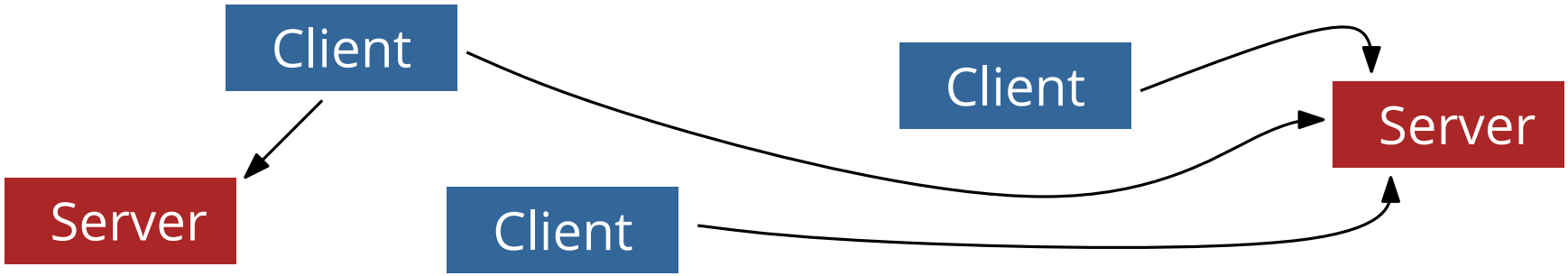
Protokollsicherheit

Kerberos, TLS

Kerberos

Authentifizierungsdienst für Client/Server-Anwendungen.

- Client und Server kommunizieren über unsicheres Netzwerk.
- Verteilte Server bieten verschiedene Dienste an (z.B. Drucker, Datenbank, . . .)
- Ziel: gegenseitige Authentifizierung von Client und Server



Anwendungen:

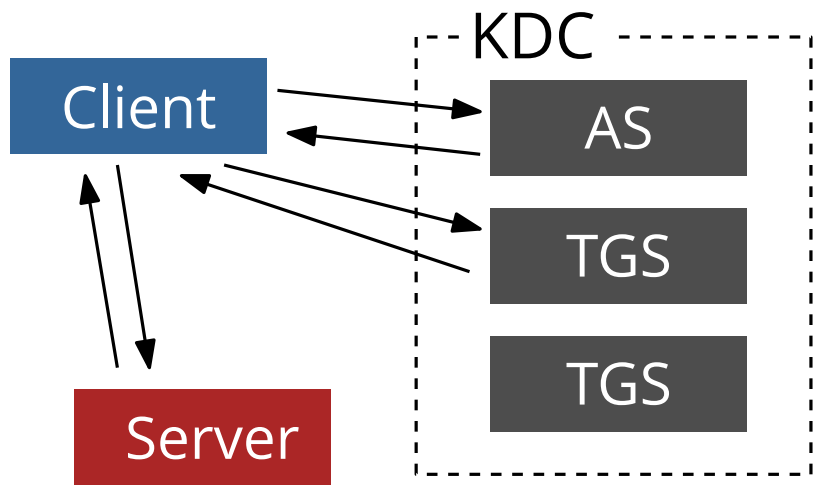
- Benutzerauthentifizierung, z.B. zum Zugiff auf Dateiserver in lokalen Netzwerken
- Kerberos ist das Standardprotokoll für die Authentifizierung in Windows-Netzwerken

Kerberos

Ziel: Client C möchte den Dienst von Server S benutzen, d.h. beide wollen einander gegenseitig authentifizieren.

Teilnehmer:

- Client (C)
- Server (S)
- Authentication Server (AS)
- Ticket Granting Server (TGS)



AS und (evtl. mehrere) TGS bilden ein Key Distribution Center (KDC).

Annahmen:

- Jeder Client teilt mit dem AS ein Geheimnis (z.B. Passwort).
- AS hat mit jedem TGT G einen Langzeitschlüssel $K_{AS,G}$
- Jeder TGT G hat mit jedem Server S einen Langzeitschlüssel $K_{G,S}$.

Kerberos

Kerberos basiert auf dem Needham-Schroeder Protokoll.

1. Der Client erfragt from Authentication Server ein Ticket für einen Ticket-Granting-Server (TGS). Er beweist seine Identität mittels des beiden bekannten Passworts.
2. Der Authentication Server stellt ein Ticket-Granting-Ticket (TGT) aus.
3. Mit dem TGT kann sich der Client vom Ticket Granting Server (TGS) für jeden Dienst, den er benutzen will, ein Ticket ausstellen lassen.
4. Die Client authentifiziert sich beim Server mittels des Tickets.

Das TGT hat eine bestimmte Lebensdauer (z.B. 8 Stunden). Es ist nützlich, da der Client somit sein Passwort nur einmal am Anfang der Sitzung eingeben muss.

Kerberos (vereinfacht)

1. $C \rightarrow AS : C, G, N_1$
2. $AS \rightarrow C : \{K_{C,G}, G, N_1\}_{K_{AS,C}}, T_{AS,C,G}$
3. $C \rightarrow G : \{C, timestamp\}_{K_{C,G}}, T_{AS,C,G}, N_2, S$
4. $G \rightarrow C : \{K_{C,S}, N_2, S\}_{K_{C,G}}, TG, C, S$
5. $C \rightarrow S : \{C, timestamp_C\}_{K_{C,S}}, TG, C, S$
6. $S \rightarrow C : \{timestamp_C + 1\}_{K_{C,S}}$

Ticket $T_{X,Y}$:

$T_{Z,X,Y} = Y, \{\text{Nonce}, \text{Adresse von } X, \text{Haltbarkeitsdatum}, K_{X,Y}\}_{K_{Z,Y}}$

Kerberos (vereinfacht)

Anfangsauthentisierung:

1. $C \to AS : C, G, N_1$
   - Der Authentication Server prüft in seiner Datenbank, dass C ein Passwort hat. Der Schlüssel $K_{AS,C}$ wird dort nachgeschlagen.
   - Der Sitzungschlüssel $K_{C,G}$ wird frisch erzeugt.
   - Das Ticket-Granting-Ticket $T_{AS,C,G} = \{N, \mathsf{addr}(C), t, K_{C,G}\}_{K_{AS,G}}$ wird erzeugt.
2. $AS \to C : \{K_{C,G}, N_1, G\}_{K_{AS,C}}, T_{AS,C,G}$

Kerberos (vereinfacht)

Ticket für einen Dienst S erzeugen:

3. $C \to G : \{C, t\}_{K_{C,G}}, T_{C,G}, N_2, S$
   - Der Ticket-Granting-Server G kann das TGT $T_{AS,C,G} = \{N, \mathsf{addr}(C), t, K_{C,G}\}_{K_{AS,G}}$ entschlüsseln.
   - G prüft, dass das TGT noch nicht abgelaufen ist.
   - G entschlüsselt $\{C, t\}_{K_{C,G}}$ und prüft in der Datenbank, dass C den Dienst S benutzen darf.
   - G erzeugt das Ticket $T_{G,C,S} = \{N', \mathsf{addr}(C), t', K_{C,S}\}_{K_{G,S}}$
4. $G \to C : \{K_{C,S}, N_2, S\}_{K_{C,G}}, T_{G,C,S}$

Kerberos (vereinfacht)

Mit dem Ticket beim Server authentifizieren:

5. $C \to S : \{C, t_C\}_{K_{C,S}}, T_{G,C,S}$
   - Der Server S kann das Ticket $T_{G,C,S} = \{N, \mathsf{addr}(C), t, K_{C,S}\}_{K_{G,S}}$ entschlüsseln.
   - S entschlüsselt $\{C, t_C\}_{K_{C,S}}$
6. $S \to C : \{t_C + 1\}_{K_{C,S}}$

Kerberos — PKINIT

Kerberos-Erweiterung mit asymmetrischer Verschlüsselung, in der Clients nicht unbedingt ein Geheimnis mit dem AS teilen müssen.

Beispiel: Microsoft Active Directory

Anfangsauthentisierung (PKINIT-26):

1. $C \to AS : Cert_C, \{t_C, N_1'\}_{sk_C}, C, G, N_1$
   - Der Authentication Server erzeugt einen neuen symmetrischen Schlüssel k.
   - Der Authentication Server holt den öffentlichen Schlüssen pk_C von C und prüft mit diesem das Zertifikat von $Cert_C$.
2. $AS \to C : \{K_{C,G}, N_1, G\}_k, \{Cert_{AS}, \{k, N_1'\}_{sk_{AS}}\}_{pk_C}, T_{AS,C,G}$

[Cervesato, Jaggard, Scedrov, Tsay, Walstad, 2008]

Kerberos — PKINIT

Mögliche Korrektur:

1. $C \rightarrow AS : Cert_C, \{t_C, N_1'\}_{sk_C}, C, G, N_1$
2. $AS \rightarrow C : \{K_{C,G}, N_1, G\}_k, \{Cert_{AS}, \{C, k, N_1'\}_{sk_{AS}}\}_{pk_C}, T_{C,G}$

PKINIT-27:

1. $C \rightarrow AS : Cert_C, \{t_C, N_1'\}_{sk_C}, C, G, N_1$
2. $AS \rightarrow C : \{K_{C,G}, N_1, G\}_k, \{Cert_{AS}, \{k, checksum\}_{sk_{AS}}\}_{pk_C}, T_{C,G}$

SSL (Secure Sockets Layer)
TLS (Transport Layer Security)

Ziele:

- Sichere Kommunikation zwischen einem Client und einem Server.
- Authentisierung des Servers (optional auch des Clients) bzgl. einer Public Key Infrastruktur.

Ursprung:

- SSL 1.0: Netscape (intern, ca. 1994)
- SSL 2.0: Netscape, November 1994, mehrere Schwächen
- SSL 3.0: Netscape, 1996
- TLS 1.0 (interne Verionsnummer: SSL 3.1): 1999
- TLS 1.1: 2006
- TLS 1.2: 2008

SSL (Secure Sockets Layer)
TLS (Transport Layer Security)

Die Kommunikation wird auf der Transportebene abgesichert.

Anwendungsprotokolle (HTTP, SMTP, IMAP, . . .) können TLS transparent verwenden.

Identifikation von TLS-Verbindungen:

- Spezielle Port-Nummer, z.B. 443 für HTTP über TLS.
- Spezielles Kommando STARTTLS zum Einleitung der Verschlüssellung über TLS, z.B. SMTP, IMAP.

TLS (Transport Layer Security)

Handshake Protocol:

- Vereinbarung von Protokollparametern (z.B. Verschlüsselungsalgorithmus)
- Erzeugung und Austausch eines Sitzungsschlüssels
- Authentifizierung von Server und evtl. Client

Record Protocol

- Protokoll zur Kodierung, Komprimierung, Authentisierung und Verschlüsselung des eigentlichen Datenstroms mit dem Sitzungsschlüssel.

TLS — Handshake Protocol

- Client und Server tauschen die Parameter der Verbindung im Klartext mittels Hello-Nachrichten aus.
- Client und Server etablieren ein gemeinsames Geheimnis (Pre-Master-Secret).
- Beide berechnen ein Master-Secret, aus dem ein Sitzungsschlüssel generiert wird.
- Ist einmal ein Master-Secret ausgetauscht, so kann die Verbindung später ohne neuen Handshake wiederaufgenommen werden.

TLS — Handshake Protocol

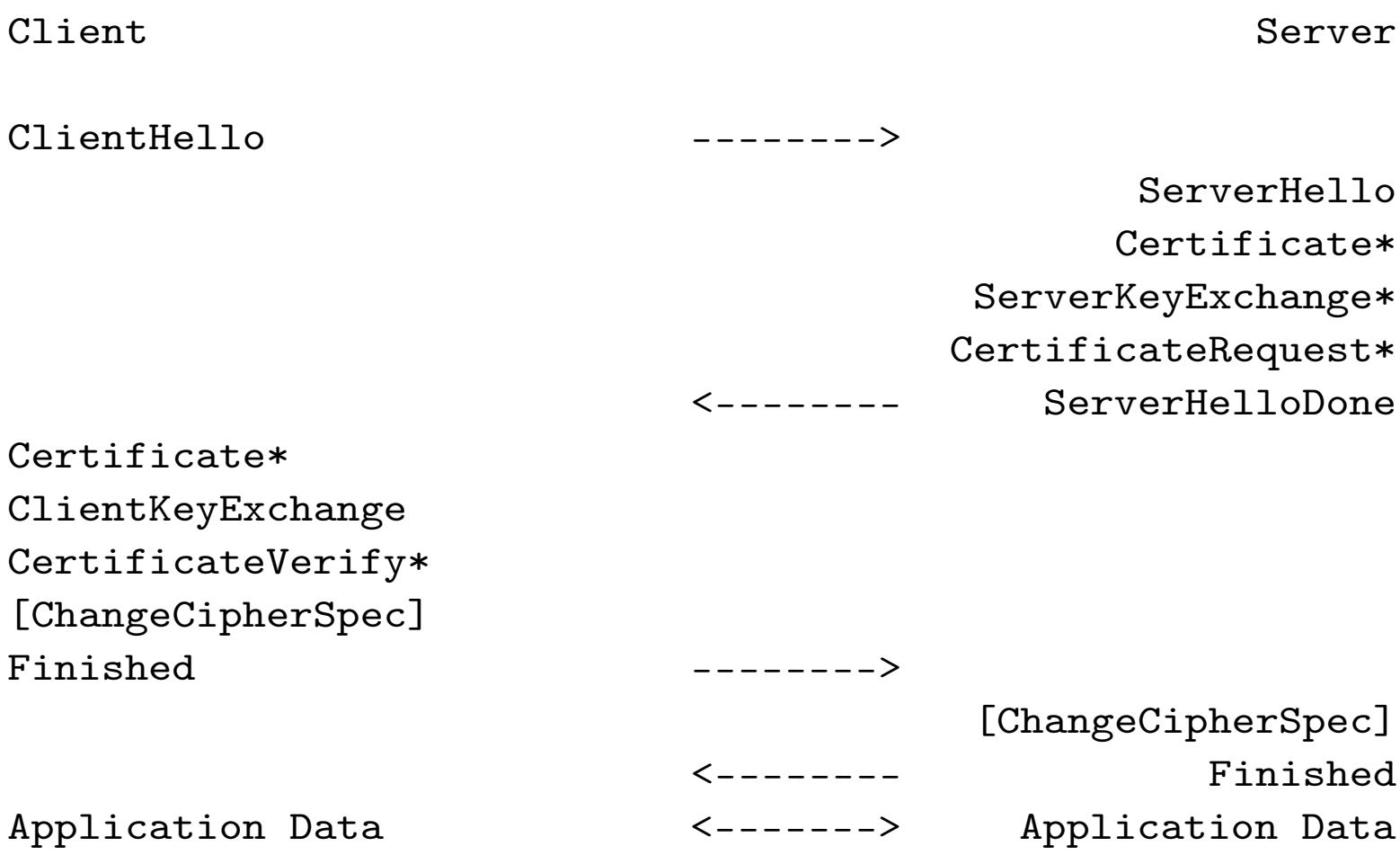


Figure 1. Message flow for a full handshake

TLS — Handshake Protocol (vereinfacht)

1. `ClientHello`: $C \to S : N_C$, Protokollparameter
2. `ServerHello`: $S \to C : N_S$, Protokollparameter, Serverzertifikat
3. `ClientKeyExchange`: $C \to S : \{N'_C\}_{pk_S}$
   - N'_C ist das frisch erzeugte Pre-Master-Secret.
   - Das Master-Secret ms wird von Client und Server jeweils aus N_C, N_S und N'_C berechnet.
   - Der Sitzungsschlüssel ist nun ms.
4. `ChangeCipherSpec`: $C \to S : *$
5. $S \to C : \{\text{Finished}\}_{ms}$
6. `ChangeCipherSpec`: $S \to C : *$
7. $C \to S : \{\text{Finished}\}_{ms}$
8. Applikationsdaten (mit ms verschlüsselt)

TLS — Tatsächlicher Inhalt der Nachrichten

`ClientHello`:

- Maximal unterstütze Protokollversion, z.B. 3.1 für TLS 1.0
- Zufallszahl, Sitzungs-Id
- Vorschläge für Verschlüsselungsalgorithmen, z.B:
  `TLS_RSA_WITH_AES_256_CBC_SHA`,
  `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
- Vorschläge für Komprimierungsalgorithmus

`ServerHello`:

- Protokollversion
- Zufallszahl, Sitzungs-Id
- Verschlüsselungsalgorithmus
- Komprimierungsalgorithmus

usw.

TLS — Abbreviated Handshake

1. `ClientHello`: $C \to S : N_C$, Protokollparameter
2. `ServerHello`: $S \to C : N_S$, Protokollparameter, Serverzertifikat
   Wenn der Server die Sitzungs-Id bereits kennt, nimmt er an, dass die Sitzung wieder aufgenommen werden soll. Dann wird das bereits beiden bekannte Master-Secret weiterhin benutzt.
3. `ChangeCipherSpec`: $C \to S : *$
4. $S \to C : \{\text{Finished}\}_{ms}$
5. `ChangeCipherSpec`: $S \to C : *$
6. $C \to S : \{\text{Finished}\}_{ms}$
7. Applikationsdaten (mit ms verschlüsselt)

TLS — Renegotiation Attack

2009 wurde in TLS ein Man-in-the-Middle-Angriff gefunden, der für HTTPS auch praktisch ausnutzen lässt.

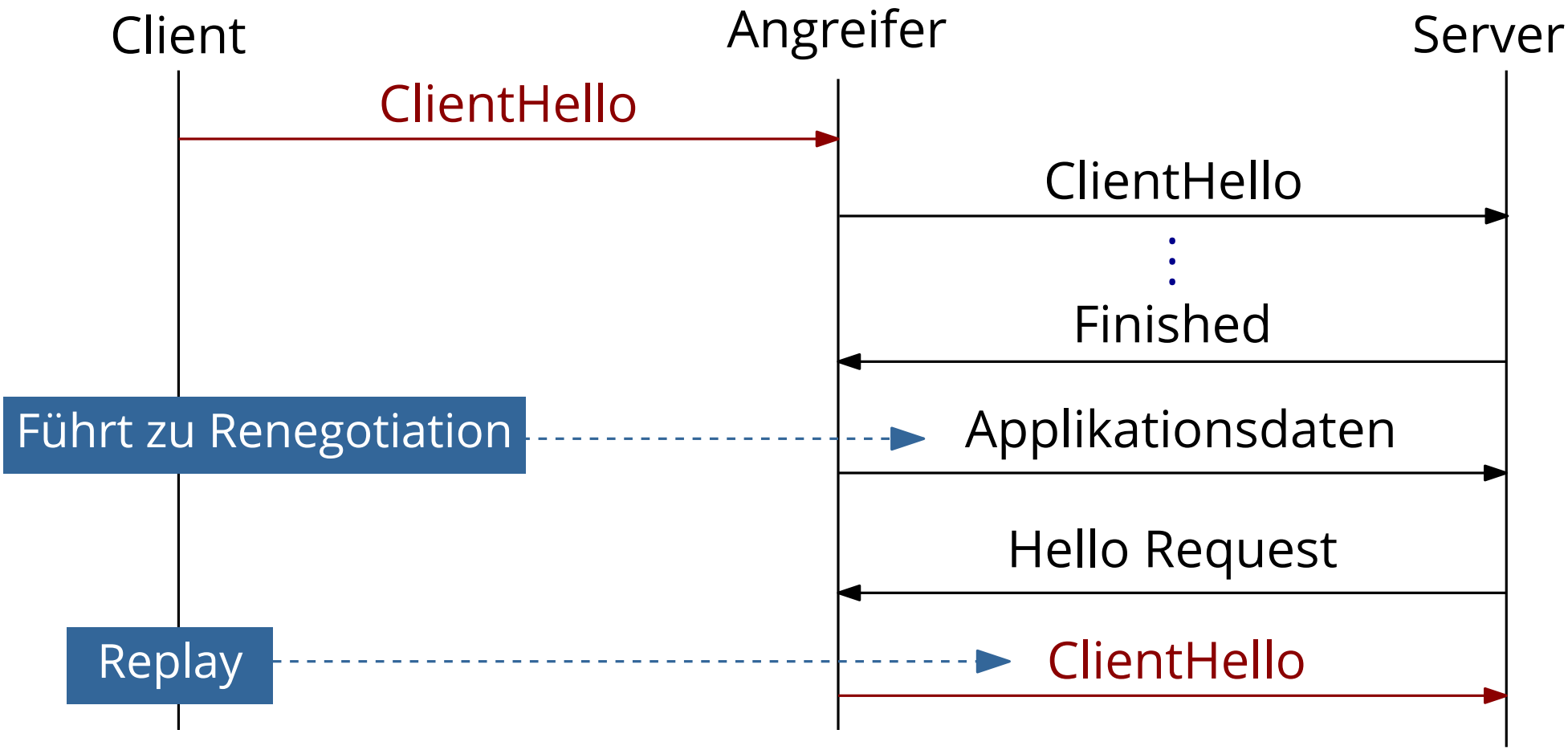
TLS erlaubt zu jedem Zeitpunkt ein erneutes Ausmachen der Protokollparameter.

In einigen Situationen hat der Server keine andere Wahl, als einen neuen Handshake einzuleiten. Beispiel:

- Ein HTTP-Server stellt sowohl Seiten bereit, die keine Authentisierung des Clients benötigen, als auch solche, die eine Authorisierung erfordern.
- Ein Client kann zunächst auf öffentliche Seiten zugreifen. Dafür wird der Handshake ohne Client-Authentisierung ausgeführt.
- Greift der Client dann auf eine geschützte Seite zu, so muss der Handshake mit Client-Authentisierung wiederholt werden.

TLS — Renegotiation Attack

Ein Angreifer kann die Renegotiation ausnutzen, um beliebige Daten am Anfang einer Sitzung einzufügen.



Angreifer leitet nun alle Nachrichten weiter.
Client führt den Handshake aus und sendet seine Daten.

TLS — Renegotiation Attack und HTTPS

Angreifer sendet folgende Applikationsdaten:

```
"GET /highsecurity/index.html HTTP/1.1
Host: example.com
Connection: keep-alive
GET /evil/do.php?evilStuff=here HTTP/1.1
Host: example.com
Connection: close
X-ignore-what-comes-next: "
```

Der Zugriff auf `/highsecurity/index.html` löst Renegotiation aus.
Der Client authentifiziert sich und sendet seine echten Daten:

```
"GET /index.html HTTP/1.1
Cookie: AuthMe=Now
..."
```

[M. Ray, S. Dispensa, 2009]

TLS — Renegotiation Attack und HTTPS

Insgesamt wird folgende HTTP-Sitzung ausgeführt.

```
GET /highsecurity/index.html HTTP/1.1
Host: example.com
Connection: keep-alive
GET /evil/do.php?evilStuff=here HTTP/1.1
Host: example.com
Connection: close
X-ignore-what-comes-next: GET /index.html HTTP/1.1
Cookie: AuthMe=Now
...
```

Im Zugriff auf `/evil/do.php` wird ein Authentifizierungs-Cookie des Clients benutzt.

[M. Ray, S. Dispensa, 2009]

Protokollanalyse

Ideal: Verifikation der tatsächlichen Protokollimplementierung

CERT® Advisory CA-2002-29:

> Multiple Kerberos distributions contain a remotely exploitable buffer overflow in the Kerberos administration daemon. A remote attacker could exploit this vulnerability to gain root privileges on a vulnerable system.

Die Verifikation von Protokollimplementierungen in C ist momentan in der Praxis noch zu aufwändig.

Protokollanalyse

Die Analyse von abstrahierten Protokollmodellen (z.B. mit Proverif) ist nur begrenzt praktikabel:

- Erstellung eines abstrakten Modells für den Code ist kompliziert und aufwändig.
- Ist das Modell korrekt?
- Das Model muss bei Änderungen in der Implementierung stets aktualisiert werden.

Lösungsansätze:

- Automatisierung der Erzeugung von Modellen von der Implementierung.
- Entwicklung von Methoden zur direkten Programmanalyse. Beispiel: Verstärkung des ML-Typsystems, so dass Protokolleigenschaften durch Typüberprüfung verifiziert werden können.

Protokollanalyse

Beispielansätze:

- Automatische Erzeugung von Proverif-Dateien aus einer F#-Implementierung eines Protokolls [Bhargavan, Fournet, Gordon, Tse]
- Direkte Verifikation von Eigenschaften einer F#-Protokollimplementierung mit einer Verfeinerung des F#-Typsystems:
  F7 [Bhargavan, Fournet, Gordon]

Protokollimplementierung in F#

Beispiel: Einfache Nachrichtenauthenthisierung [Bhargavan, Fournel, Gordon, Tse, 2006]

$$A \to B : \mathsf{HMAC}((password, text), N), \{N\}_{pk_B}, text$$

- Implementiere das Protokoll vollständig in F#
- Mit einer konkreten Kryptographiebibliothek erhält man so eine voll funktionsfähige Implementierung des Protokolls.
- Verifiziere Programmeigenschaften bezüglich symbolischer Kryptographie durch automatische Generierung von Proverif-Code aus dem F#-Quelltext.

Konkrete Kryptographie

```
module Crypto // concrete code in F#
open System.Security.Cryptography
type bytes = byte[]
type rsa key = RSA of RSAParameters
...
let rng = new RNGCryptoServiceProvider ()
let mkNonce () =
let x = Bytearray.make 16 in
rng.GetBytes x; x
...
let hmacsha1 k x =
new HMACSHA1(k).ComputeHash x
...
let rsa = new RSACryptoServiceProvider()
let rsa keygen () = ...
let rsa pub (RSA r) = ...
let rsa encrypt (RSA r) (v:bytes) = ...
```

Symbolische Kryptographie

```
module Crypto
type bytes =
| Name of Pi.name
| HmacSha1 of bytes * bytes
| RsaKey of rsa key
| RsaEncrypt of rsa key * bytes
...
and rsa key = PK of bytes | SK of bytes
...
let freshbytes label = Name (Pi.name label)
let mkNonce () = freshbytes "nonce"
...
let hmacsha1 k x = HmacSha1(k,x)
...
let rsa keygen () = SK (freshbytes "rsa")
let rsa pub (SK(s)) = PK(s)
let rsa encrypt s t = RsaEncrypt(s,t)
let rsa decrypt (SK(s)) e = match e with
| RsaEncrypt(pke,t) when pke = PK(s) -> t
| -> failwith "rsa_decrypt failed"
```

Programmausgabe

Mit konkreter Kryptographie:

```
Connecting to localhost:8081
Listening at 127.0.0.1:8081
Sending: {FADA2wu+78oU8iix1/hxk7YGfeuOsoA=... 154}
 containing 154 bytes
```

Mit symbolischer Kryptographie sieht die Nachricht folgendermaßen aus:

```
HMACSHA1{nonce3}[pwd1 | 'Hi'] |
RSAEncrypt{PK(rsa secret2)}[nonce3] | 'Hi'
```

Analyse mit Proverif

Übersetze das funktionale Programm in den pi Kalkül und analysiere diesen mit Proverif.

```
let client text =
  let c = Net.connect address in
  log tr (Send(text));
  Net.send c (marshall (make text pkB pwdA))
let server () =
  let c = Net.listen address in
  let m,en,text = unmarshall (Net.recv c) in
  verify (m,en,text) skB pwdA; log tr (Accept(text))}
```

Anfrage:

```
query ev:Ev(PwdmacAccept(text)) ==> ev:Ev(PwdmacSend(text)).
```

Analyse mit Proverif

Beispiel:

Die Funktion

```
let mac nonce pwd text =
Crypto.hmacsha1 nonce (concat (utf8 pwd) (utf8 text))
```

wird zum Prozess

```
!in(mac, (nonce,pwd,text,k));
out(k,Hmacsha1(nonce,Concat(Utf8(pwd),Utf8(text))))
```

HIerbei ist `k` der Name des Kanals, an den das Ergebnis gesendet werden soll. Dieser Parameter wird auch *Continuation* genannt.