Praktische Experimente mit OpenSSL

Ursprünglich für Tests genutzte OpenSSL-Version: 0.9.5a

Die Beispiele sollten auch für aktuellere OpenSSL-Versionen so funktionieren.

URLs:

- OpenSSL: http://www.openssl.org/
- OpenSSL-Kurzreferenz: http://www.pca.dfn.de/bibliothek/bulletins/info/tools/openssl/1.1/
- CA-Handbuch: http://www.dfn-cert.de/dfn/berichte/db089/
- SSL-MZtelnet: ftp://ftp.uni-mainz.de/pub/software/security/ssl/SSL-MZapps/
- Fortify:
  - http://www.fortify.net/
- Stunnel:
  - http://www.stunnel.org/
- Apache und mod_ssl:
  - http://httpd.apache.org/
  - http://en.wikipedia.org/wiki/Mod_ssl

🔧 Fix Me!

Anschauen: →> http://doku.fietz.net/index.php?title=SSL-Befehle

🔧 Fix Me!

---

neue Test-CA erzeugen (mit selbstzertifziertem Zertifikat)

```
<wrap em>CA.sh -newca</wrap>
```

Bei Verwendung der Standard-Einstellungen in /usr/local/openssl/openssl.cnf wird ein Verzeichnis demoCA erstellt, in dem u.a. folgende Informationen zu finden sind:

private/cakey.pem privater Schlüssel der CA cacert.pem selbstzertifziertes Zertifikat der CA index.txt Liste der bereits ausgestellten Zertifikate serial Seriennummer für das nächste Zertifikat newcerts Verzeichnis für erstellte Zertifikate

Das hier und nachfolgend genutzte Skript CA.sh dient der Vereinfachung einiger typischer OpenSSL-Operationen, indem es die jeweils benötigten OpenSSL-Kommandos mit geeigneten Argumenten aufruft.

Bei -newca wird bei OpenSSL 1.0.0c standardmäßig folgende Kommandofolge ausgeführt:

```
# openssl req -new -keyout ./demoCA/private/cakey.pem -out
./demoCA/careq.pem
```

```
# openssl ca -create_serial -out ./demoCA/cacert.pem -days 365 -batch -
keyfile ./demoCA/private/cakey.pem \
    -selfsign -extensions v3_ca -infiles ./demoCA/careq.pem
```

Ein selbstzertifziertes Zertifikat kann man ausgehend vom Schlüsselpaar der CA auch so erstellen:

```
# openssl req -new -days 1000 -x509 -out cacert_selfsigned.pem -key
private/cakey.pem
```

---

neuen Schlüssel sowie Zertifikatsanforderung erstellen

CA.sh -newreq

Diese Kommando ist auf der Ebene des Elternverzeichnisses von demoCA auszuführen. Resultat: `newreq.pem`

Diese Datei sollte gut aufgehoben werden, da sie neben der Zertifikatsanforderung auch den neu generierten privaten Schlüssel enthält.

Zertifikatsanforderung manuell für existierenden Schlüssel erstellen:

```
# openssl req -new -key newkey.pem -out newreq.pem
```

manuelle Erzeugung eines RSA-Schlüssels (ohne Passwort):

```
# openssl genrsa -out rsa_key 1024
```

Zertifikat erstellen (Zertifikatsanforderung signieren)

CA.sh -sign

Dies entspricht bei OpenSSL 1.0.0c standardmäßg dem Kommando:

```
# openssl ca -policy policy_anything -out newcert.pem -infiles newreq.pem
```

Dieses Kommando ist auf der Ebene des Elternverzeichnisses von demoCA auszuführen. Die Zertifikatsanforderung wird in der Datei newreq.pem erwartet. Resultat: `newcert.pem`

Empfehlung: Zertifikatsanforderung, den privaten Schlüssel sowie das Zertifikat umbenennen und aufheben, z.B.:

```
# cp newreq.pem HINZ/hinz_req.pem
# cp demoCA/newkey.pem HINZ/hinz_key.pem
# cp newcert.pem HINZ/hinz_cert.pem
```

Zertifikat verifizieren

```
<wrap em>CA.sh -verify</wrap>
```

Diese Kommando ist auf der Ebene des Elternverzeichnisses von demoCA auszuführen und verifiziert das Zertifikat in der Datei newcert.pem. Beispiel:

```
CA.sh -verify
newcert.pem: OK
```

—-

Anzeige, Verifikation, Hash-Links

- Schlüssel anzeigen:

```
# openssl rsa -text -noout -in newreq.pem
```

- Zertifikatsanforderung anzeigen:

```
# openssl req -text -noout -in newreq.pem
```

- Zertifikat anzeigen:

```
Für Zertifikat im PEM-Format:
# openssl x509 -text -noout -in newcert.pem

Für Zertifikat im DER-Format
# openssl x509 -text -noout -inform der -in rootcert.crt

Zertifikat selbst mit ausgeben (MIME-Block zwischen
-----BEGIN CERTIFICATE----- und -----END CERTIFICATE-----)
# openssl x509 -text -inform der -in rootcert.crt

Den Klartext weglassen, nur den MIME-Block ausgeben;
so kann man ein DER- in eine PEM-Zertifikat konvertieren
# openssl x509 -inform der -in rootcert.crt
```

- Zertifikat konvertieren:

```
Von PEM nach DER
# openssl x509 -inform pem -in ca-g3.pem -outform der -out ca-g3.crt

Von DER nach PEM
# openssl x509 -inform der -in ca-g3.crt -outform pem -out ca-g3.pem
```

- Fingerprint des Zertifikats berechnen:

```
openssl x509 -fingerprint -noout -in newcert.pem
```

- Verifikation des Zertifikats:

```
openssl verify -CAfile demoCA/cacert.pem newcert.pem

* Hash-Wert eines Zertifikats bestimmen und Link legen:<code>
```

ln -s newcert.pem $(openssl x509 -noout -hash -in newcert.pem).0 </code> Über einen solchen Link greifen die X.509-Verifikationsroutinen auf die Zertifikatsdateien zu. So nutzt z.B. der Apache mit mod_ssl im Rahmen der Überprüfung der zu einem Klienten-Zertifikat gehörenden Zertifikatskette solche Links für den Zugriff auf die einzelnen Zertifikatsdateien, die jeweils ein Zertifikat für den öffentlichen Schlüssel einer CA enthalten. Der Hashwert wird von der Komponente Subject eines Zertifikats gebildet. Unterschiedliche Zertifikate mit demselben Subject haben folglich denselben Hashwert.

Anmerkung: Das Skript c_rehash kann für die automatische Aktualisierung der Hash-Links aller Zertifikate in bestimmten Verzeichnissen genutzt werden. Beispiel:

```
c_rehash .
```

Dadurch werden für alle Dateien *.pem im aktuellen Verzeichnis neue Hash-Links erstellt.

Schlüssel und Zertifikat für SSL-Server (z.B. Apache) erstellen

Generierung eines RSA-Schlüssels ohne Passwortschutz (für Server-Betrieb sinnvoll):

```
openssl genrsa -out server.key 1024
```

nachträgliche Verschlüsselung des RSA-Schlüssels (mit Passwort), um ihn sicher aufbewahren zu können:

```
openssl rsa -in server.key -out server.key.pem -aes256
```

Entschlüsselung eines verschlüsselten RSA-Schlüssels:

```
openssl rsa -in server.key.pem -out server.key
```

selbstzertifiziertes Zertifikat für diesen privaten Schlüssel erstellen (anstelle einer neuen Zertifikatsanforderung an die CA):

```
openssl req -new -days 365 -x509 -out server_self.crt -key server.key
```

neue Zertifikatsanforderung an die CA erstellen:

```
openssl req -new -out server.req -key server.key
```

Zertifikatsanforderung durch CA signieren und so das Zertifikat erstellen:

```
openssl ca -policy policy_anything -days 500 -out server.crt -infiles
server.req
```

Durch die Option -days kann man die Gültigkeitsdauer des Zertifikats in Tagen festlegen. Fehlt eine explizite Angabe, wird die Voreinstellung aus der Konfigurationsdatei openssl.cnf entnommen. Ein typischer Wert ist 365. Unter Beachtung der Dateinamenskonventionen kann man das Zertifikat auch durch CA.sh -sign erstellen. Als Gültigkeitsdauer sind hierbei 365 Tage eingestellt. Diese Einstellung kann durch Modifikation des Skripts CA.sh geändert werden.

Die Policy legt fest, für wen (d.h. für welche DN) Zertifikate ausgestellt werden können. Die Policy-Definitionen stehen in der Konfigurationsdatei openssl.cnf.

Schlüssel und Zertifikat für den Apache-Server installieren

privater Schlüssel (Direktive SSLCertificateKeyFile):

```
cp server.key /etc/httpd/conf/ssl.key/server.key
```

Zertifikat (Direktive SSLCertificateFile):

```
cp server.crt /etc/httpd/conf/ssl.crt/server.crt
```

Zertifikat der CA (Direktive SSLCertificateChainFile):

```
cp demoCA/cacert.pem /etc/httpd/conf/ssl.crt/ca.crt
```

Statt Zertifikat und Zertifikat der CA könnte man auch das selbstzertifizierte Zertifikat verwenden:

```
cp server_self.crt /etc/httpd/conf/ssl.crt/server.crt
```

Zertifikat einer CA in den Browser laden

Die Liste der bekannten Zertifikate von CAs findet man bei Netscape 4.x unter Security –> Certificates –> Signers. Diese Liste kann erweitert werden, indem der Browser per HTTP ein Dokument mit dem MIME-Inhaltstyp

```
application/x-x509-ca-cert
```

lädt. Neben CA-Zertifikaten können auch Nutzer-, E-Mail- und Server-Zertifikate in den Browser geladen werden. Die entsprechenden Inhaltstypen lauten:

```
application/x-x509-server-cert
application/x-x509-user-cert
application/x-x509-email-cert
```

Das Laden von Zertifikaten kann z.B. über das Formular loadcert.html und das zugehörige CGI-Programm loadcert.pl erfolgen.

PKCS #12

Mit Hilfe dieses Formats kann der Netscape-Browser Zertifikate und private Schlüssel importieren und exportieren. Dies empfiehlt sich bei der (in der heutigen Praxis eher untypischen) Nutzung von Klienten-Zertifikaten für die Authentifizierung gegenüber einem WWW-Server. Diese Klienten-Zertifikate können mit OpenSSL generiert und konvertiert werden: Zertifikate und Schlüssel nach PKCS #12 exportieren:

```
openssl pkcs12 -export -in HINZ/hinz_cert.pem -inkey HINZ/hinz_key.pem -
certfile demoCA/cacert.pem -name "hinz" -out hinz.p12
```

Wenn der Browser Netscape 4.x die Datei hinz.p12 importiert, fügt er das in der Datei demoCA/cacert.pem enthaltene Zertifikat der Demo-CA zur Liste der akzeptierten CAs hinzu (Security –> Certificates –> Signers). Das Zertifikat des Nutzers Hinz ist unter Security –> Certificates –> Yours zu finden. Anzeige von Informationen, die im Format PKCS #12 vorliegen:

```
openssl pkcs12 -info -in hinz.p12
openssl pkcs12 -in hinz.p12
```

—-

Fortify

Dieser Abschnitt ist heute bei Verwendung moderner Browser irrelevant.

Damit kann die in den für den Export bestimmten Netscape-Browsern nutzbare Menge von Cipher Suites für SSL erweitert werden.

bei Apache/mod_ssl standardmäßig verfügbare Cipher Suites:

```
ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP
```

detaillierte Anzeige der bei OPENSSL verfügbaren Cipher Suites:

bei tcsh:

```
openssl ciphers -v 'ALL:\!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP'
```

bei bash/sh:

```
openssl ciphers -v 'ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP'
```

Seit der Lockerung der US-Exportbestimmungen dürfen auch Browser exportiert werden, die starke Kryptographie unterstützen. Daher wurde das Projekt Fortify mit Netscape 4.72 eingestellt.

Zugriff auf einen Apache-Server mittels Netscape und SSL

ohne Klienten-Authentifizierung:

Falls die CA des Server-Zertifikats bekannt ist, wird der Server vom Browser sofort akzeptiert.

Anderenfalls muss man in einem Dialog entscheiden, ob und wie lange das Zertifikat der Servers akzeptiert wird. Das Zertifikat kann für die aktuelle Sitzung oder bis zum Ende seiner Gültigkeit akzeptiert werden.

mit Klienten-Zertifikaten (Security –> Certificates –> Yours)

Beispiel für die Datei .htaccess, die den Zugriff auf Dokumente nur gestattet, wenn der Klient ein Zertifikat für den fiktiven Nutzer hinz vorweist:

```
SSLVerifyClient      require
SSLVerifyDepth       5
SSLRequireSSL
SSLRequire \
( %{SSL_CLIENT_S_DN } eq "/C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University
of Technology/OU=Department of Computer
Science/CN=hinz/Email=hinz@informatik.tu-chemnitz.de" )
```

mit Klienten-Zertifikaten und FakeBasicAuth bei Apache/mod_ssl

Beispiel für .htaccess:

```
SSLVerifyClient      require
SSLVerifyDepth       5
SSLRequireSSL
AuthName             SSLFake
AuthType             Basic
AuthUserFile         /usr/local/apache/conf/httpd.passwd
require              valid-user
```

Mit welchen Zertifikaten zugegriffen werden darf, wird in der durch AuthUserFile spezifizierten Passwort-Datei festgelegt. Als Nutzerkennzeichen ist dabei der Inhalt des Feldes Subject des Zertifikats anzugegeben. Sofern die UNIX-typische, auf dem DES basierende Passwort-Verschlüsselung genutzt wird, ist als Passwort immer die konstante Zeichenkette xxj31ZMTZzkVA einzutragen (das ist die Verschlüsselung des Wortes password). Beispiel für eine Passwort-Datei mit zwei Einträgen:

```
/C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University of Technology/OU=Department
```

```
of Computer Science/CN=hinz/Email=hinz@informatik.tu-
chemnitz.de:xxj31ZMTZzkVA
Rum:4YQK85NGp4Jj2
```

Der erste Eintrag enthält das Subject des Zertifikats von Nutzer hinz sowie die Verschlüsselung des konstanten Passwortes password. Der zweite Eintrag besteht aus dem Nutzernamen Rum sowie dessen verschlüsseltem Passwort (hier konkret topf). cgi-bin/printenv mit Option ExportCertData

Zugriff mit s_client oder SSL-Telnet und manuellem HTTP

Beispiele:

mit bekannter CA:

```
openssl s_client -quiet -connect kirke:443 -CAfile demoCA/cacert.pem
depth=1 /C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University of
Technology/OU=Department of Computer
Science/CN=kirke_ca/Email=hot@informatik.tu-chemnitz.de
verify return:1
depth=0 /C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University of
Technology/OU=Department of Computer Science/CN=kirke.informatik.tu-
chemnitz.de/Email=hot@informatik.tu-chemnitz.de
verify return:1
GET /
```

mit unbekannter CA, wobei der Server nur sein eigenes Zertifikat sendet:

```
openssl s_client -quiet -connect kirke:443
depth=0 /C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University of
Technology/OU=Department of Computer Science/CN=kirke.informatik.tu-
chemnitz.de/Email=hot@informatik.tu-chemnitz.de
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University of
Technology/OU=Department of Computer Science/CN=kirke.informatik.tu-
chemnitz.de/Email=hot@informatik.tu-chemnitz.de
verify error:num=27:certificate not trusted
verify return:1
depth=0 /C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University of
Technology/OU=Department of Computer Science/CN=kirke.informatik.tu-
chemnitz.de/Email=hot@informatik.tu-chemnitz.de
verify error:num=21:unable to verify the first certificate
verify return:1
GET /
```

mit unbekannter CA, wobei der Server eine Zertifikatskette sendet, die neben seinem eigenen Zertifikat auch noch das selbstzertifizierte Zertifikat seiner CA enthält:

```
openssl s_client -quiet -connect kirke:443
depth=1 /C=DE/ST=Saxony/L=Chemnitz/O=Chemnitz University of
Technology/OU=Department of Computer
```

```
Science/CN=kirke_ca/Email=hot@informatik.tu-chemnitz.de
verify error:num=19:self signed certificate in certificate chain
verify return:0
```

—-

SSL-Telnet

Der Server ließ sich in den Tests nur im Debug-Modus sinnvoll betreiben. Im Normalmodus, wenn er vom inetd gestartet wird, wird die Verbindung sehr früh beendet. Die Ursache ist unklar. Der Telnet-Server entnimmt seinen privaten Schlüssel sowie sein Zertifikat standardmäßig der Datei /usr/local/openssl/certs/telnetd.pem. Hierfür ist z.B. die weiter unten bei Stunnel genutzte Datei kirke.stunnel verwendbar:

```
cp KIRKE/kirke.stunnel /usr/local/openssl/certs/telnetd.pem
```

Start des Servers:

telnetd -debug -z ssl Es werden nur SSL-Verbindungen akzeptiert. telnetd -debug -z secure Telnet-Klienten, die kein SSL verwenden, werden mit dem Hinweis

```
telnetd: [SSL required - connection rejected].
```

abgewiesen. telnetd -debug -z secure -z key=KIRKE/kirke_key -z cert=KIRKE/kirke_cert Schlüssel und Zertifikat des Servers werden explizit spezifiziert. Nutzung des Klienten ohne Zertifikat (anonymes SSL bei Telnet, HTTP …):

```
telnet -z ssl kirke
telnet -z ssl kirke 443
```

Nutzung des Klienten mit Zertifikat:

```
telnet -z ssl -z cert=hinz_cert.pem -z key=hinz_req.pem kirke 443
```

---

s_client und s_server

Hierbei handelt es sich um einen SSL-Klienten sowie einen SSL-Server. Beide sind Bestandteil von openssl und eignen sich für Tests. Im folgenden werden einige Benutzungsbeispiele gezeigt. Klient:

```
openssl s_client -connect kirke:443
openssl s_client -cipher DES-CBC-SHA -connect kirke:443
openssl s_client -connect kirke:443 -key hinz_req.pem -cert hinz_cert.pem
```

HTTP-Anweisungen:

```
GET /test/SSLrequire/1.html
GET /sslcgi/printenv
```

Server:

einfacher Test-Server für interaktive Arbeit:

```
openssl s_server -key kirke_key -cert kirke_cert
openssl s_server
```

Hier hört der Server am Default-Port 4433. Standardmäßig werden privater Schlüssel und Zertifikat aus der Datei server.pem gelesen. Hierfür ist die weiter unten bei Stunnel genutzte Datei kirke.stunnel verwendbar.

```
openssl s_server -accept 8000 -key kirke_key -cert kirke_cert
```

Hier hört der Server auf Port 8000. Der Klient kann z.B. so darauf zugreifen:

```
openssl s_client -connect kirke:8000
```

einfacher WWW-Server mit Status-Seite (unter URL https://localhost:8000):

```
openssl s_server -accept 8000 -www -key kirke_key -cert kirke_cert
```

einfacher WWW-Server, der einen Datei-Zugriff gestattet, z.B. unter dem URL https://localhost:8000/test_seite:

```
openssl s_server -accept 8000 -WWW -key kirke_key -cert kirke_cert
```

das aktuelle Verzeichnis von openssl s_server wird als Server-Root betrachtet

der HTTP-Zugriff erfolgt durch

```
GET /test_seite HTTP/1.0
```

wobei

```
GET /test_seite H
```

auch schon genügt

---

privaten Key durch Netscape-Browser erstellen und per WWW zertifizieren lassen

Die HTML-Seite ca.html aus dem Paket Stunnel initiiert über das Tag

```
<KEYGEN NAME="SPKAC">
```

die Schlüssel-Generierung und sorgt für den Versand der Zertifikatsanforderung im SPKAC-Format (signed public key and challenge) an den WWW-Server. Das CGI-Programm ca.pl realisiert die Zertifizierung durch

```
openssl ca -batch ...
```

und sendet das Ergebnis als

```
Content-type: application/x-x509-user-cert
```

zurück.

Stunnel

Dieses Werkzeug stellt universelle SSL-Tunnel bereit. Stunnel Version 3.x:

Sofern stunnel als SSL-Server fungiert, benötigt das Programm eine Text-Datei, die sowohl den privaten Schlüssel als auch das Zertifikat enthält. Beide Informationen sind im PEM-Format bereitzustellen und werden durch folgende Trenner begrenzt:

1. —-BEGIN RSA PRIVATE KEY—–
2. —-END RSA PRIVATE KEY—–

bzw.

1. —-BEGIN CERTIFICATE—–
2. —-END CERTIFICATE—–

Erzeugung einer solchen Datei:

```
cat kirke_key kirke_cert > kirke.stunnel
```

Anwendungsbeispiele für den SSL-Tunnel: Schutz eines eigenen Echo-Servers echo.c (dessen Aufruf erfolgt wie bei inetd):

```
stunnel -d 9000 -p KIRKE/kirke.stunnel -f -l ./echo
```

Zugriff auf diesen Server mittels s_client:

```
openssl s_client -quiet -connect kirke:9000 -CAfile demoCA/cacert.pem
openssl s_client -quiet -connect kirke:9000
```

verschlüsselten Verkehr durch Sniffer beobachten, z.B. mittels ethereal:

Capture Filter:

tcp port 9000 Anzeige:

Tools –> Follow TCP Stream Echo-Server als Daten-Generator arbeiten lassen:

```
stunnel -d 9000 -p KIRKE/kirke.stunnel -f -l ./echo echo 1
```

Nutzung des Klienten-Modus von Stunnel:

```
stunnel -f -c -d 5000 -r 443
```

Stunnel hört hier auf Port 5000 und leitet jede Verbindungsanforderung über SSL zum Port 443 weiter. Beispiel-Klient:

```
telnet kirke 5000
```

Durch normales Telnet zum Port 5000 erreicht man den SSL-fähigen WWW-Server am Port 443 über SSL. Ein weiteres Nutzungsbeispiel ist der Schutz der IMAP-Verbindung des Mail-Agenten Pine, wie dies z.B. im Shell-Skript secure_imap gezeigt wird. Die Ermittlung eines freien TCP-Ports erfolgt hierbei über das Programm free_tcp_port.c.

Stunnel Version 4.x:

Der Aufruf der Software hat sich deutlich geändert. Typischerweise sieht er so aus:

```
stunnel config_file
```

Über die als Argument benannte Konfigurationsdatei steuert man die Arbeitsweise des Tunnels. Hier ein erstes Beispiel für eine Konfigurationsdatei:

```
cert = /usr/local/stunnel/mycert
chroot = /var/run/stunnel
# PID is created inside chroot jail
pid = /pids/stunnel.pid
setuid = nobody
setgid = wheel
```

```
debug = 7
output = /var/run/stunnel/stunnel.log
```

```
client = yes
```

```
[https]
accept  = 80
connect = libero:443
TIMEOUTclose = 0
```

```
[https1]
accept  = 81
connect = opac:443
TIMEOUTclose = 0
```

Stunnel arbeitet hier im Klienten-Modus, d.h., der entfernte Server unterstützt SSL, der lokale Klient dagegen nicht.

Die Service-Definition [https] bewirkt, dass der Tunnel auf Port 80 hört und eingehende Verbindungswünsche an den Port 443 (https) von Rechner libero weiterleitet.

Die Service-Definition [https1] bewirkt, dass der Tunnel auf Port 81 hört und eingehende Verbindungswünsche an den Port 443 (https) von Rechner opac weiterleitet.

Unter Verwendung des Tunnels kann man mit einem beliebigen Klienten, der kein SSL beherrscht, auf ein per SSL zugängliches Angebot zugreifen, z.B. so:

```
telnet localhost 80
GET /cgi-bin/nph-mgwcgi?LANG=Deutsch&login='start'&VERSION=2
```

Jetzt noch ein anderes Beispiel für eine Konfigurationsdatei. Hier arbeitet Stunnel im Server-Modus, d.h., der Klient unterstützt SSL, der Server dagegen nicht (die SSL-Fähigkeit des Servers wird durch Stunnel hergestellt):

```
cert = /usr/local/stunnel/mycert
chroot = /var/run/stunnel
# PID is created inside chroot jail
pid = /pids/stunnel.pid
setuid = nobody
setgid = wheel
```

```
debug = 7
output = /var/run/stunnel/stunnel.log
```

```
[ssh]
accept  = 4000
connect = 22
```

Stunnel hört in diesem konstruierten, für die Praxis wohl eher nicht sinnvollen Beispiel an Port 4000 und leitet eingehende Verbindungen an den Port 22 von localhost weiter. Man kann also mit einem SSL-Klienten Kontakt zum SSH-Dämon aufnehmen und sich den SSH-Versions-String anzeigen lassen:

```
openssl s_client -quiet -connect localhost:4000
```

Holger Trapp

letzte Modifikation: 29.12.2014

Quelle: https://www-user.tu-chemnitz.de/~hot/SSL/

Last update: **2017/12/04 12:53**