

Kryptographie

Klaus Kusche

Inhalt (1)

Wörtlich:

*“Kryptographie” =
“Die Lehre von der Verschlüsselung”*

Hier etwas weiter gefasst:

- **Verschlüsselungsverfahren**
- **Hashes**
- **Signaturen & Zertifikate**
- **... (nächste Seite)**

Inhalt (2)

- **Anwendung: PGP, De-Mail**
- **Anwendung: Blockchain (z.B. Bitcoin)**
- **Kryptographische Zufallszahlen**
- **Passwörter**
- **Steganographie**
- ... (nächste Seite)

Inhalt (3)

- **Typische Sicherheitslücken, sichere Programmierung (separate Folien)**
- **Anwendung: Disk Encryption**
- **Anwendung: TPM**

Ziel

- **Überblick und Verständnis**, z.B. von
 - Verschlüsselungsverfahren
 - deren Anwendungen
- **Sensibilisierung**, z.B. für
 - typische Designprobleme beim Einsatz krypt. Verfahren
 - typische sicherheitsrelevante Fehler bei der Programmierung
 - ...

Nicht Ziel

- **Exakte Detail-Kenntnis von**
 - Verschlüsselungs-Algorithmen
 - Mathematischen Grundlagen
- **Praktische Programmierung**

Voraussetzungen

- **Grundkenntnisse:**
 - Informatik, vor allem Programmierung
 - Zahlentheorie
(Kryptographie ist ein Teilgebiet der Mathematik, nicht der Informatik!)
- **“Blick in die Realität”**
(z.B. Heise Newsticker, ...)

Motivation

- Man braucht nur die Fachnachrichten verfolgen...
- Fehler in diesem Bereich sind

besonders “folgenreich”!

(rechtlich, finanziell, medial)

Krypt. Verfahren: Anwendungen

- **Geheimhaltung / Vertraulichkeit / Zugriffsschutz:**
Verschlüsselungsverfahren
- **Integrität =**
Schutz gegen Verfälschung / Veränderung:
Kryptographische Hash-Funktionen
(auch z.B.: Blockchain)
- **Authentizität / Nichtabstreitbarkeit =**
Sicherstellung des Senders / Autors:
Signaturen und Zertifikate
- **Authentifizierung und Passwörter**

Wichtiger Hinweis (1)

Design und Programmierung von

- Verschlüsselungs-, Signatur- oder Hash-
Algorithmen
- Passwort-Speichern
- Verschlüsselten Netz-Protokollen
- ...

Niemals selber machen!

Wichtiger Hinweis (2)

Immer

- vorhandene, weit verbreitete, für “gut” befundene Implementierungen (am besten Open Source)
- von vorhandenen, am besten standardisierten Verfahren bzw. Algorithmen

verwenden!

(z.B. *Java Cryptography Architecture*,

OpenSSL / GnuTLS / LibreSSL, nss,

libgcrypt, NaCl, linux kernel, ...)

Symmetrische Verfahren (1)

Verwenden

- *denselben Schlüssel*
- manchmal sogar *denselben Algorithmus*
zum *Verschlüsseln und Entschlüsseln*

Arbeiten *blockweise*: Blockgröße 64-512 Bit

Typische *Schlüssellänge*: 128-256 Bit (*64 Bit zu wenig!*)

Symmetrische Verfahren (2)

Basieren auf einfachen Bit-Operationen:

Xor, Shift & Rotation (auch datenabhängig!),

Plus / Minus, Permutationen, ...

(nur Shift und Xor ist zu wenig!!!)

(meist keine Multiplikation usw.!)

Relativ einfache Algorithmen,

oft in mehreren "Runden" wiederholt

==> **In HW implementierbar**: "Krypto-Beschleuniger"

==> **Vektor-Einheit** (SSE- / AVX-Befehle) verwendbar!

==> **Schnell**: Einige 100 MB/s in SW!

Symmetrische Verfahren (3)

Beispiele:

DES (zu kurzer Schlüssel: 56 Bits!)

Triple-DES

IDEA (unfrei)

AES “*Advanced Encryption Standard*” = Rijndael

(Rijndael hatte von allen AES-Kandidaten die schwächste Sicherheit, aber die höchste Geschwindigkeit in HW und SW)

(Blowfish), *Twofish*, *Serpent*

(frei, u.a. Platten-Verschlüsselung, Twofish gilt als am sichersten)

CAST, MARS, RC6

Symmetrische Verfahren (4)

Problem bei symmetrischen Verfahren:

“Schlüsseltausch”

Wie kommt der geheime Schlüssel

z.B. über das Netz

*vom Sender zum Empfänger (oder umgekehrt),
ohne dass ihn ein Dritter abfangen kann?*

Asymmetrische Verfahren (1)

Verwenden **zwei zusammengehörige Schlüssel** (“Schlüssel-Paar”):

- “Öffentlicher” Schlüssel zum Verschlüsseln
- “Privater” Schlüssel zum Entschlüsseln

Deshalb auch:

“Public-Key-Verfahren”

Asymmetrische Verfahren (2)

- Der Empfänger berechnet einmal sein **persönliches Schlüsselpaar** (für alle zukünftigen Verschlüsselungen zu ihm),
- veröffentlicht den öffentlichen Schlüssel und hält den privaten Schlüssel geheim

=> Jeder kann verschlüsseln,
nur der “richtige” Empfänger kann entschlüsseln!

=> **Kein Schlüsseltausch-Problem mehr!**
(öffentlicher Schlüssel kann problemlos im Klartext zum Sender geschickt werden!)

Asymmetrische Verfahren (3)

Arbeiten ebenfalls blockweise,
aber mit größeren Block- und Schlüssellängen:
2048-4096 Bits (512 Bit, ev. 1024 Bit sind zu wenig!)

Arbeiten intern mit entsprechend langen int-Zahlen
(2048 Bits ... Dezimalzahl mit über 600 Ziffern!)
und mit Multiplikation, Restrechnung, ...

=> Aufwändiger, mindestens 1000 mal langsamer!

=> Schlecht durch Hardware oder Vektor-Befehle
zu beschleunigen.

=> Unterschiedliche Algorithmen zur Ver- und Entschlüsselung

Asymmetrische Verfahren (4)

Basieren auf Problemen der diskreten Mathematik (z.B. Zahlentheorie), die

- ***in einer Richtung schnell zu berechnen sind***
(Berechnen eines neuen Schlüsselpaares)
- ***in der anderen Richtung nur extrem aufwändig***
(z.B. nur durch Durchprobieren aller Zahlen)
zu lösen sind
(Berechnen des privaten Schlüssels
aus dem öffentlichen Schlüssel)

“Einweg-Funktion” bzw. “Falltür-Probleme”

Asymmetrische Verfahren (5)

Beispiel 1: Faktorisierung großer Zahlen

Wenn p und q zwei sehr große Primzahlen sind,

- dann ist $n = p * q$ "leicht" zu berechnen,
- aber die Berechnung von p und q aus n praktisch unmöglich
(obwohl theoretisch bekannt
und im Prinzip eindeutig lösbar)

Anwendung u.a.: ***RSA-Verschlüsselung***

Asymmetrische Verfahren (6)

Beispiel 2: “Diskreter Logarithmus”

$$a^x \text{ kongruent } m \text{ mod } p$$

für a, x, m, p ganzzahlig

- Leicht für a, x, p gegeben und m gesucht
- Sehr schwer für a, m, p gegeben (und groß!) und kleinstes lösendes x gesucht

Anwendung: **Diffie-Hellman, DSH, Elgamal**

Asymmetrische Verfahren (7)

Ähnliche Idee: **ECC** = “*Elliptische Kurven*”

Im wesentlichen auch ein diskreter Logarithmus,
aber nicht mit ganzen Zahlen,
sondern mit Punkten auf Kurven!

Einzelne Rechenschritte sind

“schwieriger” und schlechter auf HW optimierbar

=> Kürzere Schlüssel reichen aus:

Derzeit 256 Bits, ev. 512 Bits

ECDSA für Signaturen, **ECDH** für Schlüsseltausch

“**Curve25519**” gilt als bestes ECC-Verfahren

Asymmetrische Verfahren (8)

Unterschied RSA / DSA / ECDSA und DH / ECDH:

DH verwendet zusätzlich Zufallszahlen

=> DH bietet “**Forward secrecy**”, RSA nicht:

- Kenntnis des privaten asymmetrischen Keys:
 - aufgezeichnete verschlüsselte Daten
 - symmetrische Keysalter (schon beendeter) Sessions
nicht nachträglich knackbar
- Kenntnis des symmetrischen Keys einer Session
=> *Andere Sessions nicht knackbar*

Hybride Verfahren

In der Praxis:

Kombination beider Verfahren
(z.B. SSL/TLS, IPsec, PGP, ...):

- Zuerst asymmetrisches Verfahren
zum ***Austausch des symmetrischen Schlüssels***,
z.B. Diffie-Hellman-Schlüsseltausch
(sym. Schlüssel ist kurz ==> darf langsam sein!)
- Dann symmetrisches Verfahren
zum Austausch der ***Nutzdaten***
(schnell)

Krypto-Analyse

Angriffe durch “*Krypto-Analyse*”:

- (\Rightarrow) Klartext herausfinden)
- \Rightarrow Schlüssel herausfinden!!!

Ein Verschlüsselungsverfahren ist gebrochen,
wenn das mit deutlich weniger Aufwand als
“Brute Force” (= alles durchprobieren)
gelingt!

(z.B. 2^{96} statt 2^{128} Versuche)

Anforderungen (1)

- Kein statistischer / struktureller Zusammenhang zwischen Klartext und Chiffre:

“Pseudo-zufälliger Output”

- Keine erkennbarer Zusammenhang
*Bestimmte Bits im Input / Schlüssel \Leftrightarrow
Bestimmte Bits im Output*
 - Keine Beschreibung durch ein
algebraisches Gleichungssystem
- \Rightarrow Verhindert u.a. ***statistische Analysen***
(z.B. *Zeichen-Häufigkeit*)

Anforderungen (2)

Beständigkeit gegen “*Known Plaintext*”-Attacken:

- Klartext und Chiffrat bekannt
(oder Chiffrat und Teile des Klartextes bekannt),
- Schlüssel gesucht

Extremfall:

Krypto-Box in Händen des Angreifers

=> beliebig viele Klartext/Chiffrat-Paare berechenbar!

*Trotzdem darf der Schlüssel daraus nicht “leicht”
(schneller als durch Brute Force) ermittelbar sein!*

Anforderungen (3)

Sonderfall “***Differenzielle Analyse***”:

Ganz *kleine Änderung im Original*

=> Rückschluss auf den Schlüssel / Schlüssel-Teile
aus resultierender Änderung im Chifftrat?

=> Anforderung “***Lawinen-Effekt***” = “***Diffusion***”:

Kleine Änderungen im Klartext

bewirken *große Änderungen im Chifftrat*

(Ein *einziges* Bit im Input ändern

=> *Jedes* Bit im Output ändert sich
mit 50 % Wahrscheinlichkeit)

Anforderungen (4)

Beständigkeit gegen “*Seitenkanal-Attacken*”

- Timing- und Power-Attacken
- Rückschlüsse aus Sprung- und Cache-Verhalten
- Analyse der Funk-Abstrahlung

==> Der Algorithmus muss
für Beobachter von außen
für alle Inputs & Schlüssel
“gleich” rechnen!

Anforderungen in Zukunft?

“*Quantencomputer-feste*” Krypto-Algorithmen

Weil: Quantencomputer können viele Probleme grundlegend schneller lösen!

Bisherige konkrete Ergebnisse: Quanten-Algorithmen

- ... halbieren die effektive Schlüssellänge von AES:
128 \Rightarrow 64 Bit
- ... faktorisieren/logarithmieren in polynomialer Zeit:
 $\sim L^*L^*\log(L)^*\log(\log(L))$ Schritte bei Zahl mit L Ziffern
(Faktorisierung bisher:
Superpolynomiale, aber subexponentielle Zeit: $\sim \exp(\sqrt{L^\log(L)})$)*

Verschlüsselung langer Daten (1)

Wie verwende ich die bisherigen Verfahren
(arbeiten alle auf fixer Blockgröße!),
um einen beliebig langen Strom von Daten
zu verschlüsseln?

=

Wie mache ich
aus einem **Blockchiffre**-Verfahren
ein **Stromchiffre**-Verfahren?

Verschlüsselung langer Daten (2)

ECB (“Electronic Code Book Mode”):

- Datenstrom in Blöcke teilen
- Jeden Block separat verschlüsseln

Nachteile:

- ***Gleicher Block ==> gleiches Chiffre***
- Block-Reihenfolge ev. unbemerkt verfälschbar

Vorteil:

- Wiederaufsetzen nach Fehlern / Verlusten
(nur ein Block kaputt)

Verschlüsselung langer Daten (3)

CBC (“Cipher Block Chaining Mode”):

Bei jedem Block vor der Verschlüsselung:

Xor mit vorigem Chiffrat

(beim ersten Block:

Xor mit beliebigem ***“Initialisierungsvektor”***,

z.B. Zufallszahl)

==> Problem mit gleichen Blöcken behoben

==> Vertauschung der Reihenfolge fällt auf

Bei Fehlern: 2 Blöcke kaputt, dann wieder ok

Verschlüsselung langer Daten (4)

CFB (“Cipher Feedback Mode”)

Verkettung ähnlich CBC:

Xor mit Plaintext
nach der Verschlüsselung
des vorigen Chiffrats

Selten!

Verschlüsselung langer Daten (5)

CTR (“Counter Mode”):

Arbeitet pro Block, ohne Verkettung:

Verschlüsselt wird

Zufallszahl + fortlaufender Zähler

==> Erzeugt jedesmal anderen

“zufälligen” Bitstrom

Klartext wird danach dazu verschlüsselt:

Xor mit diesem Bitstrom

Verschlüsselung langer Daten (6)

Verschlüsselung und Entschlüsselung sind ident

Beide sind

- parallelisierbar (jeder Block für sich berechenbar)
- vorausrechenbar (Verschlüsselungs-Berechnung hängt nicht vom Klartext ab)

Echte ***Stromchiffre***, kann Byte für Byte arbeiten!

Bei Bitfehlern in der Übertragung:

Nur genau entsprechende Bits

der entschlüsselten Daten betroffen!

(kann Vorteil oder Nachteil sein)

Verschlüsselung langer Daten (7)

OFB (“Output Feedback Mode”):

Ähnlich CTR, aber mit Verkettung statt mit Zähler:

Initialisierungsvektor wird
immer wieder verschlüsselt

Klartext wird danach mit Xor dazu verschlüsselt

Entschlüsselung ist ident

Verschlüsselung langer Daten (8)

GCM (“Galois/Counter Mode”):

Wie CTR mit Zähler,
aber mit zusätzlicher Auth-Tag-Berechnung

Gilt als derzeit bestes Verfahren,
in vielen Standards verwendet:

IPsec, SSH, TLS, OpenVPN, viele HW-Encodings, ...

Neben-Nutzen:

- Prüfsumme / Fehlererkennung
- Authentifizierung “**GMAC**”

Hash & Signatur

Bei Dateien, Mails, ...:

- Nur Unverfälschtheit:

Hash (kein Schlüssel o.ä.)

- Unverfälschtheit +
Authentizität des Absenders +
“Nicht-Abstreitbarkeit” des Absenders

Hash

+ **Signatur**

+ **Zertifikats-Infrastruktur**

Hash (1)

Hash = “Prüfsumme” (“fingerprint”)

Bei der Erzeugung: Kein Passwort oder Schlüssel

==> Jeder kann Hash erstellen und prüfen

Hashwert ist normalerweise kürzer als Nachricht (z.B. fix 256 Bits), daher keine bijektive Funktion

==> Informationsverlust:

Original-Nachricht nicht aus Hash herstellbar

==> Verschiedene Original-Nachrichten

können (ganz selten!) gleichen Hash haben

Hash (2)

Unterschied zu Hashfunktion z.B. bei Hashtable:
Mit besonderen Eigenschaften / Anforderungen!

“Kryptographische Hashfunktion”

- ***Kollisionsresistent*** (nächste Folie)
- ***“Lawineneffekt”***:
Kleine Änderungen im Original
==> *Große* Änderungen im Hash
- ***Einweg-Funktion***: Darf nie “rückrechenbar” sein
(selbst bei gleicher Länge wie Original-Nachricht)

Kollision Fall 1 (1)

Gegeben, fix:

- Nur Hashwert
- Oder Nachricht A und deren Hashwert

Gesucht:

Andere Nachricht B, die denselben Hash ergibt

==> A kann durch B ersetzt werden,
ohne dass es auffällt

==> Wenn der Hash des echten Passwortes A
bekannt ist, wird auch B als Passwort akzeptiert
(ohne dass man A kennen muss)

Kollision Fall 1 (2)

Ziel bei Dateien / Nachrichten in der Praxis:

Konstruiere B so, dass es besteht aus

- Großen Teilen, die ident zu A sind (z.B. Nachrichten-Anfang)
- Einer (kleinen) gezielten, fixen, böswilligen Veränderung im Vergleich zu A
- Einigen beliebigen, “dazukonstruierten”, möglichst unauffälligen Veränderungen zur “Korrektur” des Hash-Wertes

Kollision Fall 2

Gegeben:

Oft: Gewünschte Nachrichten-Schnipsel (z.B. Anfang)

Gesucht: Konstruiere 2 verschiedene Nachrichten, die diese Schnipsel enthalten und denselben (aber nicht fix vorgegebenen) Hashwert haben

==> z.B.: Verschicke B, aber behaupte nachher, dass A die echte Nachricht ist

In vielen Fällen:

Erstes Pärchen ist schwer, weitere gehen “schnell”

Kollision allgemein

*Ein Hash-Verfahren ist gebrochen,
wenn Fall 1 oder Fall 2 deutlich schneller
als mit "Durchprobieren" zu lösen ist!*

(Fall 2 ist meist leichter als Fall 1)

Hash-Verfahren

MD4 / MD5: “Message Digest”:

Zu alt (seit 1990 und 1992), zu wenige Bits (128),
seit > 10 Jahren praktisch geknackt
(auf einem PC in wenigen Minuten)

SHA, SHA-1 (1995): “Secure Hash Algorithm”

Seit 2005 theoretisch und seit 2017 praktisch
geknackt (hat 160 Bits, Hack-Komplexität $\leq 2^{64}$)

SHA-2 = SHA-256, SHA-384 und SHA-512:

Längere Varianten davon, noch sicher

SHA-3 (seit 2015)

Hash: Anwendungen (1)

- Unverfälschtheit von Dokumenten & Nachrichten

Siehe später:

- Speicherung von Passwörtern
- Lange / hochwertige Zufallszahlen
- One Time Passwords
- Challenge-Response-Verfahren
- Blockchain

Hash: Anwendungen (2)

*Das gesamte System der **Krypto-Zertifikate** beruht kritisch auf Hash-Funktionen!*

Hash-Funktion knackbar

==> Zertifikate fälschbar oder falsch zertifizierbar,
ohne dass es auffällt

(siehe z.B.: <https://de.wikipedia.org/wiki/Kollisionsangriff> :
Wie mache ich mich selbst heimlich zur Zertifizierungsstelle?)

Signatur (1)

Der Sender / Ersteller

- Berechnet Hashwert der zu signierenden Daten
- Verschlüsselt den Hash mit seinem Private Key
- Verschickt Originaldaten
plus Signatur = verschlüsselter Hashwert

==> Nur der Besitzer des Private Key kann signieren!

Der Empfänger

- Entschlüsselt Signatur mit Public Key
- Berechnet Hash der Originaldaten & vergleicht

==> Jeder kann Signatur prüfen!

Signatur (2)

Entschlüsselte Signatur und Hashwert der Daten sind nur gleich wenn:

- Daten und Signatur unverfälscht sind
- Der Public Key zum Private Key passt

Wieso signiert man den Hash statt der Daten selbst?

- Bei Daten: *Zu einer beliebigen Signatur wären “dazupassende” Daten berechenbar, ohne den Private Key zu kennen!*
- Rechenaufwand bei langen Daten

Voraussetzungen Signatur (1)

Das System ist nur vertrauenswürdig, solange

- 1.) Niemand außer dem Absender
Zugang zum Private Key hat
- 2.) Der **Zusammenhang zwischen
Schlüsselpaar und Identität der Person**
zuverlässig nachvollziehbar ist

Voraussetzungen Signatur (2)

Zu 1.):

Z.B. “*Qualifizierte elektronische Signatur*”:

Private Key ist auf Chipcard gespeichert und
verlässt die Karte nie!

=> Aktive Karte, kann rechnen

=> Daten werden zum Signieren auf die Karte
geladen, Signatur wird auf der Karte berechnet

Die CA darf keine Kopie des Private Key haben!

Voraussetzungen Signatur (3)

Zu 2.):

PKI = “Public Key Infrastructure”

System zur

- Sicherstellung der ***Vertrauenswürdigkeit*** von Public Keys
- ... und ihrer ***Zuordnung*** zu Personen, Firmen, Domains, ...
- sicheren *Identifizierung* von Personen, z.B. via Chipkarten

PKI (1)

PKI's basieren auf Zertifikaten

Zertifikat =

- *Public Key + Informationen* dazu
- ausgestellt von einer Zertifizierungsstelle
= CA "*Certificate Authority*"
- *signiert* mit dem *Key der CA*
d.h. überprüfbar, manipulationssicher, ...

PKI (2)

Informationen in einem Zertifikat,
z.B. im genormten X.509-Format:

- **Public Key**
- **Person / Firma**, zu der dieser Key gehört
- Bei https-Keys: **Für welche Domains** zulässig?
- **Aussteller** des Zertifikats (= CA),
fortlaufende Nummer
- **Gültigkeitsdatum** Anfang und Ende
- Versions- und Algorithmus-Information

PKI (3)

Aber wie prüft man den Key der CA
(um die Signatur des Zertifikats zu prüfen)???

Hierarchisches System:

Übergeordnete CA zertifiziert *Key der CA*

==> Für jedes Zertifikat muss gelten:

Lückenlose Kette von *CA's* und ihren Zertifikaten

bis zu einer **“Root CA”**

(Root CA hat “*self-signed-Zertifikat*”
= signiert ihr Zertifikat selbst mit eigenem Key)

PKI (4)

Weltweit gibt es ein paar hundert Root CA's

==> ***Explizite Liste***

von Root CA's und deren Zertifikaten

fix im Betriebssystem / Browser gespeichert,

gelten ohne weitere Prüfung als vertrauenswürdig

Beispiele Root-CA's:

- VeriSign (!?), DigiCert, D-Trust, ... (kosten!)
- *freie* "Community CA's", z.B. CAcert, "Let's Encrypt"
(CAcert ist u.a. die CA hinter der freien Krypto-Initiative von c't)

PKI (5)

Überprüfung eines Zertifikates:

- **Signatur** des Zertifikates korrekt?
=> Daten im Zertifikat vertrauenswürdig
- Zertifikat **zeitlich gültig?**
(im Zertifikat gespeichertes von...bis-Datum)
- Oft: **Richtiges** Zertifikat?
(z.B. passt die Domain im Zertifikat zur Domain in der URL?)
- Zertifikat **widerrufen?** (OCSP, CRL)
- Zertifikat **vertrauenswürdig?**
(Zertifizierungskette bis zu einer Root-CA ok?)

Aufgaben einer CA

- CA (oder vorgeschaltete Registrierungsstelle) prüft Korrektheit der Zertifikats-Daten, Identität und Berechtigung des Antragstellers
- CA erzeugt Schlüsselpaar
- CA stellt Zertifikat für Public Key aus, übermittelt Private Key geheim an den Antragsteller
- CA führt Liste aller von ihr ausgestellten Zertifikate (Antragsteller, öffentl. Schlüssel, ...)
- CA verwaltet Sperrliste

Sperren von Zertifikaten (1)

Wenn:

- Private Key geklaut
- Daten des Antragstellers stimmen nicht mehr
- Kriminelle Aktivitäten des Antragstellers

==> Ungültig-Erklärung vor Ablauf der Gültigkeit

- ***CRL = “Certificate Revocation List”:***
Sperrliste, “Gegenteil” der Root-CA-Liste

Im Browser / OS gespeicherte Datei,
wird regelmäßig automatisch aktualisiert (?)

- ***Online-Zertifikats-Prüfdienst, z.B. OCSP***

Sperren von Zertifikaten (2)

Auch Sperre einer CA

= **Sperre aller von ihr ausgestellten Zertifikate**

z.B. weil

- CA stellt Zertifikate an Unberechtigte aus (z.B. Problem Google / Symantec), besonders Sub-CA-Zertifikate (zum Zertifizieren)
- Privater Schlüssel der CA wird kompromittiert
- Angreifer bricht in CA ein und stellt (heimlich?!) Schlüsselpaare & Zertifikate aus
- CA arbeitet unsicher (z.B. “verliert” Private Keys)

Probleme CA-System (1)

Kette zu einer Root CA irgendwo unterbrochen
(z.B. Vertrauen in eine CA verloren &
CA-Zertifikat widerrufen, passiert z.B. bei Einbruch)

==> alle "darunterliegenden" CA's und Zertifikate
werden **plötzlich ungültig!**

- Behebung für SSL-Zertifikate, ID-Karten, ...:
Neue Zertifikate ausstellen

==> Hoher organisatorischer Aufwand, dauert!

- Behebung bei signierten Dokumenten: **Keine!!!**
(weil Signatur nicht nachträglich änderbar ist!)

Probleme CA-System (2)

Angriffspunkt lokale Root-CA-Liste:

Dort Fake-CA-Eintrag dazu

==> “*Alles ist möglich*”

- “Böse” SSL-Server werden als vertrauenswürdig & einer falschen Firma zugehörig angezeigt
- “Man-in-the-middle-Attacken” auf SSL möglich

(Erweiterung der Root-CA-Liste war bzw. ist üblich
z.B. für Corporate Virens scanner, SW-Auto-Update,
für lokale Self-Signed-Zertifikate, ... ==> **Don't!**)

Probleme CA-System (3)

Beantragung von Zertifikaten kostet

Zeit, Aufwand und (teilweise) Geld

==> Viele Institutionen setzen intern selbsterzeugte **Self-Signed-Zertifikate** ein, wo Zertifikate technisch nötig sind, aber die Beantragung eingespart werden soll

Don't!!!

Erzieht Benutzer, "niedrige" Sicherheit bzw. "schwache" Zertifikate zu akzeptieren!

Alternative zum CA-System (1)

“Web of Trust”:

Nicht-hierarchisches

Netz gegenseitigen Vertrauens

ohne zentrale Institutionen (CA's), z.B. für PGP

2 Merkmale:

- ***X kennt Y*** (persönlich)
==> *X signiert Schlüssel von Y*
- ***X vertraut Y,***
dass Y *nur Schlüssel von Leuten signiert,*
die er tatsächlich kennt

Alternative zum CA-System (2)

X vertraut von Y signierten Schlüsseln

&

Y hat Schlüssel von Z als “bekannt” signiert

==>

X betrachtet Schlüssel von Z als gültig

Parameter zum “Finetuning”:

- **Weniger streng:**

Auch indirekt über mehrere “vertrauende” Personen hinweg

- **Strenger:**

Mehrere vertrauenswürdige Y müssen Z signiert haben

Alternative zum CA-System (3)

Dazu notwendig: *Schlüsselsever*

Enthalten

- *Public Keys* der Teilnehmer
- Gegenseitig ausgestellte *Signaturen*

Werden zur Validierung von Public Keys abgefragt

Achtung: *Datenschutz-Problem!!!*

(Namen, Mail-Adressen, “kennt” und “vertraut” sind sensible personenbezogene Daten!)

E-Mail-Verschlüsselung

Stand der Technik:

- **Keine** “*End-to-End-Verschlüsselung*”:
Mail liegt auf allen Zwischen-Servern im Klartext!
- Verschlüsselung bestenfalls “*Host-To-Host*”:
 - Zwischen zwei Servern
 - Zwischen Server und Client

Nach “kleinsten gemeinsamen Nenner”-Prinzip:
“*Verschlüsselungsunwilliger*” Server
==> Übertragung im Klartext!

E-Mail-Authentizität

Mail-Header sind “Schall und Rauch”:

Weder Verschlüsselung noch Prüfsumme möglich!

(nur der Mail Body könnte verschlüsselt werden)

- Header können schon vom Absender beliebig gefälscht werden
- Header können von jedem Zwischen-Server beliebig gefälscht werden

==> Absender, Datum, Empfänger, Mail-Weg, ...
ist nicht nachvollziehbar (z.B. Spam)

PGP (1)

“Pretty Good Privacy”

Standard: RFC4880

Implementierungen (frei):

- **OpenPGP**
- **GPG** = “Gnu Privacy Guard”

Primärer Einsatz:

Mail-Verschlüsselung & -Authentizität

Z.B. auch:

Absicherung automatischer SW-Verteilung

PGP (2)

Kombination sym. & asym. Verfahren:

- Falls gewünscht: Fügt Signatur zur Nachricht (verschlüsselt mit dem Private Key des Senders)
- Verschlüsselt den Mail-Inhalt symmetrisch: Schlüssel wird jedesmal zufällig erzeugt
- Verschlüsselt den sym. Schlüssel asymmetrisch (verschlüsselt mit dem Public Key des Empfängers) mit Elgamal (weil lizenz- und patentfrei, ähnlich Diffie-Hellman-Verfahren)
- Base64-Kodierung für die Mail (==> nur sichtbare ASCII-Zeichen)

PGP (3)

Probleme:

- Sicherheit des Schlüsselringes?
- Authentizität:
Wer garantiert die Zuordnung
zwischen Schlüsseln und Personen?

De-Mail (1)

Technisch:

- Über's Internet, normales Mail-Format, SMTP / POP / IMAP oder HTTPS, SSL/TLS ist Pflicht!
- Kein Austausch mit "normaler" Mail
- Zwei-Faktor-Authentifizierung Pflicht (z.B. mTAN), aber qualifizierte Signatur (Chipkarte) nicht
- Zusätzliche Sende- und Empfangs-Nachweise (wie "Einschreiben"), vom Dienstanbieter signiert

Organisatorisch:

- Nur von zertifizierten Anbietern
- Durch Ausweisprüfung identifizierte Benutzerkonten

De-Mail (2)

Sicherheit:

- Verpflichtend:
*Host-to-Host-Transport-Verschlüsselung (SSL/TLS),
zusätzlich Host-to-Host-Inhalts-Verschlüsselung*
- Aber: *Inhalt wird auf jedem Server entschlüsselt!*
*End-to-End-Verschlüsselung ist nur optional
(muss vom Sender / Empfänger mit eigener SW
erfolgen, z.B. PGP), aber zentraler Key-Server*
- *Hashwert für Integrität, optional: Signatur
(beides am ersten / letzten Server, nicht am Client!)*

De-Mail (3)

Viele Datenschutz-Aspekte offen:

- ***Vorratsdatenspeicherung*** des De-Mail-Verkehrs?
- Gesetzlich sehr großzügig geregelter Zugriff auf die ***Personendaten*** zu einer Mail-Adresse und auf das ***De-Mail-Passwort!***

==> Alles andere als “vertraulich” oder “geheim”!

==> Finger weg?!

Bitcoin (1)

... besteht aus bzw. ist der Name von ...

- Einer ***Blockchain*** zum Speichern von Transaktionen zwischen Wallets
- Einem ***Peer-to-Peer-Netzwerk*** von Knoten, die Daten austauschen über
 - einzelne Transaktionen
 - die Blöcke der Blockchain
- ***Regeln*** (und einem Client, der sie implementiert):

***Wie wird die Blockchain
um neue Transaktionen erweitert?***

Bitcoin (2)

... basiert an zwei Stellen auf **Kryptographie**:

- Erzeugung und Unverfälschbarkeit der Blockchain:

Hashing mit **SHA-256**

- Absicherung der Wallets,
Sicherstellung des Zugriffs nur durch den Besitzer:

Signatur mit 256 bit **ECDSA**

(= Public-Key-Verfahren mit elliptischen Kurven)

Blockchain (1)

Blockchain =

Manipulationssichere Liste von Blöcken

Jeder Block enthält:

- Kryptographischen *Hash des vorigen Blockes*
- *Zeitstempel*
- *Beliebige Nutzdaten*

==> Sehr viele Anwendungs-Möglichkeiten!
(nicht nur Krypto-Währungen)

Blockchain (2)

- Jede nachträgliche Modifikation des Inhalts eines Blockes oder der Block-Reihenfolge ist erkennbar (weil der Hash im nächsten Block nicht mehr stimmt!)
- Verwendung für alles, was fälschungssicher protokolliert werden muss
- Es wird nur hinten angehängt, alles davor muss unverändert erhalten bleiben:
=> *Blockchain wird immer größer*
(z.B. Bitcoin derzeit >> 100GB)

Blockchain (3)

Bei Bitcoin:

- Nutzdaten = *Transaktionsdaten*
(viele Bitcoin-Transaktionen pro Block)
- Nur die Transaktionen in der Blockchain sind
“bestätigt / gültig / unwiderruflich”!
(wegen Kollisionsbehandlung:
Erst ab einer gewissen Anzahl von Blöcken dahinter)
- Jeder Block enthält zusätzlich ein paar Bits,
die durch “Mining” bestimmt werden

Blockchain (4)

==> Bis jetzt wäre alles einfach!

Bei Krypto-Währungen usw.:

- Blockchain ist **dezentral** gespeichert:
Keine Hierarchie, keine Zentralstelle

==> Viele gleichberechtigte lokale Kopien,
über die ganze Welt verteilt!

==> Nicht nur “brave” Knoten!

- Kommunikation dazwischen ist nicht synchron:
Langsam, verzögert, unsicher, unzuverlässig, ...

Blockchain (5)

“Konsensproblem”

==> Wer darf anhängen?

==> Welcher neue Block wird von allen
als nächster gültiger Block akzeptiert?

==> “Einigkeit im Netz suchen”

- Weil asynchrones Netz, “bad guys”: **Schwierig!!!**
(Problem: Mehrere verschiedene, konkurrierende neue Blöcke
an verschiedenen Orten, die voneinander zu spät erfahren!)
- Viele Verfahren denkbar!
(z.B. iterative verteilte Mehrheits-Abstimmungen)

“Proof of work” (1)

“Proof-of-Work”-Konzept (z.B. Bitcoin):

***Der erste, der eine Rechenaufgabe richtig löst,
darf anhängen***

Problem Kollisionsbehandlung:

***Was ist, wenn mehrere
fast gleichzeitig “gewinnen”?***

Regeln legen fest, wer überlebt & wer verworfen wird

==> Längerer Zeitraum, in dem eine Transaktion
wieder aus der Blockchain rausfliegen kann!

“Proof of work” (2)

“Rechenaufgabe”: *Hash-Kollisions-Problem*

Gegeben:

- Große Teile des Datenblocks: Transaktionsdaten
- Kriterien für den Hashwert: *Vorne x viele 0-Bits*

Gesucht:

*“Richtige” Ergänzung des Datenblocks,
sodass der Hashwert die Kriterien erfüllt*

*Geht nur durch **Ausprobieren!***

(sehr viele Versuche, im Schnitt ein paar Trillionen!)

“Proof of work” (3)

Warum Rechenzeit für Mining opfern?

Für jede erfolgreiche Lösung bekommt der “Gewinner”:

- Eine **Belohnung**: Fixer Betrag Bitcoins, derzeit 12,5
- **Transaktionsgebühren**
der damit bestätigten Transaktionen (weniger)

*Mining-Belohnungen sind der einzigste Weg,
durch den Bitcoins neu entstehen!*

“Proof of work” (4)

Derzeit: Rund alle 10 Minuten weltweit eine Lösung.

- Mining **zu langsam**:

Folge: Rückstau von offenen Transaktionen!

==> Belohnung (Transaktionsgebühren) erhöhen,
bewirkt Steigerung der Rechenleistung

==> Kriterium erleichtern

- Mining **zu schnell** (z.B. schnellere Hardware):

==> Belohnung senken

==> Kriterium verschärfen (mehr Versuche nötig)

“Proof of work” (5)

Kollisionsbehandlung

(wenn unabhängig voneinander
zwei neue Blöcke parallel angehängt werden):

Warten!

... bis an einem der beiden Ende
ein weiterer Block dazukommt!

Dann: Längste Kette gewinnt!

==> “Verlierender” Block wird ungültig erklärt!
(dessen Transaktionen wandern zurück
in den Pool unbestätigter Transaktionen!)

“Proof of work” (5)

Gefahr:

Wer deutlich mehr Rechenleistung kontrolliert als alle anderen zusammen, kann das System sabotieren!

Wie?

- An einem alten Block ansetzen
 - Neue Nebenkette berechnen,
die schneller wächst als die Hauptkette
- ==> Letzte gültige Blocks der Hauptkette werden *nachträglich wieder ungültig!*

Vor dem Mining eines Blocks

- ***Prüfung*** der einzelnen Transaktionen, u.a. auf Besitz der transferierten Bitcoins
- Peer-to-Peer-Broadcast “neuer” Transaktionen
==> Verteilter Pool offener Transaktionen
- Jeder Miner für sich: ***Auswahl*** offener, aber noch nicht in der Blockchain bestätigter ***Transaktionen für den nächsten neuen Block***
(nach Transaktions-Gebühr, Wartezeit, Lust, ...)
- Eigene “Belohnungs-Transaktion” dazu!

Transaktionen

- ... transferieren Bitcoins von einem Wallet zu einem anderen
- ... werden mit dem ***Private Key des abgebenden Wallets signiert***
=> Nur der Besitzer des Wallets (des Keys) kann abgehende Transaktionen erzeugen!
Aber jeder kann sie mit dem Public Key prüfen!
(Wallet vorhanden, genug Bitcoins drin, ...)
=> *Hack-Überfälle auf Bitcoins*
waren meistens Diebstahl der privaten Schlüssel!

Kryptographische Zufallszahlen

Verwendung:

- Schlüsselgenerierung (z.B. https-Verbindungen)
- Initialisierungsvektor für Stromverschlüsselungen
- Challenge-Response-Abfragen
- Password-Salt, ...

Wichtige Eigenschaft aus Sicht der Kryptographie:

Die verwendete bzw. die nächste / vorige Zufallszahl darf nicht berechenbar / vorhersagbar sein!

Normale Zufallszahlen (1)

... sind *Pseudo-Zufallszahlen*,
d.h. nach *bekannten Formeln berechnet*
(aus einem internen Status des Generators)

==> Absolut *vorhersagbar!*
(letzte Zahlen bekannt ==> nächste berechenbar)

==> *Zyklisch* = Ein “Kreis” von Zahlen
(Ein Maß für die Qualität der Generatoren:
Länge des Zyklus = $2^{15} \dots 2^{\text{(einige Zehntausend)}}$,
wobei die Länge des Zyklus kleinergleich
(meist gleich) der Größe des internen Status ist)

==> Nicht für Kryptografie verwenden!!!

Normale Zufallszahlen (2)

Startwert = “**Seed**” = Anfangswert des internen Status

Bei gleichem Startwert:

Genau gleiche Folge von Zufallszahlen!

In der Praxis üblich:

Bei jedem Programmstart anderer Startwert

(bitte nicht “Zeit in Sekunden”,
sondern 64 bit Nanosekunden, TSC o.ä.!)

Trotzdem nicht für Kryptografie verwenden!!!

Physikalischer Zufall (1)

Wirklich zufällige Zahlen müssen auf Kombination mehrerer physikalischer Ereignisse basieren!

De facto meist Zeitabstände zwischen Interrupts:

- 1.) Netz-Paketen auf Ethernet, WLAN, ...
- 2.) Tastatureingaben, Mausbewegungen, Touch
- 3.) Wartezeit bei Festplatten-Zugriffen
(nicht bei SSD, streut zu wenig!)

plus **Bewegungs-Koordinaten** von Maus & Touch,
Touch-Zeit und -Stärke, ...

Physikalischer Zufall (2)

Nur in endlicher Menge / Datenrate verfügbar!

Problem vor allem ...

- kurz nach dem Booten: Noch wenig / kein I/O
- bei Servern und vor allem Embedded Devices, Smartcards, ...: Keine Eingabe-Geräte, keine beweglichen Teile, ev. kein Netzwerk
- bei virtuellen Maschinen: Keine phys. Geräte, viele Maschinen mit identem Umfeld
=> generieren gleiche Zufallszahlen?

Physikalischer Zufall (3)

Besser:

Zusätzlich *echter Zufallsgenerator*
in Hardware!

Basiert auf

“weißem bzw. thermischem Rauschen”
von Halbleitern u.ä., d.h. digitalisiertem
zufälligem elektrischem Analogsignal

(oder: Radioaktiver Zerfall, elektromagnet. Hintergrund-Rauschen, ...)

==> Schneller / besser,
aber auch begrenzte Datenrate

Zufallszahl berechnen (1)

Im einfachsten Fall:

- Phys. Inputs auf Qualität prüfen,
Quelle bei zu wenig Zufall verwerfen!
- Mehrere phys. Quellen zusammenhängen
- Kryptografischen Hash davon berechnen

Zufallszahl berechnen (2)

Für mehr Zufallszahlen als die Physik liefert
(z.B. bei Linux `/dev/random` und `/dev/urandom`):

- Pseudo-Zufallszahl-Generator mit *langer Periode*
(z.B. Mersenne Twister usw.)
- Initialisierung und regelmäßige Re-Initialisierung
(in etwa nach Größe Ausgabe = Größe interner Status)
mit Hardware-basierter Zufallszahl
- Ev. mehrere Ausgabe-Zahlen nochmals hashen,
damit interner Status nicht rückrechenbar ist

Kryptographische Generatoren (1)

Alternative z.B.

BBS-Generator

(Blum-Blum-Shub-Generator)

Beruhet auf Produkt n von 2 geheimen, großen
(hunderte Stellen!) Primzahlen (frisch berechnet)
und mod-Rechnung

Unbrechbarkeit hängt u.a. ab von
der Schwierigkeit der Faktorisierung

Kryptographische Generatoren (2)

Außerdem:

Man nutzt nicht alle Bits des internen Status für die Ausgabezahl, sondern nur eines oder ein paar

==> Aus den gelieferten Zufallszahlen ist

- der interne Status

- der Startwert

- die große Zahl n

nicht praktisch berechenbar

==> Ohne Kenntnis von n ist

Vorwärts-Rechnen nicht möglich!

Zufallszahlen-Fails (1)

Entweder:

- ***Ergebnismenge*** ist viel *kleiner als erwartet*,
d.h. *nur wenige Zahlen* kommen als Ergebnis vor
==> *Durchprobieren* geht *schneller* als erwartet

Oder:

- Berechnen der ***vorigen / nächsten Zahl*** oder
des internen Status ist viel *einfacher als erwartet*
==> Aus der Kenntnis einer Zahl
sind *alle weiteren vorhersagbar*

Zufallszahlen-Fails (2)

1.) NSA hatte heimlich geschwächten Generator “***Dual_EC_DRBG***” standardisieren lassen

==> Für bestimmte Parameter:

- Schlecht verteilte Zufallszahlen
- Nächste Zahl voraus berechenbar

2.) Debian baute versehentlich einen **Bug im KeyGen** von **OpenSSL** ein: **Nur 2^{15} Zufallszahlen!**

==> Viele Millionen leicht knackbare Keys weltweit für SSH, OpenVPN, DNSSEC, X.509 Zertifikate, ...

3.) Der **MIFARE Hack** (schlechter Random Seed)

Passwort-Speicherung (1)

- Wenn Klartext-Passwort nicht gebraucht wird (sollte der Normalfall sein!):

Einweg-Verschlüsselung = Hash

(==> wenn Passwort-Datei geklaut:
Passwords nicht rückrechenbar!)

- Hash-Algorithmus darf & muss langsam sein!!!

==> Hash ev. einige tausend Mal rechnen!

(Argon2, **scrypt**, **bcrypt**, (PBKDF2))

==> verzögert Brute-Force-Attacken

und vor allem Berechnung von Rainbow Tables

Passwort-Speicherung (2)

- Beim Hashen: ***“Salzen”!!!***

Salz = Zufallszahl

wird

- bei jedem Ändern eines Passworts neu erzeugt
- mitgehasht und im Klartext mitgespeichert
- beim Prüfen des Passworts auch mitgehasht

Wenn n Möglichkeiten für's Salz:

Rainbow Tables werden
 n Mal so groß & n Mal so langsam!

Passwort “online” knacken

Online = durch Anmelde-Versuche

“Brute force” = viele Passwörter durchprobieren

==> “Brute-Force-Bremsen” einbauen!!!

- 1.) Wartezeit nach jedem Fehlversuch*
- 2.) Account-Sperre nach x Fehlversuchen*
- 3.) Fehlermeldung darf nicht zwischen “User falsch” und “Passwort falsch” unterscheiden*

==> Quadriert die Anzahl der Versuche bei unbekanntem User!

Passwort “offline” knacken (1)

Offline = gestohlene Passwort-Datei liegt vor

Grundprinzip:

- Zu testende Passwort-Kandidaten hashen
- Prüfen, ob Hash in der Datei vorkommt

“Passwort-Kandidaten”: “Alle” dauert viel zu lange!

==> Aus “Wörterbuch”!

- Normale Wörter, umgedrehte Wörter,
normale Wörter in Leetspeak, ...
- Häufig benutzte Passwörter,
“tippfreundliche” Zeichenkombinationen, ...
- *Zuerst kurz, dann immer länger!*

Passwort “offline” knacken (2)

Beschleunigung:

Hashes nicht jedesmal frisch berechnen & prüfen, sondern Hashwerte der Passwort-Datei in fertigen, vorberechneten Tabellen mit Klartextpasswort & Hashwert suchen!

- 1.) Tabelle nur mit Wörterbuch-Passwörtern
- 2.) Tabelle aller möglichen Zeichenkombinationen
... wäre bei direkter Speicherung viel zu groß!
(viele TB oder PB)

Passwort “offline” knacken (3)

“Rainbow Table” = *Spezielle Datenstruktur dafür*

(kleiner, effizient durchsuchbar)

Praktikabel für einfache Hash-Algorithmen (MD4/5)
ohne Salz, kurze Passwörter (max. 10 Zeichen).

Gegenmaßnahmen:

==> **Salzen**

==> **Lange Passwörter** erzwingen

==> **Guten** (= aufwändigen) **Hash** verwenden

Passwort setzen (1)

- ***Mindestlänge*** fordern!

Empfohlen: Min. 12 Zeichen, Obergrenze min. 64

- ***Großen Zeichensatz*** (Sonderzeichen) erlauben, Groß- und Kleinschreibung unterscheiden (!!!), Ziffern & Sonderzeichen fordern

Bei 12 Zeichen:

- Nur Kleinbuchstaben und Ziffern: $36^{12} = 4,7 \cdot 10^{18}$
- Alle sichtbaren ASCII-Zeichen: $95^{12} = 5,4 \cdot 10^{23}$

==> Über 100000 Mal so viele Möglichkeiten!

Passwort setzen (2)

Ziel: Schutz gegen Wörterbuch-Attacken

Daher:

Schon beim Eingeben

gegen Wörterbücher prüfen!

(Passwort-Wörterbücher, nicht normale)

Weiters:

Prüfung gegen Passwort = Userid usw.

Passwort setzen (3)

Pflicht zur regelmäßigen Passwort-Änderung ist
eher konterproduktiv!

Hilft nur gegen heimlich geklaute
Klartext-Passwörter!

==> Technisch: Für Knackbarkeit irrelevant!

==> Praktisch: Benutzer

- verwenden deshalb eher einfachere Passwörter
- notieren sich das Passwort eher
- ändern Teile des Passworts systematisch
("Jan", "Feb", ... oder fortl. Nummer)

Passwort setzen (4)

Größte Gefahr:

**Selbes Passwort
für viele Dienste, Webseiten, ...**

Eine davon wird gehackt

**==> selber Benutzer
auf allen anderen Systemen kompromittiert!
... ohne dass es gleich auffällt / bekannt wird!**

Gegenmaßnahme:

Dienstanweisung, Schulung, ...

Passwort setzen (5)

Mögliche Gegenmaßnahme für hochsichere Systeme:

Nur Passwörter von Passwort-Generator erlauben!

Müssen gut merkbar sein
(werden sonst aufgeschrieben!)

z.B. Statt kurzem, zufälligen “Zeichensalat”:
Lange Passphrasen aus “Silben” und Sonderzeichen!

Ev. mehrere Passwörter zur Wahl!

Achtung auf ausreichend Möglichkeiten &
nach außen unbekannte Generator-Logik!

Programmierung (1)

Im Systemlog protokollieren:

Alle Passwort-Änderungen

Alle erfolgreichen & fehlerhaften Anmeldungen

Ev. auch bei jedem Login anzeigen:

“*Letzte Anmeldung am ...*”

Passwort-Änderung:

Neues Passwort muss
doppelt eingegeben werden
(gegen Tippfehler)

Programmierung (2)

Systempasswörter:

Es darf kein Betrieb ohne Passwort oder mit Default-Passwort möglich sein!

*==> Regelbetrieb blockieren,
bis Systempasswort geändert wurde!*

Benutzer-Passwort:

Administrator-gesetztes Passwort

- muss bei jedem Benutzer anders sein
- muss sofort bei erster Anmeldung geändert werden

Programmierung (3)

- Jede String-Variable, die ein Passwort enthielt, sofort mit gleichlangem String überschreiben (nicht auf Leerstring setzen, nicht einfach freigeben!)
- Variablen mit Klartext-Passwörtern in nicht-swapbaren Speicherbereich legen
- Programme mit Klartext-Passwörtern
 - dürfen nicht debug-bar sein
 - dürfen keine Core dumps schreiben

100 % Schutz nicht möglich!
(z.B. Ausführung in VM?!)

Mehr-Faktor-Authentifizierung

- Etwas, das man **hat**:
Handy, Chipkarte, Key-Token, ...
iTAN-Karte
biometrische Eigenschaften
- Etwas, das man **weiß**:
Passwort oder Passphrase
Sicherheitsabfrage, ...
- Etwas, das sich jedesmal **ändert**:
Zeit, zufällige Challenge

Authentifizierung über's Netz (1)

Niemals ein Klartext-Passwort
über das Netz übertragen
(wenn wirklich kritisch:
auch nicht SSL-verschlüsselt!)

Und auch nie einen ***jedesmal gleichen*** Hash!

Weil: ***Replay-Attacken!***

(= Netz-Daten abfangen & damit nochmals anmelden)

=> Hash schon am Client rechnen!

=> Selbe Message vom Client
darf nicht zwei Mal gültig sein!

Authentifizierung über's Netz (2)

1.) Listenbasierte One-Time-Passwörter:

z.B. TAN-Listen

2.) Hashbasierte One-Time-Passwörter:

***i -ter Passcode =
 $n-i$ Mal gehashter Geheimcode***

==> Aus dem i -ten Passcode
ist der vorige berechenbar,
aber nicht der nächste!

Authentifizierung über's Netz (3)

Der Server

- speichert den zuletzt verwendeten Passcode
(uninteressant, 1 Mal mehr gehasht als das aktuelle)
- hasht den empfangenen Passcode ein Mal
==> muss gleich dem gespeicherten Passcode sein!

Der Client

- hasht das eingegebene Passwort
in der richtigen Anzahl (?)
- oder hat eine geheime, vorberechnete Liste

Authentifizierung über's Netz (4)

3.) Security Token (z.B. RSA SecurID)

zeigt einzugebenden Passcode an
(oder überträgt ihn via USB, ...)

“Einweg-Fall”: Passcode wird berechnet aus:

- *Fix eingebranntem, geheimen, individuellen **Key***
- **Zeit** (==> jedesmal anderer Passcode!)
- Ev. eingetipptem Geheimcode

oder “Challenge-Response-Verfahren”

Challenge-Response-Verfahren

- Server schickt **“Challenge”**
(bei Tokens: Übertragung z.B. optisch, manuell, Funk, ...)
z.B. Zufallszahl & Zeitstempel gehasht
(==> jedesmal anders)
- Client oder Token berechnet den **“Response”**
aus Challenge & Passwort und/oder Key
(==> auch jedesmal anders,
Passwort darf daraus nicht rückrechenbar sein!)
- Server kann an Hand des Response prüfen,
 - ob Passwort bzw. Key am Client stimmen
 - ob die Antwort zu genau dieser Challenge passt

Zero Knowledge Protocol (1)

*“Beweise, dass Du einen Schlüssel besitzt,
ohne den Schlüssel preiszugeben”*

- A besitzt geheimen Schlüssel
- B will prüfen, ob A diesen Schlüssel besitzt, darf aber den Schlüssel dabei nicht erfahren und auch nicht berechnen können
- A darf nicht vortäuschen können (durch “trickreiche” Antworten an B), dass er den Schlüssel kennt, obwohl er ihn gar nicht kennt

Zero Knowledge Protocol (2)

1.) Vorbereitung:

- Große Zahl n als Produkt zweier Primzahlen berechnen und veröffentlichen (nur das n , nicht die Primzahlen)
- A wählt zufälligen privaten Schlüssel s , berechnet öffentlichen Schlüssel v aus n und s , veröffentlicht v

2.) Prüfung (in mehreren Runden):

- A und B tauschen pro Runde 3 Zahlen aus, die sie in jeder Runde neu aus einer Zufallszahl berechnen
- B prüft mit einer Rechnung die ausgetauschten Zahlen gegen die öffentlichen n und v

Zero Knowledge Protocol (3)

Mit jeder erfolgreichen Prüfungsrunde
verdoppelt sich die Sicherheit

mit der B annehmen kann,

dass A wirklich das s zu n und v besitzt,

ohne dass B (oder ein außenstehender Zuhörer)
daraus dieses s berechnen könnte!

Steganographie (1)

Wörtlich:

“Verdeckt schreiben”
(oder *“geheim schreiben”*)

De facto:

Verstecken von Daten in anderen Daten
(zum Speichern oder zum Transport),
ohne dass es auffällt
(= Tarnung, nicht Verschlüsselung)

Steganographie (2)

Die “*offensichtlichen*” Daten

- sind *unverschlüsselt*
- sehen *harmlos & unverdächtig* aus
- stehen in *keinem inhaltlichen Zusammenhang* zu den eigentlichen “Nutzdaten”
- lassen *nicht auf den ersten Blick erkennen*, dass darin andere Daten versteckt wurden
- sind aber im Normalfall *unauffällig verfälscht* (meist nur *bei Kenntnis des Originals* feststellbar)

Steganographie heute (1)

Die “offensichtlichen” Daten sind im Normalfall

Bilddateien (Fotos) oder *Tondateien*

Eine Veränderung der

1 oder 2 hintersten (niederwertigsten) Bits

in Farb- oder Tondaten fällt kaum auf

=> Pro Farbwert oder Ton-Sample

können *1-2 Bits “Nutzdaten”*

“verpackt” werden!

Steganographie heute (2)

In der Praxis:

*“Nutzdaten” trotzdem
vor dem Verpacken verschlüsseln!*

- Verschlüsselung liefert
zufällige, unauffällige, gleichverteilte Bits
- ==> Manipulation der Trägerdaten
fällt weniger auf
- Nutzdaten bleiben geschützt,
auch wenn Steganographie entdeckt wird

Steganographie früher (1)

Statt Nutzdaten nachträglich
in vorgegebene Originaldaten
zu “verpacken”:

Nachträglich extra
rund um vorgegebene Nutzdaten
eine unauffällige
“Verpackung” konstruieren!

Steganographie früher (2)

In der Praxis:

*“Harmlose” Texte rund um
vorgegebene Wörter
oder Buchstaben erfinden.*

Beispiele für einfache Codes:

- Der 2. Buchstabe jedes Wortes zählt
- Tabelle mit mehreren möglichen Wörtern
für jeden Buchstaben

Auch wesentlich komplizierter!

Sonderfall MIC

“Machine Identification Code”

Z.B. bei Laserdruckern in sensiblen Organisationen:

Laser druckt außer dem gewünschten Ausdruck z.B. einzelne “verirrte” Pixel an ganz bestimmten Stellen (bei Farblasern meist gelb, fällt am wenigsten auf) oder “verliert” einzelne Pixel an Buchstabenkanten

Lage dieser Pixel codiert

- **Druckernummer** &
 - **Datum+Uhrzeit** des Ausdrucks
- ==> Ausdruck rückverfolgbar!

Sonderfall Wasserzeichen

Zweck:

*Kodiert unauffällig **Copyright / Besitzer**
oder **eindeutige Kopien-Nummer**
in Video-, Bild- oder Tondateien*

Anforderung:

- Wenig Daten
- Aber hohe Robustheit gegen
 - Umcodierung, Kompression, Ausschnitte, ...
 - Wiedergabe und Re-Digitalisierung

Disk Encryption (1)

Ist (außer dem User-Interface zur Konfiguration) normalerweise keine Anwendung, sondern sitzt

- *im Betriebssystem*
- *als Treiber*
- unmittelbar über den Disk-I/O-Hardware-Treibern weit unterhalb der Filesystem-Logik

==> Installation erfordert normalerweise
Admin-Rechte

Disk Encryption (2)

- Verschlüsselt jeden einzelnen Disk-Block unmittelbar bevor er zur Platte geschickt wird
 - Entschlüsselt jeden einzelnen Disk-Block unmittelbar nachdem er von der Platte kommt
- ==> Die Verschlüsselung **“sieht” nur Disk-Blöcke** (ohne Kenntnis ihrer Bedeutung), hat keine Ahnung von Files, Filesystemen und File-Metadaten
- ==> ... ist Filesystem-unabhängig
- ==> ... verschlüsselt File-Inhalte und File-Metadaten, aber auch Swap-Spaces, Suspend-Partitionen usw.

Disk Encryption (3)

Arbeitet *transparent* und “*on-the-fly*”
(d.h. man merkt davon in der laufenden Arbeit nichts):

Sobald das Disk-Passwort o.ä. eingegeben ist
(meist ein Mal beim Booten oder beim Login),

sind alle Daten
auf dem verschlüsselten Datenträger

normal (=unverschlüsselt)
sichtbar, lesbar und schreibbar

(grundsätzlich für alle Benutzer & Dienste)

Unterschied File Encryption

File Encryption

- verschlüsselt *nur die File-Inhalte*, aber nicht die File-Metadaten
- wird *explizit* zum Ver- oder Entschlüsseln einzelner Dateien / Verzeichnisse *aufgerufen*
- ist meist eine *normale Anwendung*, keine Betriebssystem-Komponente, erfordert keine besonderen Rechte
- erlaubt *verschiedene Passwörter* pro Benutzer / pro Datei oder Verzeichnis

Zweck (1)

Schutz vor

- Auslesen der Daten bei ***HW-Diebstahl***
(vor allem bei Laptops usw.)
- Auslesen der Daten durch ***Booten***
eines ***fremden Betriebssystems***
(z.B. Linux von einem USB-Stick)
- Verwendung der Datenträger
nach ***Einbau in andere Rechner***
(bei *Hardware-Bindung durch TPM*)

Zweck (2)

Schützt nicht vor Zugriff auf alle Daten

- *in einer angemeldeten Sitzung*
(z.B. “Kollege während der Kaffeepause”)
- bzw. durch am Rechner laufene *Malware*,
via *Fileshares* über's Netz usw.

Für diese Fälle:

File-Encryption verwenden!

Angriffe (1)

- **RAM** enthält im laufenden Betrieb ständig den Key zur Entschlüsselung
=> Zugriff auf das RAM über **ext. Schnittstellen**, die eigenständig RAM-Zugriffe machen können (Thunderbolt, Firewire, ...)
- **RAM** hält die Daten nach dem Abschalten für **einige Minuten** (je nach Temperatur, ...)
=> Schneller Boot mit Diagnose-OS, Memory Dump
- **RAM** wird bei Suspend in den **Hibernate-File** geschrieben... (Disk-PW auch???)

Angriffe (2)

- Disks bleiben *im Sleep-Mode entsperrt*, eingegebenes Passwort bleibt *erhalten*
=> *Bei Laptop-Diebstahl im Sleep-Mode sind die Daten zugreifbar!!!*
- Bei Platten-Hardware-Verschlüsselung: Passwort gilt, *solange Strom da ist*
=> *Ohne Unterbrechung der Stromversorgung mit anderem Betriebssystem booten oder Platten mit anderem Rechner verbinden*

Backup

- ***File-basiertes Backup*** einer verschlüsselten Disk:
Files werden im Klartext am Backup gespeichert!

Gegenmaßnahmen:

- *File-Encryption nutzen*: Files mit File-Encryption werden verschlüsselt am Backup gespeichert
- *Backup separat verschlüsseln*
(Bandlaufwerk mit Hardware-Verschlüsselung)
- ***Image- bzw. Block-basiertes Backup***
ist *je nach Software* verschlüsselt oder nicht!

Recovery

Beim Konfigurieren der Verschlüsselung
und Festlegen eines Passwords:

Recovery-Key oder Recovery-File
werden angezeigt

Zweck: Daten-Rückgewinnung

- bei ***Verlust des Passwords***
- bei ***Defekt des TPM***

Offline sicher aufbewahren!

(Papier & Tresor oder USB-Stick & Tresor)

Datenträger (1)

Was kann eine Disk Encryption verschlüsseln?

1.) ***“Physical Volume” = Komplette Platte***

(incl. Boot Sector, Partitionstabelle, ...
und allen Partitionen)

Die ganze Platte enthält dann *“zufälligen Bitsalat”*

==> Sieht *“unbenutzt”* bzw. *“bewusst gelöscht”*
(mit Zufallszahlen überschrieben) aus

Datenträger (2)

2.) ***“Logical Volume” = Ein Filesystem***

- Eine Partition
- Ein logisches RAID-Laufwerk
- Eine Volume (auch über mehrere Platten)
in einem Logical Volume Manager

Ohne Boot-Sektor bzw. Partitionstabelle

3.) ***Container (“virtual drive”) = Ein eigenständiges Filesystem innerhalb eines großen Files***

(in Linux: “Loop mount”)

Datenträger (3)

4.) *“Hidden Volume”*:

Eigenständiges Filesystem wie ein Container, aber nicht in einem File gespeichert, sondern

*“unsichtbar” in den unbenutzten Blöcken
eines anderen Filesystems
 (“Outer Volume”,
meist ebenfalls verschlüsselt)*

==> Kann im Kontrollfall *“abgestritten”*
bzw. verheimlicht werden:
Nur “Outer Volume” herzeigen!

Datenträger (4)

Zu 2. auf Linux:

Alle Benutzer-Daten
stehen unter **/home/username/...**

=> Häufig: Entweder **/home**
oder jedes **/home/xxx** einzeln
auf eigene Partition legen und verschlüsseln

System-Partition wird oft gar nicht verschlüsselt
=> *Normaler, einfacher Boot-Vorgang möglich*

Login-Password entsperrt zugleich **Home-Partition**
=> Benutzer-Daten erst nach Login sichtbar

HW-Verschlüsselung (1)

“Self encrypting drives”

Es gibt Platten- und Bandlaufwerke
mit eingebauter Hardware-Verschlüsselung
(OPAL-Standard, verwendet AES 128 oder 256)

... oder “Verschlüsselungs-Zwischenstecker”
für die Datenleitung zum Laufwerk

=> Ver- und Entschlüsseln selbsttätig
ohne Performance-Verlust

HW-Verschlüsselung (2)

- Key wird im Laufwerk zufällig erzeugt, **verlässt das Laufwerk nie!**
- Key ist auf dem Laufwerk mit einem Passwort verschlüsselt gespeichert
- “Freischalten” des Keys bzw. des Datenzugriffs durch Übermittlung des Passwords beim Booten
- “**Quick delete**”-Feature:

Key im Laufwerk **löschen**
=> Alle Daten am Laufwerk
dauerhaft nicht mehr lesbar

Key-Eingabe (1)

- Meist beim Booten: Passwort- oder Pin-Eingabe, Smartcard bzw. USB-Stick, Biometrie

*Bei kompletter Verschlüsselung
der System-Platte:*

“Pre-Boot”-Key-Eingabe nötig
(derzeit noch kaum umgesetzt)

- Wenn nur Hardware-Bindung gewünscht:
Nur TPM, ohne Authentifizierung
- Wenn nur Diebstahlschutz gewünscht:
Nur ***Erreichbarkeit*** eines firmeninternen ***Servers***

Key-Eingabe (2)

De facto bei verschlüsselter System-Partition meist:

- **Windows, Mac:**

BIOS bootet Minimal-System nur für PW-Abfrage aus versteckter, unverschlüsselter Partition

Dieses bootet nach Passwort-Eingabe das “echte” System in der verschlüsselten Partition

Key-Eingabe (3)

- **Linux:**

Normales Booten des unverschlüsselten Kernels
mit unverschlüsseltem Bootloader
und unverschlüsselter initrd

Passwort-Abfrage im initrd-Modus,
danach Umschaltung auf
“echtes” verschlüsseltes Root-Filesystem

Bei Verwendung von TPM:

***TPM erkennt jede Veränderung
an den unverschlüsselten Teilen!***

Produkte

BitLocker: Bestandteil von Windows

dm-crypt: Bestandteil von Linux und BSD

(basierend auf LUKS =

Offener Standard für verschlüsselte Filesysteme)

FileVault: Bestandteil von macOS

VeraCrypt (früher TrueCrypt):

Open-Source-Software für Win, Mac, Linux, ...

- Sehr gründliches Security-Audit
- Sehr viele Krypto-Algorithmen zur Wahl
- Kann “Hidden Volumes”

Verschlüsselung (1)

- ***Symmetrisch***, meist ***AES*** 128 oder 256, ev. Twofish, Serpent, ...
- ***Key*** wird meist ***zufällig erzeugt*** und mit Zugangscode verschlüsselt
- Die einzelnen ***Disk-Blöcke*** (512 B bzw. 4 KB) werden ***unabhängig voneinander*** verschlüsselt (*ohne Verkettung* der Verschlüsselung)

Verschlüsselung (2)

Innerhalb eines Disk-Blocks werden die AES-Blöcke miteinander verkettet, damit sich wiederholende Klartext-Bitmuster unterschiedliche Verschlüsselungen haben

- Früher: Normales **CBC** (Cipher-Block Chaining)
- Dann: **LRW** (Liskow, Rivest, Wagner)
- Heute meist: **XTS**
(extra für Disk-Verschlüsselung entwickelt)

Verschlüsselung (3)

Problem dabei:

*Welcher IV (Initialisierungsvektor)
wird für jeden Disk-Block verwendet?*

- **Fixer IV: Ganz schlecht**
(gleicher Klartext-Disk-Block
=> Gleicher verschlüsselter Disk-Block)
- **IV = Sektornummer des Disk-Blocks**
Etwas besser, aber Teil-Angriffe möglich
(z.B. feststellen, ob eine Datei mit präpariertem Inhalt verschlüsselt gespeichert wurde oder nicht)

Verschlüsselung (4)

- **Jedesmal neuer, zufälliger IV:**
Sicherer, aber aufwändig:
IV für jeden Disk-Block muss
separat verschlüsselt gespeichert werden
- **IV = verschlüsselte Sektor-Nummer:**
Heute *bestes Verfahren*, z.B. in XTS
Verwendet einen Key *doppelter Länge*:
 - Eine Hälfte für
die *eigentliche Daten-Verschlüsselung*
 - Eine Hälfte für
die *Verschlüsselung der Sektor-Nummer* für IV

TPM (1)

TPM = “Trusted Platform Module”

Auf den ersten Blick: Ähnlich Smartcard usw.

- ***Generiert und speichert Schlüsselpaare***
- ***Verschlüsselt und entschlüsselt “on Chip”***
(Schlüssel braucht TPM nicht verlassen),
prüft Schlüssel
- Berechnet sichere ***Zufallszahlen*** und ***Hashes***
- ***Signiert***

RSA, ECC, AES, SHA1, SHA256, HMAC, ...

TPM (2)

Auf den zweiten Blick:

“Trusted System”
“Tampering Protection”

- Berechnet, speichert und verifiziert
Prüfsummen von Software & Hardware

Offiziell: Zur Software-Sicherheit

Lückenlos abgesicherte Boot-Kette
von HW & BIOS bis zur Anwendung

==> PC bootet nur in “sicherem” (?) Zustand

TPM (3)

Weniger:

Der Benutzer kann seinem System vertrauen

Vor allem:

Dritte (Rechte-Inhaber)

können dem System vertrauen

==> TPM schützt die Software vor dem Benutzer?!

Dazu:

- TPM verwaltet **weltweit eindeutige ID's**
(pro Gerät und pro User)

TPM (4)

- ==> **“Zugesperrte” PC’s**,
lückenlose Kontrolle der Software
wie bei Smartphones, Spielkonsolen, ...
(“Schutz” vor Knack-Software)
- ==> **“Hardware-gesicherte” Lizenz- & DRM-Prüfung**
- ==> individuelle, fixe **Bindung**
von Lizenzen & DRM-Inhalten
an ein bestimmtes Gerät
- ==> Bindung an bestimmte Software-Anbieter
(nur Anbieter, die SW gültig signieren können)

Von wem?

TPM ist hersteller-unabhängiger Standard
verwaltet von der

TCG = “Trusted Computing Group”

= Viele Hersteller von HW & SW
(Intel, Microsoft, ...)

+ alle großen PC-Hersteller
(HP, Dell, Lenovo, Acer, Asus, ...)

Hardware

TPM = Eigener Prozessor + RAM + ROM + EEPROM

angebunden z.B. über LPC- oder SPI-Bus
als steckbares Modul
(oder im Chipset integriert)

==> TPM ist kein Busmaster,
kann das System nicht von sich aus ansprechen

==> TPM wird nur aktiv,
wenn es von der Haupt-CPU abgefragt
bzw. angestoßen wird!

Software (1)

CRTM = “Core Root of Trust for Management”

Kurz: *“Trusted Root”* oder *“Authenticated Root”*

In BIOS oder eigenem Chip realisiert,
muss *von sich aus sicher & vertrauenswürdig* sein

Vor dem eigentlichen Booten, vor dem TPM:

- Initialisiert TPM
- Schickt Prüfsummen von HW & BIOS an das TPM
- Fragt Passworteingabe, Smartcard o.ä. ab und schickt sie an das TPM

Software (2)

***TSS = “Trusted Software Stack”
oder “TCG Software Stack”***

In Betriebssystem, Libraries usw.:

***Treiber, Dienste und
Software-Schnittstellen (API's)
für den Zugriff von Anwendungen
auf das TPM***

Noch kaum vollständige Implementierungen!

Status

Große Diskussion vor 10-15 Jahren

==> TPM wurde großteils skeptisch aufgenommen bzw. abgelehnt

- Wegen Angst vor DRM und SW-Monopolisierung
- Wegen Politik der TCG:
 - Geheime Arbeit, geheime Dokumente
 - Hohe Kosten für Mitgliedsbeiträge, Lizenzen und Zertifizierungen (nur für Große)

Seitdem ist es sehr still um TPM geworden

(Open-Source-TPM-Code ruht de facto seit 2006)

Angriffe

Auf die TPM-Hardware:

Wenige, nur spezifisch für bestimmte Module:

- ***Seitenkanal-Attacken***
(meist über Stromverbrauch)
- Ein Hersteller erzeugte
knackbare RSA-Schlüsselpaare

==> HW gilt als relativ sicher und ausgereift

SW ist Stückwerk in zweifelhaftem Zustand

Der EK (1)

EK = “Endorsement Key”

Hart eingebrannt, **nicht** auslesbar

Codiert Echtheit des TPM & Hersteller des TPM &
weltweit eindeutige ID

= Basis zur Prüfung der TPM-Vertrauenswürdigkeit

Endorsement-Zertifikat zum EK + Plattform-Zertifikat

+ Conformance-Zertifikat + Validation-Zertifikate

(alle Zertifikats-Schlüssel im TPM hinterlegt)

= “gesamtes System (HW + SW) entspricht der TCG-Spezifikation”

= Basis für alle weiteren Keys

Der EK (2)

Prüfung der EK-basierten Vertrauenswürdigkeit eines Systems mit TPM erfolgt

nicht durch den Dienste-, Lizenz- bzw. DRM-Inhaber

(er darf nur “seinen” AIK erfahren, aber nicht die mit dem EK verbundene eindeutige Identität!)

sondern

- indirekt über unabhängige CA's
- oder mittels Zero-Knowledge-Protokoll

Netz-Verbindung erforderlich!

Der EK (3)

Problem:

- TPM-Hersteller lässt sich aus dem TPM ermitteln
- ***TPM-Hersteller kennt den privaten EK***

=> Der TPM-Hersteller könnte

- ... das System tracken
- ... Keys im TPM knacken / Keys fälschen
- ... den EK an Agencies usw.. weitergeben

Daher TPM-Feature ***“Neuen EK erzeugen”***

Problem: Neuer EK ist nicht vertrauenswürdig,
nicht zertifiziert => weitgehend nutzlos!

Der SRK

SRK = “Storage Root Key”

= *“Wurzel” des gesamten Schlüssel-Baumes*

wird bei Erstbenutzung des TPM generiert aus
EK + Passwort des Users (oder Bio-Daten, ...)

einmalig, Neu-Generierung oder User-Wechsel
ist nicht möglich

Die PCR's (1)

PCR = "Platform Configuration Register"

(mehrere, mindestens 24)

= im TPM sicher berechnete und gespeicherte

Hashwerte über

- ***Hardware*** (z.B. Platten-Seriennummern)

- ***Firmware*** (BIOS, MBR, ...)

- ***Betriebssystem***

- ***Software & Konfig-Dateien***

kann *in ALK-Berechnung mit einbezogen* werden

Die PCR's (2)

Jedes PCR ist nach dem Booten leer & wird ein- oder mehrmals um Hashes erweitert

In die jede Hash-Berechnung eines PCR's geht sein voriger Wert mit ein

=> Nachträglich nicht fälschbare Kette von Hash-Werten

Jeder Schritt vom Boot-Prozess bis zum Anwendungsstart verlängert die PCR-Ketten um die Prüfsummen des als nächstes zu startenden Schrittes

Die AIK's (1)

AIK = “Attestation Identity Key”

Im TPM erzeugte und permanent gespeicherte Keys zum SW-Schutz, für Lizenz- und DRM-Zwecke, ...

Für jeden Dienste-, Lizenz- oder DRM-Anbieter wird ein neuer AIK erstellt

==> Jeder Anbieter “sieht” einen anderen Key

Grund: ***Schutz der Anonymität***

Eigene, unabhängige Identitäten für jeden Anbieter sollen Anbieter-übergreifendes Tracking und Rückschluss auf den konkreten PC vermeiden.

Die AIK's (2)

Erstellung von AIK's, z.B. bei Lizenz-Erwerb:

Mittels Challenge-Response-Protokoll aus

- **Challenge** & Daten des Anbieters
(z.B. *individueller Lizenz-Key*)
- Der **EK**
=> Bindung an ein *bestimmtes Gerät* / TPM
- **Signierte Hashes von PCR's** nach Wahl
=> Bindung an *bestimmten HW- & SW-Zustand*,
Erkennung von HW- und SW-Änderungen

Die AIK's (3)

Prüfung von AIK's bei jedem Start der SW, ...:

Ebenfalls mit Challenge-Response-Protokoll,
u.a. mit Vergleich mit aktuellen PCR-Werten

==> Für eine AIK-Prüfung müssen Netzverbindung
und Server des Anbieters verfügbar sein

Bei jeder AIK-Erstellung und AIK-Prüfung:

Validierung der Vertrauenswürdigkeit des TPM
(Prüfung von EK-Zertifikat usw. durch CA)