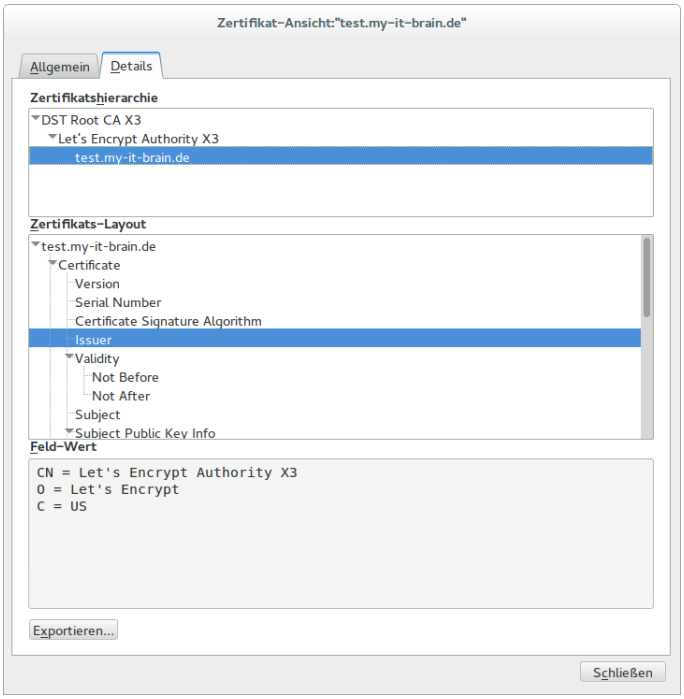

Abbildung 3.2: Details aus 2016-07-07_test.cert

4 Implementierung von HPKP

Dieses Kapitel beschreibt, wie man Certificate Pinning auf dem eigenen Server konfigurieren kann. Es wird dabei auf notwendige Vorüberlegungen eingegangen und die Berechnung der PINs beschrieben. Im Anschluss wird das Certificate Pinning beispielhaft an der Domain test.my-it-brain.de demonstriert. Hierfür wurde ein Ubuntu 14.04 LTS Server mit einem NGINX-Webserver verwendet.

4.1 Vorüberlegungen

Zu Beginn stellt sich die Frage, welchen öffentlichen Schlüssel man „pinnen“ möchte. Denn Certificate Pinning kann auf jeden öffentlichen Schlüssel der gesamten Zertifizierungskette angewendet werden. So kommen sowohl das eigene Serverzertifikat, als auch das der Intermediate CAs und das der Root CA in Frage [vgl. 24, S. 122, Spalte 1].

Pinnt man den öffentlichen Schlüssel einer Intermediate CA oder gar einer Root CA, so werden alle von dieser Zertifizierungsstelle ausgestellten Zertifikate vom Browser als gültig und sicher akzeptiert. Daher erscheint es am sichersten, wenn man den öffentlichen Schlüssel des eigenen Serverzertifikats festnagelt. Hierbei muss allerdings folgendes Risiko beachtet werden.

Wird der Schlüssel durch Kompromittierung unbrauchbar, oder geht durch Hardwaredefekt verloren, können Benutzer die gesicherten Dienste eventuell nicht mehr nutzen. Denn die Browser haben den PIN gespeichert und werden vor Ablauf der Lebensdauer keinen neuen PIN akzeptieren. Die Lebensdauer kann je nach Konfiguration mehrere Wochen bis Monate betragen [siehe 24, S. 122, Spalte 2].

Um das im vorigen Absatz beschriebene Risiko zu minimieren, definiert RFC 7469 die zusätzliche Verwendung eines Backup-PINs. Dabei wird ein Hash-Wert über den öffentlichen Schlüssel eines Schlüsselpaares gebildet, welches aktuell noch nicht genutzt wird [7, Abschnitt 4.3]. Dies kann zum Beispiel dadurch erreicht werden, dass ein Hash-Wert für den öffentlichen Schlüssel eines Certificate Signing Request (vgl. [16]) generiert wird.

Der Backup-PIN wird ebenfalls über ein HTTP-Header-Feld ausgeliefert und vom Browser eines Clients gespeichert. Der erzeugte Backup-CSR sollte sicher offline aufbewahrt werden. Er kann genutzt werden, um ein neues Zertifikat für die entsprechende Domain ausstellen zu lassen. Auf diese Weise können Zertifikate verlängert bzw. erneuert werden, ohne dass der Zugriff auf die entsprechende Domain unterbrochen wird.

4.2 Backupstrategie

Der für den Backup-PIN verwendete CSR ist sicher aufzubewahren. Denn mit ihm kann ein Zertifikat für eine Domain bzw. einen Host ausgestellt werden, welches von Clients als gültig anerkannt wird, da sie den dazugehörigen Backup-PIN gespeichert haben.

Hinweis: Neben dem Backup-CSR muss selbstverständlich auch der dazugehörige private Schlüssel sicher verwahrt werden. Nur zusammen mit diesem lässt sich das Zertifikat später auch verwenden.

Ich persönlich speichere den CSR und den dazugehörigen privaten Schlüssel in KeePassX[1]. Auf diese Weise werden die Informationen sicher in einer Datenbank mit einer 256 Bit AES-Verschlüsselung abgelegt. Die KeePassX-Datenbank selbst bewahre ich in einem TeamDrive[2]-Space auf. Dadurch wird die Datenbank sicher auf meine Endgeräte synchronisiert, sodass die Daten auch bei Ausfall eines Endgerätes erhalten bleiben.

Die lokalen Datenträger der zur TeamDrive-Synchronisation verwendeten Endgeräte sind ebenfalls verschlüsselt. So sind die Daten im Falle eines Diebstahls eines Endgerätes gleich doppelt geschützt. Erstens durch die Verschlüsselung des lokalen Datenträgers und zweitens durch die verschlüsselte KeePassX-Datenbank.

4.3 Die PINs berechnen

Die benötigten PINs werden auf der Kommandozeile mit *openssl* generiert [siehe 7, Appendix A]. Für ein existierendes SSL-Zertifikat kann die PIN mit folgendem Code erzeugt werden:

Listing 4.1: PIN-Berechnung über einem Zertifikat

```
1 openssl x509 -noout -in certificate.pem -pubkey | openssl asn1parse -noout -
      inform pem -out public.key
2 openssl dgst -sha256 -binary public.key | openssl enc -base64
```

Der erzeugte PIN wird auf der Standardausgabe ausgegeben und endet immer auf das Zeichen „=“.

Für den Backup-PIN wird zunächst ein neuer privater Schlüssel und Certificate Signing Request erstellt. Über diesen wird nun ebenfalls ein PIN mit folgendem Code erzeugt:

Listing 4.2: PIN-Berechnung über einem CSR

```
1 openssl req -noout -in example.com.csr -pubkey | openssl asn1parse -noout -
      inform pem -out example.com_csr.key
2 openssl dgst -sha256 -binary example.com_csr.key | openssl enc -base64
```

Der erzeugte Backup-PIN wird ebenfalls auf der Standardausgabe ausgegeben und muss ebenfalls auf das Zeichen „=“ enden.

Hinweis: Es ist darauf zu achten nicht die im letzten Schritt erstellten Dateien zu überschreiben. Am besten legt man die Dateien für den CSR und den Backup-CSR in unterschiedlichen Verzeichnissen ab.

[1] The Official KeePassX Homepage – `https://www.keepassx.org/` [Letzter Abruf: 26.07.2016]
[2] TeamDrive Homepage – `https://www.teamdrive.com/de/` [Letzter Abruf: 26.07.2016]

4.4 Konfiguration von NGINX

Sind die PINs erzeugt, kann anschließend der Webserver konfiguriert werden, um diese auszuliefern. Neben der PIN-Direktive [7, Abschnitt 2.1.1] ist hier die Max-Age-Direktive [7, Abschnitt 2.1.2] von Bedeutung. Letztere gibt die Zeit in Sekunden an, für die ein PIN durch den Browser gespeichert wird.

Für die ersten Tests sollte die Max-Age-Direktive auf wenige Minuten gesetzt werden. Dieser Wert sollte erst bei reibungslosem Betrieb auf mehrere Wochen erhöht werden. Dadurch wird das Risiko minimiert, bei etwaigen Fehlern für längere Zeit nicht mehr auf die durch „Pinning“ geschützte Seite zugreifen zu können.

Das zusätzliche Header-Feld muss im SSL-Block der NGINX-Konfiguration hinzugefügt werden. Dazu wird folgender Code in den SSL-Block eingefügt:

```
1 add_header Public-Key-PINs 'pin-sha256="PRIMARY-PIN"; pin-sha256="BACKUP-PIN";
      max-age=300; includeSubDomains';
```

Dabei sind PRIMARY-PIN und BACKUP-PIN durch die entsprechenden PINs zu ersetzen. Die optionale includeSubDomains-Direktive [7, Abschnitt 2.1.3] gibt an, dass die übermittelten PINs auch für alle Subdomains des Hosts gelten.

Anschließend muss die Konfiguration neu geladen werden, um diese zu aktivieren:

```
1 sudo service nginx reload
```

4.5 Validierung der Backupstrategie

Die in Abschnitt 4.2 auf der vorherigen Seite aufgestellte Backupstrategie basiert auf der Annahme, dass ein PIN über einen CSR gebildet wird. Mit diesem soll es möglich sein, sich im Bedarfsfall ein neues Zertifikat von einer Zertifizierungsstelle ausstellen zu lassen. In diesem Abschnitt möchte ich zeigen, dass die genannte Backupstrategie funktioniert. Die Validierung erfolgt in einer Laborumgebung, welche im Folgenden beschrieben wird.

4.5.1 Laborumgebung

Als Laborumgebung kommt ein Ubuntu Server 14.04 LTS zum Einsatz. Auf diesem wird ein NGINX Webserver und eine PHP-FPM-Umgebung installiert. Es wird eine Testkonfiguration erstellt und mit einem SSL-Zertifikat von CAcert[3] abgesichert.

Der private Schlüssel und der CSR werden dabei mit dem folgenden Befehl erzeugt:

```
1 openssl genrsa -out test.my-it-brain.de.key 2048
2 openssl req -new -key test.my-it-brain.de.key -out test.my-it-brain.de.csr
```

[3] http://cacert.org

Für den Backup-CSR wird analog verfahren:

```
openssl genrsa -out test.my-it-brain.de.bak.key 2048
openssl req -new -key test.my-it-brain.de.key -out test.my-it-brain.de.bak.csr
```

Das Certificate Pinning wird wie im Abschnitt 4.3 und 4.4 auf der vorherigen Seite beschrieben durchgeführt. Dabei wird lediglich darauf verzichtet auch sämtliche Subdomains mit einzuschließen. Konkret bedeutet dies, dass die Angabe von `includeSubDomains` in der NGINX-Konfiguration entfällt. Die maximale Gültigkeitsdauer wird auf 1 Stunde gesetzt.

Das Zertifikat wird anschließend unter dem Namen `test.my-it-brain.de.crt` gespeichert. Die Konfiguration der Testseite im NGINX sieht im konkreten Fall wie folgt aus. Es werden nur die relevanten Zeilen wiedergegeben:

```
## BEGIN Testlabor CONFIGURATION ##############################################
server {
        listen 80;
        listen [::]:80;
        server_name test.my-it-brain.de;

        # Path to the root of your installation
        root /var/www/test.my-it-brain.de/public;

}
server {
    # Listen on Port 443
    listen 443 ssl;
    listen [::]:443 ssl;
    server_name test.my-it-brain.de;

    ssl_certificate /var/www/test.my-it-brain.de/ssl/test.my-it-brain.de.crt;
    ssl_certificate_key /var/www/test.my-it-brain.de/ssl/test.my-it-brain.de.key
        ;

        # Add headers to serve security related headers
        add_header Public-Key-Pins 'pin-sha256="<HIER STEHT DER PIN"; pin-sha256
            ="HIER STEHT DER BACKUP-PIN"; max-age=3600';

    # Path to the root of your installation
    root /var/www/$host/public;

    error_page 403 /core/templates/403.php;
    error_page 404 /core/templates/404.php;
}
## END Testlabor CONFIGURATION ################################################
```

4.5.2 Test der Backupstrategie

Sind PIN und Backup-PIN eingerichtet und ist die Testseite über HTTPS abrufbar, so kann die Backupstrategie getestet werden. Dazu wird mit dem Backup-CSR ein Zertifikat beantragt. Für den hier geschilderten konkreten Fall wird dazu das Zertifikat bei CAcert widerrufen und mit dem Backup-CSR ein neues Zertifikat beantragt. Das neue Zertifikat wird parallel zum alten unter dem Namen `test.my-it-brain.de.bak.crt` auf dem Server gespeichert.

Um nun die Validierung durchzuführen, wird das neue Zertifikat in die NGINX-Konfiguration eingetragen und der PIN des alten Zertifikats gelöscht. Jetzt ist nur noch ein PIN in der Konfiguration hinterlegt. Dies verstößt zwar formal gegen den RFC 7469, da die Laborumgebung anschließend jedoch wieder abgerissen wird, geht dies in Ordnung. Die Konfiguration sollte nun wie folgt aussehen:

```
## BEGIN Testlabor CONFIGURATION ############################################
server {
        listen 80;
        listen [::]:80;
        server_name test.my-it-brain.de;

        # Path to the root of your installation
        root /var/www/test.my-it-brain.de/public;

}
server {
    # Listen on Port 443
    listen 443 ssl;
    listen [::]:443 ssl;
    server_name test.my-it-brain.de;

    ssl_certificate /var/www/test.my-it-brain.de/ssl/test.my-it-brain.de.bak.crt
        ;
    ssl_certificate_key /var/www/test.my-it-brain.de/ssl/test.my-it-brain.de.bak
        .key;

        # Add headers to serve security related headers
        add_header Public-Key-Pins 'pin-sha256="HIER␣STEHT␣DER␣BACKUP-PIN"; max-
            age=3600';

    # Path to the root of your installation
    root /var/www/$host/public;

    error_page 403 /core/templates/403.php;
    error_page 404 /core/templates/404.php;
}
## END Testlabor CONFIGURATION ##############################################
```

Nach Neuladen der Konfiguration zeigt eine Prüfung, dass weiterhin auf die Webseite zugegriffen werden kann. Damit wurde die Backupstrategie erfolgreich validiert.

4.5.3 Gegenprobe

Ich gehe noch einen Schritt weiter und führe auch die Gegenprobe durch, indem ein Man-In-The-Middle-Angriff simuliert wird. Dazu erstelle ich einen weiteren privaten Schlüssel und einen neuen CSR, mit welchem ich ein weiteres Zertifikat auf `test.my-it-brain.de` ausstellen lasse. In der Theorie sollte der Webbrowser, welcher den bzw. die PINs gespeichert hat, den Zugriff auf die Webseite verweigern, wenn dieses neue Zertifikat in die Seite eingebunden wird. Denn schließlich passt der im Webbrowser gespeicherte PIN nicht zu dem, der über das neue Zertifikat gebildet werden kann.

Die Erstellung des CSR für das „böse“ Zertifikat erfolgt mit folgenden Befehlen:

```
        openssl genrsa -out trudy.key 2048
        openssl req -new -key trudy.key -out trudy.csr
```

Das damit erzeugte „böse“ Zertifikat wird in die Laborumgebung eingebaut:

```
## BEGIN Testlabor CONFIGURATION ############################################
server {
        listen 80;
        listen [::]:80;
        server_name test.my-it-brain.de;

        # Path to the root of your installation
        root /var/www/test.my-it-brain.de/public;

}
server {
    # Listen on Port 443
```

```
13      listen 443 ssl;
14      listen [::]:443 ssl;
15      server_name test.my-it-brain.de;
16
17      ssl_certificate /var/www/test.my-it-brain.de/ssl/trudy.crt;
18      ssl_certificate_key /var/www/test.my-it-brain.de/ssl/trudy.key;
19
20      # Path to the root of your installation
21      root /var/www/$host/public;
22
23      error_page 403 /core/templates/403.php;
24      error_page 404 /core/templates/404.php;
25  }
26  ## END Testlabor CONFIGURATION ###############################################
```

Nachdem die NGINX-Konfiguration neu geladen wurde, verweigert der Webbrowser einen erneuten Aufruf der Testseite mit dem Hinweis, dass der Server keine Zertifikatskette verwendet, die mit dem PIN-Set übereinstimmt (siehe 4.1).

Abbildung 4.1: Key Pinning Failure

Damit wurde erfolgreich gezeigt, dass der Webbrowser den Zugriff auf die Webseite verweigert, wenn die Zertifikatskette nicht zum „gepinnten“ Schlüssel passt.

4.6 Fazit

Certificate Pinning wird heute bereits von den weit verbreiteten Browsern Chrome und Firefox unterstützt wird. Auch die Konfiguration weit verbreiteter Webserver wie Apache, lighttpd und NGINX ist mit geringem Aufwand möglich, wie hier am Beispiel von NGINX gezeigt wurde.

Die sichere Aufbewahrung des Backup-CSR ist zu gewährleisten, um eine längere Nichterreichbarkeit einer Domain vermeiden zu können. Mit diesem lässt sich

im Bedarfsfall ein neues Zertifikat erzeugen und in die betroffene Webseite einbinden. Im Abschnitt 4.5 auf Seite 40 konnte die Backupstrategie für das Certificate Pinning validiert werden. Darüber hinaus konnte mit der Gegenprobe in Abschnitt 4.5.3 auf Seite 42 auch die generelle Funktionalität des Verfahrens nachgewiesen werden.

Damit ist Certificate Pinning grundsätzlich geeignet, die verschlüsselte Kommunikation im Internet noch sicherer zu machen.

5 Fallbeispiel: Let's Encrypt und HPKP

Bis hier her wurden in diesem Kochbuch Rezepte für die Erstellung von Zertifikatsanfragen, die Beantragung von Zertifikaten, die Implementierung von TLS/SSL und HPKP wiedergegeben. Im vorliegenden Kapitel werden diese Rezepte zu einem mehrgängigen Menü arrangiert. Als Fallbeispiel dient dabei die Domain `test.my-it-brain.de`, anhand der folgende Punkte gezeigt werden:

1. Konfiguration von NGINX zur Auslieferung einer HTTP-Testseite unter der genannten Domain
2. Erstellung der erforderlichen Certificate Signing Requests mit OpenSSL
3. Implementierung des Let's Encrypt Clients „acme-tiny“ und eines Skripts zur automatisierten Erneuerung der Zertifikate
4. Implementierung von TLS, HSTS und HPKP für die genannte Domain

Die in den vorangegangenen Kapiteln vorgestellten Methoden sollen hier in einen Workflow überführt werden, welcher zur Orientierung bei ähnlichen Implementierungen dienen kann. Das Fallbeispiel zeigt dabei, wie der Prozess der Zertifikatserneuerung automatisiert werden kann, wodurch sich der Wartungsaufwand für TLS-Implementierungen minimieren lässt.

5.1 Konfiguration von NGINX für die Test-Domain

Für den zu erstellenden HTTP-Host wird eine einfache HTML-Datei im Verzeichnis /var/www/sites/test.my-it-brain.de/public/ erstellt:

Listing 5.1: /var/www/sites/test.my-it-brain.de/public/index.html

```
1  <!DOCTYPE html>
2  <html lang="de">
3    <head>
4      <meta charset="utf-8">
5      <title>Testlabor My-IT-Brain.de</title>
6    </head>
7    <body>
8          Hier befindet sich das Testlabor von My-IT-Brain.de.
9
10         Hier gibt es nichts besonderes zu sehen.
11   </body>
12 </html>
```

Anschließend wird der HTTP-Host für NGINX in der Datei /etc/nginx/sites-available/testlab erstellt:

Listing 5.2: /etc/nginx/sites-available/testlab

```
##########################################################################
#                                                                        #
# NGINX - VHOSTS CONFIGURATION                                           #
#                                                                        #
# Author : Joerg Kastning <webmaster(aet)my-it-brain(Punkt)de>           #
# Site: http://test.my-it-brain.de                                       #
# Version: 2016-07-27                                                    #
#                                                                        #
##########################################################################

## BEGIN Testlabor CONFIGURATION ###################################
server {
        listen 80;
        listen [::]:80;
        server_name test.my-it-brain.de;

    # Path to the root of your installation
    root /var/www/sites/$host/public;

    index index.html;
    error_page 403 /core/templates/403.php;
    error_page 404 /core/templates/404.php;
}
## END Testlabor CONFIGURATION #####################################
```

Um nun den HTTP-Host zu aktivieren, ist ein symbolischer Link im Verzeichnis /etc/nginx/sites-enabled/ zu erstellen und NGINX einmal neu zu laden:

Listing 5.3: Aktivierung des HTTP-Host

```
# cd /etc/nginx/sites-enabled
# ln -s /etc/ngnix/sites-available/jk_testlab jk_testlab
# service nginx reload
```

Wurde alles richtig gemacht, sollte sich nun die konfigurierte Seite im Webbrowser aufrufen lassen:

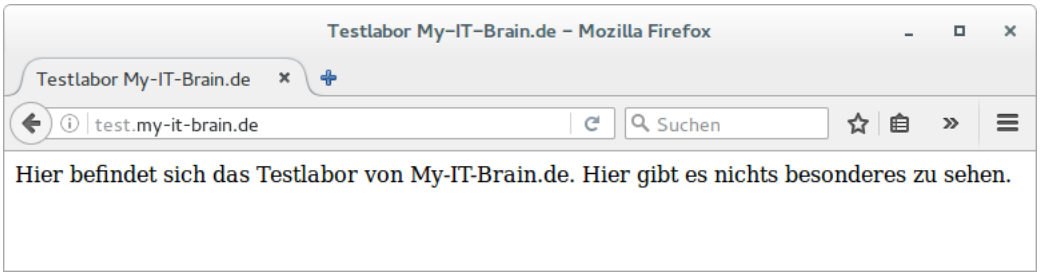

Abbildung 5.1: Webseite test.my-it-brain.de

5.2 Erstellung des CSR mit OpenSSL

Listing 5.4 auf der nächsten Seite zeigt die Erstellung von insgesamt drei Certificate Signing Requests. Einer wird zur Beantragung des TLS-Zertifikats verwendet, die beiden weiteren dienen der Generierung von Backup-Pins für die HPKP-Implementierung. Es wird jeweils ein Backup-CSR mit 2048 Bit und 4096 Bit erstellt. Dies hat den Hintergrund, dass 2048 Bit evtl. nicht mehr ausreichend sicher sind, wenn der Backup-CSR benötigt wird.

Ich erstelle die Certificate Signing Requests nicht auf dem Webserver, sondern auf einem meiner eigenen Rechner. Auf den Webserver wird nur der private Schlüssel (ohne Passphrase) und der CSR für die Produktion in das Verzeichnis /var/www/sites/test.my-it-brain.de/ssl hochgeladen. Die auf dem lokalen Rechner vorhandenen Dateien werden sicher gespeichert (vgl. Abschnitt 4.2 auf Seite 39).

Listing 5.4: Erstellung der benötigten CSR mit OpenSSL

```
# Production-CSR
openssl genrsa -aes256 -out test.my-it-brain.de.pass 2048
Generating RSA private key, 2048 bit long modulus
.....................................................................+++
.................................................+++
e is 65537 (0x10001)
Enter pass phrase for test.my-it-brain.de.pass: <passphrase>
Verifying - Enter pass phrase for test.my-it-brain.de.pass: <passphrase>

openssl rsa -in test.my-it-brain.de.pass -out test.my-it-brain.de.key
Enter pass phrase for test.my-it-brain.de.pass: <passphrase>
writing RSA key

openssl req -batch -sha256 -new -key test.my-it-brain.de.key -out test.my-it-brain.de.csr -subj "/C=DE/L=Musterstadt/O=Musterfirma/OU=Musterabteilung/CN=test.my-it-brain.de/emailAddress=webmaster@my-it-brain.de"

# Backup-CSR-1
mdir -p backup2048
cd backup2048
openssl genrsa -aes256 -out test.my-it-brain.de.pass 2048
Generating RSA private key, 2048 bit long modulus
.....................................................................+++
.................................................+++
e is 65537 (0x10001)
Enter pass phrase for test.my-it-brain.de.pass: <passphrase>
Verifying - Enter pass phrase for test.my-it-brain.de.pass: <passphrase>

openssl rsa -in test.my-it-brain.de.pass -out test.my-it-brain.de.key
Enter pass phrase for test.my-it-brain.de.pass: <passphrase>
writing RSA key

openssl req -batch -sha256 -new -key test.my-it-brain.de.key -out test.my-it-brain.de.csr -subj "/C=DE/L=Musterstadt/O=Musterfirma/OU=Musterabteilung/CN=test.my-it-brain.de/emailAddress=webmaster@my-it-brain.de"

# Backup-CSR-2
cd ..
mdir -p backup4096
cd backup4096
openssl genrsa -aes256 -out test.my-it-brain.de.pass 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................+++
.................................................+++
e is 65537 (0x10001)
Enter pass phrase for test.my-it-brain.de.pass: <passphrase>
Verifying - Enter pass phrase for test.my-it-brain.de.pass: <passphrase>

openssl rsa -in test.my-it-brain.de.pass -out test.my-it-brain.de.key
Enter pass phrase for test.my-it-brain.de.pass: <passphrase>
writing RSA key

openssl req -batch -sha256 -new -key test.my-it-brain.de.key -out test.my-it-brain.de.csr -subj "/C=DE/L=Musterstadt/O=Musterfirma/OU=Musterabteilung/CN=test.my-it-brain.de/emailAddress=webmaster@my-it-brain.de"
```

In Vorbereitung auf die HPKP-Implementierung werden an dieser Stelle die PINs für die Certificate Signing Requests aus Listing 5.4 erstellt. Dazu wird das Kommando aus Listing 4.2 auf Seite 39 verwendet.

5.3 Beantragung und Implementierung des TLS-Zertifikats mit HSTS und HPKP

Die Installation des Let's Encrypt Clients acme-tiny [22] wurde bereits in Abschnitt 3.6.2 auf Seite 34 beschrieben. Die Listings 3.16, 3.17 und 3.18 gelten wie angegeben für dieses Fallbeispiel. Der Code aus Listing 5.5 veranschaulicht, wie das Zertifikat beantragt und ausgestellt wird. Das TLS-Zertifikat wird im gleichen Verzeichnis wie der private Schlüssel und der CSR gespeichert.

Listing 5.5: Beantragung und Ausstellung des TLS-Zertifikats

```
1  $ python acme-tiny/acme_tiny.py --account-key /var/www/sites/test.my-it-brain.de
       /ssl/account.key --csr /var/www/sites/test.my-it-brain.de/ssl/test.my-it-
       brain.de.csr --acme-dir /var/www/sites/test.my-it-brain.de/public/.well-
       known/acme-challenge/ > /var/www/sites/test.my-it-brain.de/ssl/test.my-it-
       brain.de.crt
2  Parsing account key...
3  Parsing CSR...
4  Registering account...
5  Already registered!
6  Verifying test.my-it-brain.de...
7  test.my-it-brain.de verified!
8  Signing certificate...
9  Certificate signed!
```

Das ausgestellte Zertifikat wird im nächsten Schritt mit dem Intermediate-Zertifikat der CA kombiniert. Das Intermediate-Zertifikat wird dazu von der Webseite der CA[1] heruntergeladen (siehe Listing 5.6).

Listing 5.6: Erstellung der Zertifikatskette

```
1  $ cd /var/www/sites/test.my-it-brain.de/ssl/
2
3  $ wget https://letsencrypt.org/certs/lets-encrypt-x3-cross-signed.pem
4  --2016-07-31 19:32:27--  https://letsencrypt.org/certs/lets-encrypt-x3-cross-
       signed.pem
5  Resolving letsencrypt.org (letsencrypt.org)... 2a02:26f0:64:59e::2a1f, 2a02:26f0
       :64:590::2a1f, 104.109.115.176
6  Connecting to letsencrypt.org (letsencrypt.org)|2a02:26f0:64:59e::2a1f|:443...
       connected.
7  HTTP request sent, awaiting response... 200 OK
8  Length: 1647 (1,6K) [application/x-x509-ca-cert]
9  Saving to:  lets -encrypt-x3-cross-signed. p e m
10
11 100%[=============================================================>] 1.647
       --.-K/s   in 0s
12
13 2016-07-31 19:32:27 (274 MB/s) -  lets -encrypt-x3-cross-signed. p e m  saved
       [1647/1647]
14
15 $ cat test.my-it-brain.de.crt lets-encrypt-x3-cross-signed.pem >chain.crt
```

Die Konfiguration in /etc/nginx/sites-available/testlab wird erweitert, um die Implementierung von TLS für die Domain abzuschließen. Die vollständige Konfiguration wird in Listing 5.7 auf der nächsten Seite dargestellt. Die Angabe der Zertifikats-Datei und des privaten Schlüssels erfolgt dabei in den Zeilen 42 und 43.

[1] `https://letsencrypt.org/certificates/` [Letzter Abruf: 31.07.2016]

Listing 5.7: Vollständige Konfigurationsdatei für test.my-it-brain.de

```
1  ################################################################################
2  #                                                                              #
3  # NGINX - VHOSTS CONFIGURATION                                                 #
4  #                                                                              #
5  # Author : Joerg Kastning <webmaster(aet)my-it-brain(Punkt)de>                 #
6  # Site: http://test.my-it-brain.de                                             #
7  # Version: 2016-07-31                                                          #
8  #                                                                              #
9  ################################################################################
10 ## BEGIN Testlabor CONFIGURATION ###############################################
11 server {
12         listen 80;
13         listen [::]:80;
14         server_name test.my-it-brain.de;
15         return 301 https://$server_name$request_uri;
16
17     # Path to the root of your installation
18     root /var/www/sites/$host/public;
19
20     index index.html;
21     error_page 403 /core/templates/403.php;
22     error_page 404 /core/templates/404.php;
23 }
24 server {
25     # Listen on Port 443
26     listen 443 ssl;
27     listen [::]:443 ssl;
28     server_name test.my-it-brain.de;
29
30     # Path to the root of your installation
31     root /var/www/sites/$host/public;
32
33     # Path to the logfiles
34     access_log /var/www/jkastning/sites/logs/test.my-it-brain.de_access.log
           combined;
35     error_log /var/www/jkastning/sites/logs/test.my-it-brain.de_error.log error;
36
37     index index.html;
38     error_page 403 /core/templates/403.php;
39     error_page 404 /core/templates/404.php;
40
41     ssl_certificate /var/www/sites/test.my-it-brain.de/ssl/chain.crt;
42     ssl_certificate_key /var/www/sites/test.my-it-brain.de/ssl/test.my-it-brain.
           de.key;
43
44     ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE
           -RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-
           SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:
           ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:
           ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:
           ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-
           AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-
           SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:
           AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!aNULL:!eNULL:!EXPORT:!
           DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-RSA-DES-CBC3-SHA:!
           KRB5-DES-CBC3-SHA';
45
46     ssl_prefer_server_ciphers on;
47     ssl_dhparam /etc/nginx/dh_params.pem;
48
49         # Add headers to serve security related headers
50         add_header Strict-Transport-Security "max-age=15768000;␣
               includeSubDomains;␣preload;";
51         add_header X-Content-Type-Options nosniff;
52         add_header X-Frame-Options "SAMEORIGIN";
53         add_header X-XSS-Protection "1;␣mode=block";
54         add_header X-Robots-Tag none;
55         add_header Public-Key-Pins 'pin-sha256="<PIN-Production-CSR>"; pin-
               sha256="<PIN-Backup-CSR-1>"; pin-sha256="<PIN-Backup-CSR-2>"; max-
               age=15768000';
56
57         # Disable gzip to avoid the removal of the ETag header
58         gzip off;
59 }
60 ## END Testlabor CONFIGURATION #################################################
```

Mit Zeile 16 in Listing 5.7 wurde eine Weiterleitung eingefügt, welche Client-Anfragen automatisch auf die HTTPS-Version der Webseite weiterleitet. Dies ist die Voraussetzung für die erfolgreiche Implementierung des HSTS-Headers in Zeile 51. Die Implementierung des HPKP-Headers erfolgt in Zeile 56. Hier werden die PINs für sämtliche in Listing 5.4 auf Seite 47 erstellten öffentlichen Schlüssel (CSR) angegeben.

Der Dienst muss mit dem Kommando `sudo service nginx reload` neu eingelesen werden, um die neue Konfiguration zu aktivieren.

5.4 Verlängerung des TLS-Zertifikats mittels SmartRenew

Für die automatische Verlängerung des TLS-Zertifikats werden zwei kurze Skripte benötigt. Ein Skript zur Ausstellung und Implementierung des neuen Zertifikats (siehe Listing 5.8) und das SmartRenew-Skript von Scott Helme [10]. Das SmartRenew-Skript kann von GitHub[2] auf den Webserver heruntergeladen werden. Anschließend ist es wie in Listing 5.9 dargestellt anzupassen.

Hinweis: Beide Skripte sind vor ihrer ersten Benutzung ausführbar zu machen.

Listing 5.8: Skript: le_oc_cert_renew.sh

```
1  #!/bin/bash
2  # Datum: 2016-07-25
3  # Autor: Joerg Kastning <webmaster(aet)my-it-brain(Punkt)de>
4  #
5  # Beschreibung:
6  # Dieses Skript dient der erneuerung des TLS-Zertifikats fuer
7  # oc.my-it-brain.de  ber  Let's Encrypt mit dem Client acme-tiny.
8
9  python acme-tiny/acme_tiny.py --account-key /var/www/sites/test.my-it-brain.de/
       ssl/account.key --csr /var/www/sites/test.my-it-brain.de/ssl/test.my-it-
       brain.csr --acme-dir /var/www/sites/test.my-it-brain.de/public/.well-known/
       acme-challenge/ > /var/www/sites/test.my-it-brain.de/ssl/test.my-it-brain.
       crt
10
11 cat /var/www/sites/test.my-it-brain.de/ssl/test.my-it-brain.crt /var/www/sites/
       test.my-it-brain.de/ssl/lets-encrypt-x3-cross-signed.pem > /var/www/sites/
       test.my-it-brain.de/ssl/chain.crt
12
13 sudo service nginx reload
```

Listing 5.9: Skript: smartrenew.sh

```
1  #!/bin/bash
2
3  # Get the current date as seconds since epoch.
4  NOW=$(date +%s)
5  # Get the expiry date of our certificate.
6  EXPIRE=$(openssl x509 -in /var/www/sites/test.my-it-brain.de/ssl/test.my-it-
       brain.crt -noout -enddate)
7  # Trim the unecessary text at the start of the string.
8  EXPIRE=${EXPIRE#*=}
9  # Convert the expiry date to seconds since epoch.
10 EXPIRE=$(date --date="$EXPIRE" +%s)
11 # Calculate the time left until the certificate expires.
12 LIFE=$((EXPIRE-NOW))
13 # The remaining life on our certificate below which we should renew (7 days).
14 RENEW=604800
15 # If the certificate has less life remaining than we want.
16 if (($LIFE < $RENEW))
```

[2] `https://github.com/ScottHelme/Lets-Encrypt-Smart-Renew` [Letzter Abruf: 31.07.2016]

```
17          then
18                  # Then call the renewal script.
19                  /path/to/le_oc_cert_renew.sh
20  fi
```

Das Skript `smartrenew.sh` kann nun z. B. stündlich über den CRON-Dienst ausgeführt werden. Bei jedem Aufruf wird geprüft, ob die Gültigkeit des Zertifikats den Schwellwert von 7 Tagen erreicht hat. Ist dies der Fall, wird das Skript `le_oc_cert_renew.sh` aufgerufen, um das genutzte Zertifikat zu erneuern.

6 Schlussworte

Mir persönlich hat der vorliegende Text geholfen, mich detailliert mit der Thematik rund um TLS/SSL-Zertifikate auseinanderzusetzen, mein Wissen zu erweitern und zu vertiefen. Es freut mich, wenn dieser Text auch Ihnen das Verständnis des Themas etwas näher bringen konnte.

Ein großes Dankeschön geht an dieser Stelle an meine Frau Barbara und meinen Arbeitskollegen Dr. Maik Stührenberg für Korrektur, Ratschläge zum Text und so manchen LaTeX-Tipp.

Möchten Sie weitere Themen aus dem Gebiet der TLS/SSL-Verschlüsselung in diesem Rahmen behandelt wissen, senden Sie mir Ihre Vorschläge gern via E-Mail an: *tls-rezepte@my-it-brain.de*

Abbildungsverzeichnis

2.1 Symmetrische Verschlüsselung [20, S. 6] 3
2.2 Asymmetrische Verschlüsselung [13, S. 51] 4
2.3 Digitale Signatur [13, S. 51] 4
2.4 Asymmetrische Verschlüsselung beim Nachrichtenversand von Bob an Alice [20, S. 13] 5
2.5 TLS Handshake mit Server-Authentifizierung [20, S. 27] 6
2.6 Struktur einer Zertifikatskette [20, S. 71] 9

3.1 2016-07-07_test.cert 37
3.2 Details aus 2016-07-07_test.cert 37

4.1 Key Pinning Failure 43

5.1 Webseite test.my-it-brain.de 46

Listings

2.1 Zertifikatskette der HRZ Uni Bielefeld CA 10
2.2 OpenSSL Changelog 12

3.1 Generierung des privaten Schlüssels 17
3.2 Ausgabe des privaten Schlüssels (verschlüsselt) 17
3.3 Passphrase entfernen 18
3.4 Interaktive CSR-Erstellung 18
3.5 CSR-Erstellung mit Parametern 19
3.6 CSR-Ausgabe im Klartext 19
3.7 make-cert.sh 20
3.8 Durch make-cert.sh erstellte Dateien 22
3.9 Beispiel einer Konfigurationsdatei 23
3.10 OpenSSL-Kommando zur Erstellung des CSR 24
3.11 make-cert-Aufruf zur Erstellung des CSR 25
3.12 Beispielkonfiguration eines VirtualHost 30
3.13 Beispielkonfiguration eines HTTPS-Servers in NGINX 30
3.14 Erstellung der Zertifikatskette 30
3.15 Erstellung eines DV-Zertifikats für test.my-it-brain.de 33
3.16 Installation des Let's Encrypt Clients acme-tiny 34
3.17 Generierung eines Account-Keys 34
3.18 Verzeichnis für die Acme-Challenge 35
3.19 Verlängerung eines Zertifikats mit acme-tiny 36

4.1 PIN-Berechnung über einem Zertifikat 39
4.2 PIN-Berechnung über einem CSR 39

5.1 /var/www/sites/test.my-it-brain.de/public/index.html 45
5.2 /etc/nginx/sites-available/testlab 46
5.3 Aktivierung des HTTP-Host 46
5.4 Erstellung der benötigten CSR mit OpenSSL 47
5.5 Beantragung und Ausstellung des TLS-Zertifikats 48
5.6 Erstellung der Zertifikatskette 48
5.7 Vollständige Konfigurationsdatei für test.my-it-brain.de 49
5.8 Skript: le_oc_cert_renew.sh 50
5.9 Skript: smartrenew.sh 50

Literatur

[1] J. Callas u. a. *OpenPGP Message Format*. IETF RFC 4880. The Internet Engineering Task Force (IETF®), 2007. URL: https://tools.ietf.org/html/rfc4880.

[2] D. Cooper u. a. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. IETF RFC 5280. The Internet Engineering Task Force (IETF®), 2008. URL: https://tools.ietf.org/html/rfc5280 (besucht am 12. 12. 2015).

[3] T. Dierks und E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF RFC 5246. The Internet Engineering Task Force (IETF®), 2008. URL: https://tools.ietf.org/html/rfc5246 (besucht am 12. 12. 2015).

[4] *Dokumentation für den lighttpd Webserver*. [Letzter Abruf: 02.07.2016]. URL: https://redmine.lighttpd.net/projects/lighttpd/wiki.

[5] *Dokumentation zum Apache HTTP Server Version 2.4*. [Letzter Abruf: 01.07.2016]. URL: https://httpd.apache.org/docs/2.4/de/.

[6] *Dovecot SSL configuration*. [Letzter Abruf: 02.07.2016]. URL: http://wiki2.dovecot.org/SSL/DovecotConfiguration.

[7] C. Evans u. a. *Public Key Pinning Extension for HTTP*. IETF RFC 7469. The Internet Engineering Task Force (IETF®), Apr. 2015. URL: https://tools.ietf.org/html/rfc7469 (besucht am 15. 11. 2015).

[8] A. Freier, P. Karlton und P. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. IETF RFC 6101. The Internet Engineering Task Force (IETF®), Apr. 2011. URL: https://tools.ietf.org/html/rfc6101 (besucht am 12. 12. 2015).

[9] Scott Helme. *Getting started with Let's Encrypt!* [Letzter Abruf: 03.07.2016]. 2015. URL: https://scotthelme.co.uk/setting-up-le/.

[10] Scott Helme. *Let's Encrypt Smart Renew*. [Letzter Abruf: 31.07.2016]. 2016. URL: https://scotthelme.co.uk/lets-encrypt-smart-renew/.

[11] J. Hodges u. a. *HTTP Strict Transport Security (HSTS)*. IETF RFC 6797. [Letzter Abruf: 26.07.2016]. Internet Engineering Task Force (IETF®), 2012. URL: https://tools.ietf.org/html/rfc6797.

[12] Jörg Kastning. *Der eigene Mailserver Teil 2*. [Letzter Abruf: 02.07.2016]. 2015. URL: http://www.my-it-brain.de/wordpress/der-eigene-mailserver-teil-2/.

[13] Charlie Kaufman, Radia Perlman und Mike Speciner. *Network security : private communication in a public world ; [now includes IPsec, SSL, PKI, AES, and web security]*. eng. 2. ed. Prentice Hall series in computer networking and distributed systems. Upper Saddle River, N.J. ; London: Prentice Hall PTR, 2002, XXVI, 713 S. : graph. Darst. ISBN: 0-13-046019-2.

[14] *Let's Encrypt: How It Works*. [Letzter Abruf: 03.07.2016]. URL: https://letsencrypt.org/how-it-works/.

[15] *nginx documentation*. [Letzter Abruf: 01.07.2016]. URL: https://nginx.org/en/docs/.

[16] M. Nystrom, B. Kaliski und RSA Security. *PKCS 10: Certification Request Syntax Specification*. IETF RFC 2986 1.7. [Letzter Abruf: 26.07.2016]. The Internet Engineering Task Force (IETF®), Nov. 2000. URL: https://tools.ietf.org/html/rfc2985.

[17] M. Nystrom, B. Kaliski und RSA Security. *PKCS 9: Selected Object Classes and Attribute Types*. IETF RFC 2985 2.0. [Letzter Abruf: 26.07.2016]. The Internet Engineering Task Force (IETF®), 2000. URL: https://tools.ietf.org/html/rfc2985.

[18] W. Polk u. a. *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. IETF RFC 3270. [Letzter Abruf: 26.07.2016]. The Internet Engineering Task Force (IETF®), Apr. 2002.

[19] B. Ramsdell u. a. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. IETF RFC 5751. [Letzter Abruf: 26.07.2016]. The Internet Engineering Task Force (IETF®), Jan. 2010. URL: https://tools.ietf.org/html/rfc5751.

[20] Ivan Ristić. *Bulletproof SSL and TLS : [understanding and deploying SSL/TLS and PKI to secure servers and web applications]*. eng. London: Feisty Duck, 2014, XXII, 506 S. : graph. Darst., Kt. ISBN: 978-1-907117-04-6.

[21] Ivan Ristić. *OpenSSL Cookbook*. Hrsg. von Jelena Girić-Ristić. Version 2.0. Feisty Duck Limited, März 2015. URL: https://www.feistyduck.com/library/openssl-cookbook/.

[22] Daniel Roesler. *A tiny script to issue and renew TLS certs from Let's Encrypt*. [Letzter Abruf: 07.07.2016]. 2015. URL: https://github.com/diafygi/acme-tiny.

[23] Jürgen Schmidt. "Festgenagelte Zertifikate: TLS wird sicherer durch Certificate Pinning". In: *c't magazin für computer technik* 23 (2015), S. 118–121.

[24] Jürgen Schmidt. "Sicher mit Pin: Zertifikats-Pinning auf dem eigenen Server". In: *c't magazin für computer technik* 23 (2015), S. 122–125.

[25] *SMTP Server specific settings*. [Letzter Abruf: 02.07.2016]. URL: http://de.postfix.org/httpmirror/TLS_README.html#server_tls.

[26] *SSL Certificate Installation in Microsoft IIS 7*. [Letzter Abruf: 03.07.2016]. URL: https://www.digicert.com/ssl-certificate-installation-microsoft-iis-7.htm.

[27] *SSL Certificate Installation in Microsoft IIS 8 and IIS 8.5*. [Letzter Abruf: 03.07.2016]. URL: https://www.digicert.com/ssl-certificate-installation-microsoft-iis-8.htm.