

IT-Sicherheit

- Sicherheit vernetzter Systeme -

Kapitel 13: Netzsicherheit -
Schicht 7: Application Layer
Secure Shell (SSH)

Wer ist Berndi?

- 28.01.13: Polizei durchsucht Verlagshaus der Augsburger Allgemeinen (AA) und verlangt Klartextnahmen des Foren-Nutzers „Berndi“.
- Beleidigungsklage des Ordnungsreferenten der Stadt Augsburg Dr. Volker Ulrich
 - Forenbeitrag des Nuters Berndi zu einem Bericht über Ulrichs Vorschlag zum nächtlichen Verkaufsverbots von Alkohol an Tankstellen:
„Dieser Ullrich verbietet sogar erwachsenen Männern sein Feierabend-Bier ab 20.00 Uhr, indem er geltendes Recht beugt und Betreiber massiv bedroht.“
 - Vorwurf der Rechtsbeugung ist für den Ordnungsreferenten eine Beleidigung
- AA händigt Klartextnahmen aus
- AA hatte Forenbeitrag bereits gelöscht
- Massive Diskussion
 - Gefahr für die Pressefreiheit
 - Vorgehen der Staatsanwaltschaft rechtmäßig und verhältnismäßig?

Inhalt

- SSH: Historie
- SSH v1 vs. SSH v2
- SSH-Protokollarchitektur
 - Transport Protocol
 - Authentication Protocol und Authentisierungsarten
 - None
 - Host based
 - Password
 - Public key
 - Connection Protocol
 - Interactive session
 - X11-Forwarding
 - Port-Forwarding

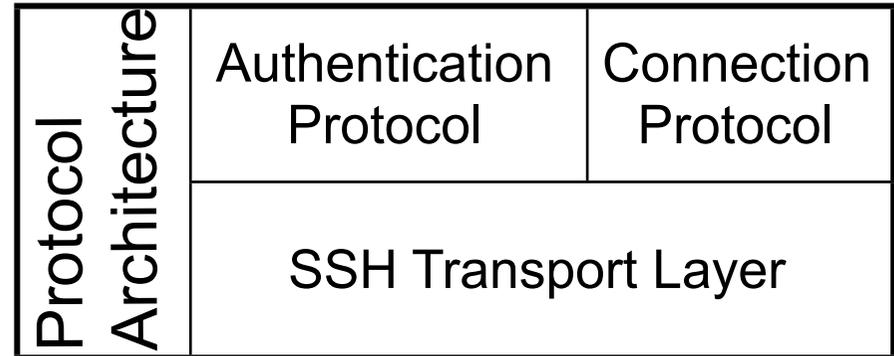
Historie

- Entwickelt von Tatu Ylönen (TU Helsinki, Finland); 1995;
- Reaktion auf Passwort-Sniffing-Angriff an der TU Helsinki
- Ersatz für unsichere Unix-Tools: telnet, rlogin, rsh, rexec (ftp); (Übertragen Passworte im Klartext)
- Erste Version SSH-1 und Implementierung 1995, frei verfügbar
- Schnelle Verbreitung
- Kommerzielle Variante über SSH Communication Security; Gegründet Dez. 1995 von Ylönen
- Weiterhin freie Version (OpenSSH) verfügbar
- 1996 verbesserte Version SSH v2; inkompatibel zu SSH v1
- 2006 SSH v2 über IETF standardisiert [RFC 4251, RFC 4252, RFC 4253, RFC 4254, RFC 4255, RFC 4256]

SSH v1 vs. SSH v2

- Verbesserungen in SSH v2:
 - Diffie-Hellman Schlüsselaustausch
 - Verbesserte Integrität: Message Authentication Code (MAC) statt CRC32
 - Mehrere Shell-Sessions über eine SSH-Verbindung
 - 2001: Mehrere Schwachstellen in SSH-1 gefunden:
 - RC4-verschlüsselte Passwörter können mitgelesen werden, falls beim Server leere Passwörter verboten sind
 - Replay-Angriff möglich
 - CRC32 erlaubt gezielte Modifikation der Nachrichten
 - CRC32 Attack detection code contains remote integer overflow; Angreifer kann Code mit Rechten des ssh-Damon (i.d.R. root) ausführen
- ➔ SSH v1 ist unsicher, NICHT mehr verwenden
- ➔ Modus-Fallback auf SSH v1 server-seitig deaktivieren
- ➔ Im Folgenden nur SSH v2

SSH Architektur



■ Geschichtete Architektur:

■ SSH Protocol Architecture [RFC 4251]

■ SSH Authentication Protocol [RFC 4252]

- Authentisierung des Clients gegenüber dem Server („client-driven“)

■ SSH Transport Layer Protocol [RFC 4253]

- Authentisierung des Servers gegenüber dem Client
- Vertraulichkeit, ggf. mit Kompression
- Integrität
- Perfect Forward Secrecy, Re-Keying (oft nach 1 GB oder 1 h)

■ SSH Connection Protocol [RFC 4254]

- Multiplexing (mehrere logische Kanäle über eine SSH-Verbindung)

SSH Protocol Architecture

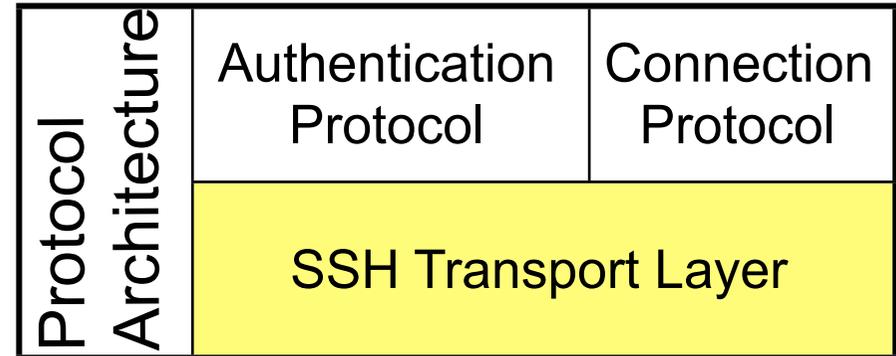
- Client (ssh) verbindet sich mit Host (Server, sshd)
- Jeder Host besitzt Schlüsselpaar
 - Zur Authentisierung gegenüber den Clients
 - Zwei Trust-Modelle:
 - Lokale Datenbasis mit Hostname - Public Key Paaren
 - Hostname und Public Key werden von CA zertifiziert, Client benötigt nur öffentlichen Schlüssel der CA
 - Client kann öffentlichen Schlüssel bei der ersten Verbindung zum Host in lokale Datenbasis übernehmen
 - ABER: Gefahr eines Man-in-the-Middle Angriffs
 - Sollte in der Praxis nicht verwendet werden, but „there is no widely deployed key infrastructure available on the Internet“
- Designprinzip: Erweiterbarkeit
 - Basisprotokoll so einfach wie möglich
 - Mindestsatz an zu unterstützenden Algorithmen (wg. Interoperabilität)
 - Erweiterungen v. Algorithmen, Formaten u. Protokoll möglich

Protocol Architecture: Sicherheit

- Algorithmen für jede Kommunikationsrichtung frei wählbar:
 - Verschlüsselung
 - Integritätssicherung
 - Schlüsselaustausch
 - Datenkompression
- Multiple Authentisierungsverfahren für jeden Client
- Server darf keine (eigene) Verbindung zum Client aufbauen
 - Ausnahme: Client fordert Forwarding bestimmter Dienste an
- Server darf keine Kommandos auf Client ausführen
- Perfect Forward Secrecy
 - Kompromittierung eines Session Key oder eines privaten Schlüssels darf nicht zur Kompromittierung einer früheren Session führen

SSH Architektur

■ Geschichtete Architektur:



■ SSH Authentication Protocol [RFC 4252]

- Authentisierung des Clients gegenüber dem Server

■ SSH Transport Layer Protocol [RFC 4253]

- Authentisierung des Servers gegenüber dem Client
- Vertraulichkeit
- Integrität
- Perfect Forward Secrecy

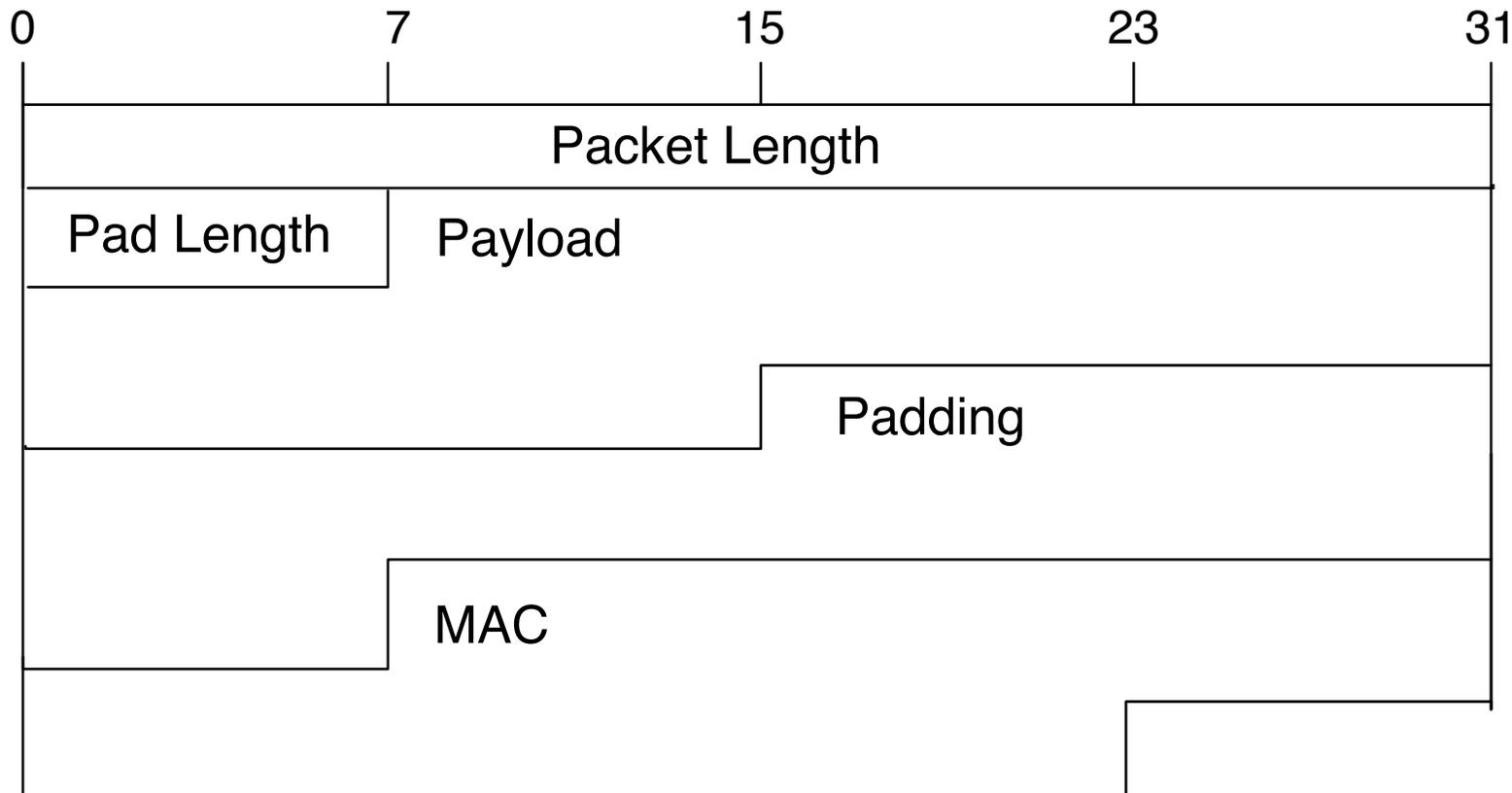
■ SSH Connection Protocol [RFC 4254]

- Multiplexing

SSH Transport Protocol

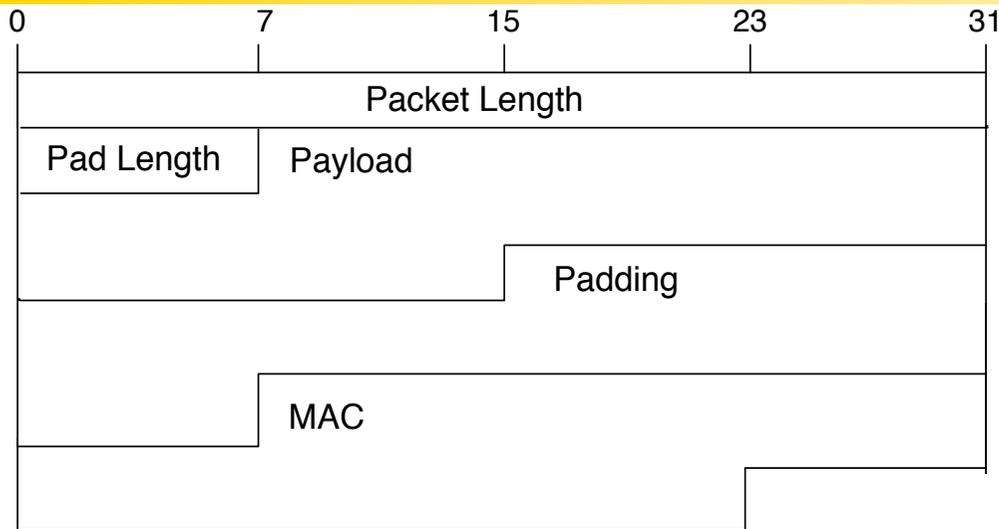
- Setzt auf verlässliches Schicht 4 Protokoll auf (i.d.R. TCP)
- Sicherheitsdienste:
 - Verschlüsselung der Nutzdaten
 - Integritätssicherung
 - Server-Authentifizierung (es wird (nur) der Host authentisiert)
 - Kompression der Nutzdaten
 - Aushandlung von Algorithmen
- Unterstützte Verschlüsselungsalgorithmen
 - 3DES, Blowfish, Twofish, AES, IDEA, CAST im CBC Modus
 - Arcfour (entspricht RC4, „has problems with weak keys“)
 - none („not recommended“)
- Hash-Algorithmen
 - HMAC-SHA1, HMAC-MD5, HMAC-SHA1-96, HMAC-MD5-96
 - HMAC-RIPEMD-160
- Asymmetrische Verfahren: RSA und DSA

Transport Protocol Packet Format



- Gesamtlänge von Packet Length inkl. Padding muss ein Vielfaches der Blocklänge oder ein Vielfaches von 8 ergeben
- Maximale Blocklänge: < 35.000 Byte

Transport Protocol Packet Format (Forts.)



- $\text{Packet Length} = \text{Pad Length} + \text{Payload} + \text{Padding}$
- **Payload:** Daten; falls Kompression aktiviert, wird dieses Feld mit zlib (deflate-Algorithmus, vgl. .gz-Dateien) komprimiert
- **MAC über gesamtes Paket;** berechnet vor Verschlüsselung
 - $\text{MAC} = \text{HMAC}(\text{gemeinsamer Schlüssel}, \text{SeqNr.} \mid \text{unencrypted Packet})$
- Falls Verschlüsselung aktiviert, wird alles bis auf MAC verschlüsselt

Aushandlung der Algorithmen

- Schlüsselaushandlung erfordert explizite Authentisierung des Server-Hosts
 - Jeder Host besitzt mindestens 1 Schlüsselpaar in /etc/ssh
ssh_host_dsa und ssh_host_dsa.pub oder ssh_host_rsa u. .._rsa.pub
- Jede Seite kann `kexinit` Nachricht schicken; enthält:
 - Cookie: Zufallszahl
 - jeweils Komma-separierte Liste mit absteigender Priorität für:
 - `kex_algorithm`: Schlüsselaustausch
 - `server_host_key_algorithm`: Server listet Algorithmen, für die er Schlüsselpaare besitzt
 - `encryption_client_to_server` und `server_to_client`
 - `mac_algorithm_client_to_server` und `server_to_client`
 - `compression_client_to_server` und `server_to_client`
 - `language_client_to_server` und `server_to_client`: verwendete Sprache (Optional)
 - `first_kex_packet_follows`

Aushandlung der Algorithmen; Schlüsselaustausch

- Server analysiert die Client-Listen; gewählt wird erster Algorithmus, der auch vom Server unterstützt wird (!)
- Nach Abschluss der Aushandlung beginnt Schlüsselaustausch
- Optimierung: Jede Seite kann präferierten Algorithmus „raten“ und sofort das entsprechende key-exchange-Paket anhängen
 - richtig geraten: Folgepaket wird als key-exchange-Paket akzeptiert
 - falsch geraten: Paket wird verworfen

Schlüsselaustausch

- Schlüsselaustausch über Diffie-Hellmann mit SHA-1
 - Kombiniert mit digitaler Signatur durch Host -> Authentisierung
 - zwei definierte DH-Gruppen
- Ergebnis des Schlüsselaustausches: Secret K und Hash H
 1. Client wählt x und sendet $e = g^x \text{ mod } p$
 2. Server wählt y und berechnet $f = g^y \text{ mod } p$
 - 2.1. $K = e^y \text{ mod } p$
 - 2.2. $H = \text{Hash}(V_c | V_s | I_c | I_s | K_s | e | f | K)$ mit
 $V = \text{Identität von C und S}$
 $I = \text{kexinit Nachrichten}$
 $K_s = \text{Public Host Key von S}$
 - 2.3. $s = \{H\}$ (mit Private Host Key signiert)
 - 2.4. Übertragen wird $(K_s | f | s)$
 3. Client verifiziert Signatur, berechnet K und H

Schlüsselmaterial

- Zum Ableiten des Keymaterials werden K und H verwendet:

- IV Client -> Server = Hash (K | H | "A" | session_id)
- IV Server -> Client = Hash (K | H | "B" | session_id)
- Encryption Key Client ->Server = Hash (K | H | "C" | session_id)
- Encryption Key Server-> Client = Hash (K | H | "D" | session_id)
- Integrity Key Client -> Server = Hash (K | H | "E" | session_id)
- Integrity Key Server -> Client = Hash (K | H | "F" | session_id)

- Daten werden immer vom Anfang des Hashes genommen

- Falls mehr Daten benötigt werden:

$K1 = \text{Hash}(K | H | x | \text{session_id})$ mit x aus ["A" ... "F"]

$K2 = \text{Hash}(K | H | K1)$

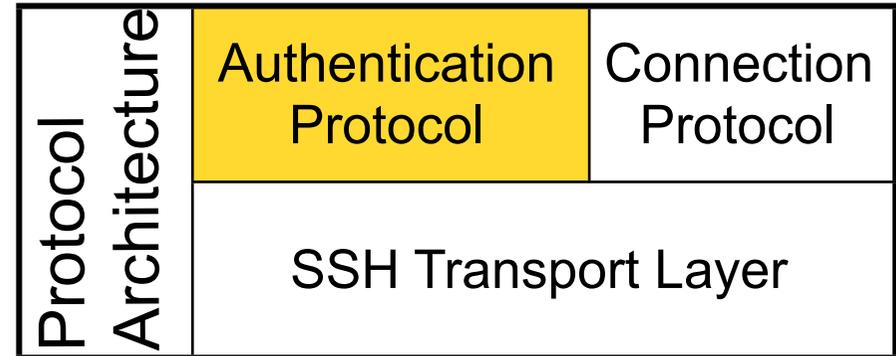
...

$KX = K1 | K2 | \dots | Kn$

- Bis zu diesem Zeitpunkt weiß der Server nichts über den Client!

SSH Architektur

■ Geschichtete Architektur:



■ SSH Authentication Protocol [RFC 4252]

- Authentisierung des Clients gegenüber dem Server

■ SSH Transport Layer Security [RFC 4253]

- Authentisierung des Servers gegenüber dem Client
- Vertraulichkeit
- Integrität
- Perfect Forward Secrecy

■ SSH Connection Protocol [RFC 4254]

- Multiplexing

SSH Authentication Protocol

- Zur Authentisierung des Nutzers
- Verwendet vertraulichen Kanal des SSH Transport Protocols
 - damit sichere Übertragung von Passwörtern o.ä. möglich
- Authentisierungsarten:
 - NONE
 - Public-Key: muss von allen Implementierungen unterstützt werden. Private Key sollte passphrase-protected sein!
 - Password: Sollte unterstützt werden, überträgt ganzes Passwort
 - Hostbased: Optional
 - keyboard-interactive: Fragen-und-Antworten-Ablauf, z.B. für SecurID
 - GSS-API: Externe Authentifizierung, i.d.R. via Kerberos (single sign-on)

Authentication: Host Based

- Angelehnt an den .rhosts Mechanismus der r-Commands
- Vertrauenswürdige Hosts und Benutzernamen werden eingetragen in
 - /etc/hosts.equiv oder /etc/shosts.equiv
 - ~/.rhosts oder ~/.shosts
 - Bsp.: testsrv.lrz.de reiser
- Client-Host wird über Host-Key (vgl. Transport Protocol) authentisiert
- Damit Verhinderung von IP-, DNS-, oder routing-spoofing
- ABER:
 - Keine richtige Benutzerauthentisierung
 - Alle Nutzer auf der (Client-) Maschine können Server nutzen
- Sollte nicht verwendet werden

Authentisierung: Password

- Client fordert Nutzer zu Eingabe auf von
 - Benutzername
 - Passwort
- Validierung gegen (Server-) lokale Benutzerdaten
- Hinweis: Transport Layer bietet verschlüsselten Kanal
- Benutzername und Passwort dadurch vor Eve geschützt
- Passwort wird aber vollständig übertragen
 - Ist also kein Challenge-Response-Protokoll!
 - Böartiger SSH-Server kann Benutzerpasswörter abgreifen

Authentication: Public Key

- Nutzer muss Schlüsselpaar besitzen
 - ssh-keygen erzeugt Schlüsselpaar
 - Private Key mit Passphrase verschlüsselt
 - Private Key gespeichert in ~/.ssh/id_dsa oder id_rsa
 - Public Key in ~/.ssh/id_dsa.pub oder id_rsa.pub
 - Public Key auf Zielmaschine in ~/.ssh/authorized_keys kopieren
- Client signiert Nachricht
- Verifikation durch Server
- Sicherste Variante der Authentisierung
- ABER:
 - Leere Passphrase für Private Key erlaubt
 - Schutz ist dann einzig durch Dateisystemberechtigungen gegeben
 - Falls Maschine kompromittiert, kann der Angreifer in diesem Fall private Schlüssel zum Login (auf anderen Maschinen) verwenden
- **KEINE leere Passphrase verwenden!!**

Einschub: RSA-768 geknackt (01/2010)

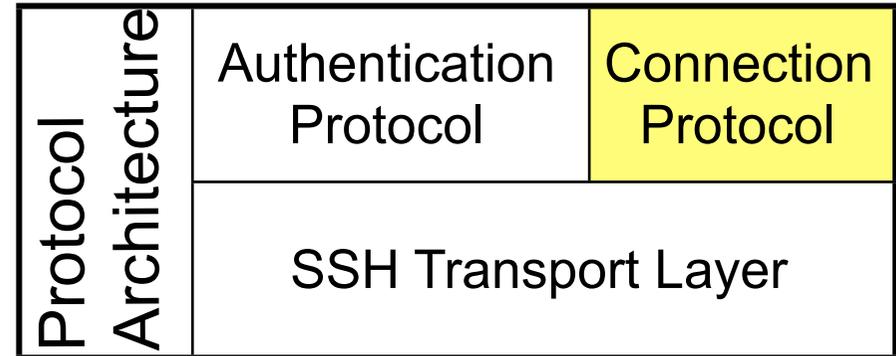
- Details unter <http://eprint.iacr.org/2010/006.pdf>
- 768 Bit (232 Dezimalstellen) lange Zahl in ihre Primfaktoren zerlegt
- Zahl wurde im Rahmen der RSA-Challenge als RSA-768 veröffentlicht
- RSA-Schlüssel mit „nur“ 768 gelten somit als geknackt

- Dauer: Ca. 2,5 Jahre
 - Erster Rechenschritt (Polynomial Selection) ca. 0,5 Jahre auf 80 PCs
 - Zweiter Rechenschritt (Sieving) ca. 2 Jahre auf Cluster mit mehreren hundert Rechnern
 - Eine aktuelle AMD Server CPU hätte alleine rund 1.500 Jahre benötigt

- Prognose: Genug Rechenleistung zum Knacken von RSA-1024 in 10 Jahren einfach verfügbar.

SSH Architektur

■ Geschichtete Architektur:



■ SSH Authentication Protocol [RFC 4252]

- Authentisierung des Clients gegenüber dem Server

■ SSH Transport Layer Security [RFC 4253]

- Authentisierung des Servers gegenüber dem Client
- Vertraulichkeit
- Integrität
- Perfect Forward Secrecy

■ SSH Connection Protocol [RFC 4254]

- Multiplexing

SSH Connection Protocol

- Setzt auf SSH Transport Protocol auf
- Dienste:
 - Interaktive Login Session (`ssh username@testsrv.lrz.de`)
 - Entfernte Ausführung von Kommandos (`ssh testsrv.lrz.de ps`)
 - X11-Forwarding
 - Statisches Forwarding von einzelnen TCP/IP - Ports (lokal / remote)
 - Inzwischen auch dynamisches Forwarding auf SOCKS5-Basis
- Für jeden Dienst wird ein oder mehrere Channels aufgebaut
- Multiplexing aller Channels über einen SSH Transport Protocol Kanal
 - Damit Vertraulichkeit und
 - Integritätssicherung aller übertragenen Daten

SSH Connection Protocol: Aufbau v. Channels

- Jede Seite kann `SSH_MSG_CHANNEL_OPEN` schicken
- Nachrichtenformat
 - channel type: String der Art des Kanals („session“, „X11“, usw.)
 - sender channel: 32 Bit Integer als lokaler Identifikator des Channels
 - initial window size: 32 Bit Integer; maximale Anzahl Bytes, die an den Initiator gesendet werden können
 - maximum packet size: 32 Bit Integer; Maximale Paketgröße
 - ... weitere Parameter abhängig vom channel type
- Ablehnen des Channel Request mit `SSH_MSG_CHANNEL_OPEN_FAILURE`:
 - recipient channel: channel id des Senders
 - reason code: Grund der Ablehnung (z.B. administratively prohibited, connect failed, unknown channel, ...)
 - additional textual information
 - language tag

Aufbau von Channels (cont.)

- Akzeptieren des Channel Request mit `SSH_MSG_CHANNEL_OPEN_CONFIRMATION`:
 - ❑ recipient channel: channel id des Senders
 - ❑ sender channel: channel id des Responders
 - ❑ initial window size
 - ❑ maximum packet size
 - ❑ ... weitere Parameter abhängig vom channel type
- Falls Channel aufgebaut, sind folgende Aktionen möglich
 - ❑ Datentransfer (Empfänger muss wissen, was er mit den Daten tun soll, d.h. ggf. weitere Aushandlung erforderlich)
 - ❑ Channel-type spezifische Nachrichten
 - ❑ Schließen des Channels

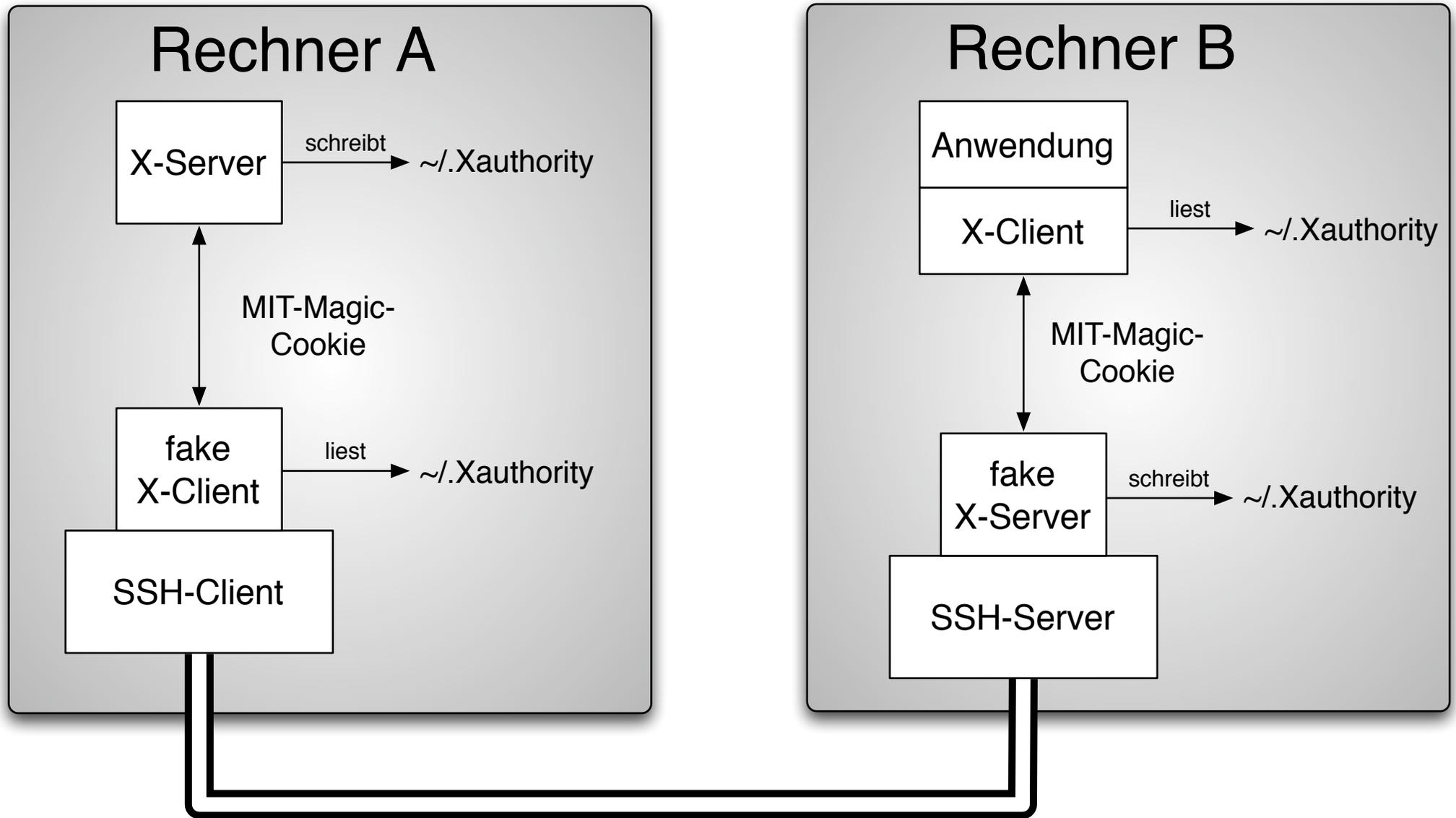
Interaktive Login Session

- SSH_MSG_CHANNEL_OPEN mit type „session“
- Nach Channel-Aufbau Anforderung eines Pseudo-Terminals;
SSH_MSG_CHANNEL_REQUEST:
 - recipient channel
 - „pty-request“: Anforderung des Pseudo-Terminals (PTY)
 - want_reply: Boolesche Variable ob Antwort erforderlich
 - Term Environment Variable: Art des PTY (z.B. vt100, xterm,....)
 - Terminal width und height characters: Anzahl der Zeichen
 - Terminal width und height pixel: Anzahl der Pixel
 - Terminal modes: Betriebsarten des Terminals
- Umgebungsvariablen setzen mit
SSH_MSG_CHANNEL_REQUEST mit type „env“
 - Paare mit Variablenname und Variablenbelegung
- Starten einer Shell mit SSH_MSG_CHANNEL_REQUEST mit
type „shell“ (Default shell aus /etc/passwd wird gestartet)

X11

- X11 Window System im Rahmen von MIT Athena Projekt entwickelt
- X-Server läuft auf dem Rechner, bei dem die Grafikausgabe bzw. -eingabe erfolgen soll
- Programm, das Ausgabe macht, verbindet sich als X-Client
- X-Server (X-Display) und X-Clients (Anwendungen) können auf getrennten Maschinen laufen
- Nachrichten werden im Klartext übertragen, d.h. X11 hat selbst keine Mechanismen bzgl. Vertraulichkeit

SSH: X11-Forwarding



SSH: Port Forwarding

- Port auf einer lokalen Maschine wird zu einem Port auf entferntem System über SSH getunnelt
- Bsp.: Sichere Remote Administration über VNC und SSH
- VNC-Server wartet auf Port 5900
- VNCViewer erwartet Display-Nummer; Display Nummern n werden auf Port 5900 + n abgebildet
- `ssh -L 5902:mobject:5900 admin@mobject`
- VNCViewer localhost:2

