Kryptographie

Klaus Kusche

Inhalt (1)

Wörtlich:

"Kryptographie" =
"Die Lehre von der Verschlüsselung"

Hier etwas weiter gefasst:

- **Verschlüsselungsverfahren**
- **Anwendung: Disk Encryption**
- **Hashes**
- … (nächste Seite)

Inhalt (2)

- **Kryptographische Zufallszahlen**
- **Passwörter**
- **Signaturen & Zertifikate**
- **Anwendung: DKIM usw., PGP, De-Mail**
- **Anwendung: Blockchain (z.B. Bitcoin)**
- **TPM (Trusted Platform Module)**
- ... (nächste Seite)

Inhalt (3)

- **Steganographie**
- **Typische Sicherheitslücken, sichere Programmierung** (separate Folien)

Ziel

- **Überblick und Verständnis**, z.B. von
  - Verschlüsselungsverfahren
  - deren Zielen, Merkmalen, 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**
- **Netzwerk-Verschlüsselung usw.**
  (SSL/TLS, IPsec, Kerberos, ...)
- **IT-Sicherheit**
  (Schutz vor Angriffen, Einbrüchen, Datenverlust, ...)

  ==> *Beides sollte anderes Fach sein...*

Voraussetzungen

- **Grundkenntnisse:**

  - Informatik,
    im Sicherheitslücke-Teil
    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** (u.a. auch Hash)

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, nss,

libgcrypt, NaCl / libsodium, linux kernel, ...)

Einteilung Krypto-Algorithmen

- ***Symmetrische*** Verfahren
- ***Asymmetrische*** Verfahren

Unterscheiden sich *wesentlich* in

- ***Idee*** / Konzept
- Verwaltung der ***Schlüssel***
- Art der ***Rechenoperationen***
  ==> Implementierung, Geschwindigkeit, ...
- ***Anwendungsbereichen!***

Oft *beide in Kombination!*

Symmetrische Verfahren (1)

Verwenden

- ***denselben Schlüssel***
- manchmal sogar *denselben Algorithmus*

zum *Verschlüsseln und Entschlüssseln*

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, GB/s in HW!

Symmetrische Verfahren (3)

Daher:

"Massendaten"

= Dokumente, Backups, ganze Platten

= die eigentlichen Nutzdaten
bei Netzwerk-Übertragungen

werden immer

symmetrisch verschlüsselt!

Symmetrische Verfahren (4)

Beispiele:

DES (zu kurzer Schlüssel: 56 Bits!)

Triple-DES

IDEA (unfrei)

AES "Advanced Encryption Standard" = Rijndael

(Rijndael hatte von allen AES-Kandidaten die geringste 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 (5)

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)

Man kann

- *beide* Schlüssel *gemeinsam* erzeugen
- bei manchen Verfahren auch den öffentlichen Schlüssel zu einem bekannten privaten Schlüssel berechnen

aber ***niemals***

(mit vertretbarem Aufwand)

aus dem öffentlichen Schlüssel den privaten Schlüssel berechnen!

Asymmetrische Verfahren (3)

- Der *Empfänger* berechnet *einmal*
  sein ***persönliches Schlüsselpaar***

  (für *alle zukünftigen Verschlüsselungen* zu ihm)
- ***veröffentlicht*** den *öffentlichen Schlüssel*
- hält den *privaten Schlüssel* ***geheim (!!!)***

- Der *Sender* einer Nachricht
  bekommt oder holt sich den

  öffentlichen Schlüssel des Empfängers

Asymmetrische Verfahren (4)

- Der Sender ***verschlüsselt*** die Nachricht mit dem ***öffentlichen*** *Schlüssel des* ***Empfängers***
- Die Nachricht kann nur

mit dem dazugehörigen privaten Schlüssel entschlüsselt werden

==> *Jeder kann verschlüsseln,*
nur der "richtige" Empfänger kann entschlüsseln!

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

Asymmetrische Verf.: Technik (1)

- Arbeiten ebenfalls *blockweise*,
  aber mit *größeren* Block- und Schlüssellängen:

  2048-4096 Bits (512 Bit oder 1024 Bit sind zu wenig!)

- Beruhen auf ***komplexen mathematischen Ideen,***
  arbeiten intern
  - mit entsprechend ***langen ganzen Zahlen***
    (2048 Bits ... Dezimalzahl mit über *600 Ziffern*!)
  - primär mit ***Multiplikation und Restrechnung***
- *Unterschiedliche* Algorithmen zur Ver- und Entschlüsselung

Asymmetrische Verf.: Technik (2)

==> Viel *aufwändigere Grund-Operationen*

==> Keine typischen / effizienten Prozessor-Befehle, *schlecht* durch Hardware oder Vektor-Befehle zu *beschleunigen*.

==> Mindestens 1000 mal *langsamer*!

***Nicht** für die effiziente*
Ver- und Entschlüsselung
großer Datenmengen geeignet!

Asymmetrische Verf.: Mathe (1)

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 Verf.: Mathe (2)

Beispiel 1: *Faktorisierung großer Zahlen*

Wenn *p* und *q* *zwei sehr große* ***Primzahlen*** sind (einige hundert Stellen),

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

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

Asymmetrische Verf.: Mathe (3)

Beispiel 2: *"Diskreter Logarithmus"*

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

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 Verf.: Mathe (4)

Ähnliche Idee: ***ECC = "Elliptische Kurven"***

Im wesentlichen ähnlich dem *diskreten Logarithmus*,
aber *nicht* mit *ganzen Zahlen*, sondern mit
Punkten auf Kurven über endlichen Körpern!

Die einzelnen 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***" (DJB) gilt als bestes ECC-Verfahren

Hybride Verfahren

In der Praxis:

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

- Zuerst ein *asymmetrisches Verfahren*
  zum ***Austausch des symmetrischen Schlüssels***,

  z.B. *Diffie-Hellman-Schlüsseltausch*

  Der sym. Schlüssel ist *kurz* ==> darf langsam sein!
- Dann ein *symmetrisches Verfahren*
  zum *schnellen* Austausch der ***Nutzdaten***

*SA vs *DH (1)

Unterschied RSA / DSA / ECDSA und DH / ECDH:

- ***SA** sind universelle *Verschlüsselungsverfahren*
- ***DH** sind spezielle Algorithmen
  ausschließlich zur Berechnung
  und zum sicheren Austausch
  eines symmetrischen Schlüssels:

  "Schlüsselaustauschverfahren"

  (primär auf *Netzwerk-Verbindungen*)

*SA vs *DH (2)

DH verwendet zusätzlich ***Zufallszahlen***

==> DH bietet ***"Forward secrecy"***, RSA nicht:

- Bei Knacken/Stehlen des *privaten asymm. Keys*:
  - Aufgezeichnete verschlüsselte *Daten*
  - Symmetrische *Keys*

  alter (schon *beendeter*) *Verbindungen*
  sind ***nicht nachträglich knackbar***
- Bei Knacken/Stehlen des *symmetrischen Keys einer einzelnen* Verbindung
  ==> ***Andere*** *Verbindungen sind damit nicht knackbar*

Krypto-Analyse

Angriffe durch ***"Krypto-Analyse"***:

- (==> *Klartext* herausfinden)
- ==> *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 Chiffrat:

"Pseudo-zufälliger Output"

- Keine erkennbarer Zusammenhang
  Bestimmte Bits im Input oder im Schlüssel <==>
  Bestimmte Bits im Output
- Keine Beschreibung des Verfahrens durch ein *algebraisches Gleichungssystem*

==> Verhindert u.a. ***statistische Analysen***
(z.B. Zeichen-Häufigkeit)

Anforderungen (2)

Beständigkeit gegen ***"Known Plaintext"-Attacken***:

Klartext und dessen Chiffrat bekannt
(oder Chiffrat und Teile des Klartextes bekannt),

==> Schlüssel gesucht

Extremfall:

Krypto-Box in den 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 Chiffrat?

==> Anforderung ***"Lawinen-Effekt" = "Diffusion"***:

Kleine Änderungen im Klartext
bewirken *große Änderungen im Chiffrat*
(Ein einziges Bit im Input ändern
==> Jedes Bit im Output ändert sich
mit 50 % Wahrscheinlichkeit)

Anforderungen (4)

Beständigkeit gegen ***"Seitenkanal-Attacken"***

- *Timing*- und *Stromverbrauchs*-Analyse, "Hot Spots" auf der CPU, ...
- 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 ==> 64 Bit
- ... *faktorisieren/logarithmieren* in *polynomialer Zeit*:
  *~L*L*log(L)*log(log(L))* Schritte bei Zahl mit *L* Ziffern
  (Faktorisierung bisher:
  Superpolynomiale, aber subexponentielle Zeit: *~ 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)

Viele Qualitäts-Kriterien, teils *widersprüchlich*:

- *Erkennen* von Block-*Vertauschungen*, *Bitfehlern* oder *Manipulationen*
  (u.a.: Ändern eines Bits im Chiffrat darf nicht genau das entsprechende Bit im Klartext ändern)
- *Wiederaufsetzen* nach kaputten Bits oder fehlenden Blöcken
- *Gleiche* Blöcke im Input ==> *Verschiedene* Blöcke im Output?
  (Sicherheits-Kriterium!)
- *Byteweise* statt blockweise Verschlüsselung möglich?
  (z.B. Tastatur-Input!)
- *Parallelisierbar, Voraus-berechenbar?*
  (schneller, weniger Latenz)

Verschlüsselung langer Daten (3)

Die Verschlüsselung der einzelnen Blöcke wird meist

irgendwie verkettet

d.h. *interne Daten* (meist in der Größe eines Blockes)
werden zur Verschlüsselung des *nächsten* Blockes *weitergereicht*

Womit *beginnt* die Kette,
d.h. welche Verkettungs-Daten gehen in den ersten Block?

"Initialisierungsvektor" IV

- entweder ***doppelt langer Schlüssel***
  (eine Hälfte ist der eigentliche Schlüssel, die andere der IV)
- oder ***Zufallszahl***

Verschlüsselung langer Daten (4)

ECB ("Electronic Code Book Mode"):

- Datenstrom in Blöcke teilen
- Jeden Block *separat* verschlüsseln

Nachteile:

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

Vorteil:

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

Verschlüsselung langer Daten (5)

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 (6)

CFB ("Cipher Feedback Mode")

Verkettung ähnlich CBC:

Xor mit Plaintext
nach der Verschlüsselung
des vorigen Chiffrats

Selten!

Verschlüsselung langer Daten (7)

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 (8)

Verschlüsselung und Entschlüsselung sind *ident*:

- Der Empfänger erzeugt auf dieselbe Art (mit *derselben* Zufallszahl) ***denselben Bitstrom***
- ... und berechnet das *Xor von Chiffrat und Bitstrom*

  ==> Liefert den Klartext!

Bei *Bitfehlern* in der Übertragung:
Nur *genau entsprechende Bits*
der entschlüsselten Daten betroffen!
(kann Vorteil oder Nachteil sein: ***Verfälschbar!!!***)

Verschlüsselung langer Daten (9)

Die Erzeugung des Bitstroms
(= der aufwändige *Verschlüsselungs-Algorithmus*) ist

- *parallelisierbar*
  (*jeder Block ist für sich* berechenbar)
- *vorausrechenbar*
  (die Verschlüsselungsberechnung
  hängt *nicht vom Klartext* ab)

Echte ***Stromchiffre***, kann *Byte für Byte* arbeiten!
(Byte für Byte Xor von Klartext mit Bitstrom)

Verschlüsselung langer Daten (10)

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 (11)

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, ...

Auth-Tag = Prüfsumme = kryptografischer Hash:

==> ***fälschungssichere** Fehlererkennung!*

***"GMAC"**: Nur* Auth-Tag-Berechnung, unverschlüsselt

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*

==> Die *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
- Arbeitet ***transparent*** und ***"on-the-fly"***
  (d.h. man merkt davon in der laufenden Arbeit nichts)

Disk Encryption (3)

- Die Verschlüsselung ***"sieht" nur Disk-Blöcke*** *(ohne Kenntnis ihrer Bedeutung)*, hat *keine Ahnung* von *Files, Filesystemen und File-Metadaten*
- ... ist daher *Filesystem-unabhängig*
- ... *verschlüsselt* ***File-Inhalte und File-Metadaten***, aber auch *Swap-Spaces*, Suspend-Partitionen usw.

- Die Verschlüsselung weiß ***nicht***, ***wer*** (welches Programm / welcher Benutzer) die Daten liest oder schreibt!

Disk Encryption (4)

Konsequenz:

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

(für *alle* gerade aktiven Benutzer & Dienste!!!)

Unterschied Disk-/File- Encryption

File Encryption

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

Zweck der Disk-Encryption

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*)

Nicht Zweck der Disk-Encryption

Schützt nicht vor Zugriff auf alle Daten

- ***in einer angemeldeten Sitzung***
  (z.B. *"Kollege während der Kaffeepause"*)
  oder durch Remote-Benutzer
- 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, ... (sollte inzwischen gefixt sein)

- ***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*** oft ***entsperrt***, eingegebenes Passwort bleibt ***erhalten***

  ==> *Bei Laptop-Diebstahl im Sleep-Mode*
  sind die Daten zugreifbar!!!

- Bei Platten-Verschlüsselung in ***Hardware***
  (direkt im Platten-Laufwerk / in der SSD):
  Das 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!

Datenträger (1)

Welche "Platten"
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** als Ganzes
oder jedes **/home/*username*** einzeln
auf *eigene Partition* legen und *verschlüsseln*

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

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

Passwort-Eingabe (1)

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

Bei kompletter Verschlüsselung der System-Platte:
***"Pre-Boot"-Passwort-Eingabe** nötig*
(derzeit noch kaum umgesetzt)

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

Passwort-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

Passwort-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*

  (unter Linux stehen Kernel & Initrd
  in vielen Fällen *nicht* in der System-Partition, sondern
  entweder in einer eigenen, unverschlüsselten *Boot-Partition*
  oder als UEFI-Exe in der *UEFI-Partition*)

Passwort-Eingabe (4)

Bei vollständiger Implementierung
und Verwendung von **TPM**
(*"Secure Boot"* in Maximalversion):

TPM ***erkennt*** *jede* ***Veränderung***
an den ***unverschlüsselten Teilen***
vor dem Booten bzw. ***bis zur PW-Abfrage!***

Verschlüsselung (1)

Symmetrisch, meist

- ***AES*** 128 oder 256 (in kommerziellen Produkten)
- ***Twofish*** oder ***Serpent*** o.ä. (bei Open Source)

Problem:

Key dafür muss 128 oder 256 Bits lang sein!

==> Key direkt als Passwort wäre *unpraktisch*,
schlecht einzugeben (32 oder 64 Hex-Ziffern)!

Entweder: Key ist ***gehashtes Passwort***

Verschlüsselung (2)

Oder:

Der eigentliche ***Key*** für die Verschlüsselung
wird meist ***zufällig erzeugt***
und ***mit dem Passwort verschlüsselt***

Nur der verschlüsselte Key wird gespeichert!
(der *Klartext-Key* wird als *Recovery Key* für den Tresor ausgegeben
und dann im Rechner / auf der Platte *gelöscht*)

==> *Zugriff* auf die Daten nur nach Entschlüsselung
des *verschlüsselten Keys* mit dem *Passwort*

Key weg *oder* Passwort weg ==> *kein Zugriff*!

Recovery

Beim *Konfigurieren* der Verschlüsselung
und *Festlegen eines Passwords*:

Recovery-Key oder Recovery-File
(mit dem unverschlüsselten Key) werden angezeigt

Zweck: Daten-Rückgewinnung

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

Offline sicher aufbewahren!
(Papier & Tresor oder USB-Stick & Tresor)

Verschlüsselung (3)

- Die einzelnen ***Disk-Blöcke*** (512 Bytes bzw. 4 KB) werden ***unabhängig voneinander*** verschlüsselt (*ohne Verkettung* der Verschlüsselung zwischen Blöcken)
- ***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 (4)

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 (5)

- **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 den IV

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)

- Der Key wird *im Laufwerk zufällig erzeugt*,

  verlässt das Laufwerk nie!

  (weder verschlüsselt noch unverschlüsselt)
- Der Key ist im Laufwerk
  nicht im Klartext gespeichert, sondern

  mit einem Passwort verschlüsselt
- *"Freischalten"* des Keys bzw. des Datenzugriffs
  durch Übermittlung des Passwords *beim Booten*

HW-Verschlüsselung (3)

"Quick delete"-Feature:

Beim "Quick Delete"-Befehl wird

*der verschlüsselte **Key***
*im Laufwerk **gelöscht***

==> Alle Daten am Laufwerk sind
dauerhaft nicht mehr lesbar
(nicht mehr entschlüsselbar)

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"*

Hash, Anwendungen (1)

Bei Dateien, Mails, Downloads, ...:

- Nur ***Unverfälschtheit***:

  Hash (*kein* Schlüssel o.ä.)

- *Unverfälschtheit*

  & *Authentizität* & *Vertrauenswürdigkeit*

  & "*Nicht-Abstreitbarkeit*" des Absenders

  Hash

  + *Signatur* *(beruht auch auf Hashes)*

  + *Zertifikats-Infrastruktur*

Hash: Anwendungen (2)

*Das gesamte **Signatur- & Zertifikats-System***
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?)

Hash: Anwendungen (3)

Zur ***Reduktion des Krypto-Rechenaufwandes***
beim Signieren & Prüfen von Signaturen:

Signaturen sind
asymmetrische Verschlüsselungs-Verfahren
==> Daher sehr *rechen-aufwändig*!

Es wird ***nur der Hash***
der Daten signiert,
nicht die Original-Daten
(der Hash ist viel *kürzer*)

Hash, Anwendungen (4)

Hashing & Passwörter:

- Hashes dienen zur ***"Einweg-Verschlüsselung"*** von ***Passwörtern***
- ... und für ***One Time Passwords***
- Hashes stecken in ***Challenge-Response-Verfahren***

  zur ***sicheren Passwort-Abfrage über's Netz*** (gegen ***"Replay-Attacken"***)

Hash: Anwendungen (5)

- Einen Hash durch Probieren aller Möglichkeiten ***"rückwärts"*** zu rechnen
  (d.h. Daten zu finden, die einen bestimmten Hashwert haben)
  ist ***sehr aufwändig***

  ==› Dient als ***"Rechenaufgabe"***
  beim ***Blockchain-Mining***

- Hashing liefert wie eine Verschlüsselung ***"Pseudo-zufällige"*** Daten

  ==› Zum ***"Mischen, Verbessern und Strecken"***
  von ***echten (physikalischen) Zufallszahlen***

Hash (1)

Hash = "Prüfsumme"
("fingerprint")

Bei der Erzeugung und Prüfung
eines Hashwertes ist

kein Passwort oder Schlüssel beteiligt

==> ***Jeder*** kann den Hash
eines Dokumentes, Passwords usw.

erstellen und prüfen!

Hash (2)

Der Hashwert ist normalerweise

kürzer als die ursprünglichen Daten

(z.B. fix 256 Bits),

Hashing ist daher ***keine bijektive Funktion***

==> *Informationsverlust* (meist sogar *gewünscht*):

Die Original-Nachricht ist garantiert

nicht aus dem Hash wiederherstellbar!

==> *Verschiedene* Original-Nachrichten können

gleichen Hash haben *(ganz selten, schlecht!)*

Kryptographischer Hash

Unterschied zur Hashfunktion z.B. bei Hashtables:
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 ein *Hashwert*
- Oder eine Datei/Nachricht A und deren Hashwert

Gesucht:

Neue Nachricht B, die genau diesen Hash ergibt

==> Nachricht/Datei A kann durch B *ersetzt* werden, ohne dass es auffällt

==> Wenn das der Hash des echten Passwortes A 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 mit *gleichem* Hash 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 (z.B. in Grafiken)

Kollision Fall 2

Gegeben:
Oft: Gewünsche *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 du **A verschickt** hast*

Oft: 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"** lösbar ist!*

(d.h. wenn Kollisionen intelligenter
als durch simples Durchprobieren
erzeugt / gefunden werden können)

(Fall 2 ist meist leichter als Fall 1)

Hash-Verfahren (1)

... sind vom Code her ähnlich
wie ***symmetrische*** *Verschlüsselungen:*

Bitoperationen (Xor, Shift, ..)
auf großen Bit-Blöcken fixer Länge

==> ***schnell***

==> ***mit HW-Unterstützung*** (z.B. SSE / AVX)
oder rein in HW mit speziellen Schaltungen
implementierbar

Hash-Verfahren (2)

- ***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, aber Hack-Komplexität nur <= 2^64)

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

  Längere Varianten davon, noch sicher

- ***SHA-3*** (seit 2015, 224 - 512 Bits): *Empfohlen!*

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 = Von **2^15** bis **2^(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*:

*Liefert **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:

- Physikalische Inputs *auf Qualität prüfen*, die 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* (spätestens nach Größe Ausgabe = Größe interner Status) mit *Hardware-basierter Zufallszahl*
- Mehrere Ausgabe-Zahlen *nochmals hashen*, damit interner Status nicht rückrechenbar ist

Kryptographische Generatoren (1)

Alternative z.B.

BBS-Generator
(Blum-Blum-Shub-Generator)

Beruht auf Produkt *n* von 2 geheimen, *großen* (hunderte Stellen!) *Primzahlen* und mod-Rechnung

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

Aber wie jeder "gerechnete" Generator:

***Periodisch**, nur endlich viele "Zufalls"-Zahlen!*

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

  Einweg-Verschlüsselung = Hash

  (==› wenn die Passwort-Datei geklaut wurde:
  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 die *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!***

==> Kandidaten aus einem ***“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: $36^{12} = 4{,}7 * 10^{18}$
  - *Alle* sichtbaren ASCII-Zeichen: $94^{12} = 4{,}7 * 10^{23}$

  ==> ***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 Coredumps* 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 bzw. regelmäßig ***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 Passcode* 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

4.) *"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!

Signaturen und Zertifikate (1)

Siehe Hash: Garantieren gemeinsam

- ***Unverfälschtheit*** *der Nachricht*
- ***Authentizität***
- ***Vertrauenswürdigkeit***
- *"**Nicht-Abstreitbarkeit**"* des Absenders

Signaturen und Zertifikate (2)

Signatur:

Garantiert, dass die Nachricht / Datei
vom ***Besitzer eines bestimmten***
geheimen Schlüssels erstellt wurde

(aber sagt nichts über dessen reale Identität)

==> *Primär technisches / mathematisches Problem*

==> ***Jeder***, der den geheimen Schlüssel ***besitzt***
(bzw. geklaut hat), kann sich

unerkennbar als echter Ersteller ausgeben!

Signaturen und Zertifikate (3)

Zertifikat:

Verbindet einen bestimmten
geheimen Schlüssel
mit der realen ***realen Identität***
des rechtmäßigen Besitzers

==> *Primär organisatorisches Problem*

==> *Hilft aber nicht gegen Schlüsselklau*

==> *Nutzt intern wieder an vielen Stellen Signaturen*

Signatur (1)

Der Sender / Ersteller

- *Berechnet den Hashwert* der zu signierenden Daten
- *Verschlüsselt* den Hash mit seinem *Private Key*
- Verschickt die *Originaldaten plus Signatur*
  (die Signatur enthält den verschlüsselten Hashwert)

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

Signatur (2)

Der Empfänger

- *Entschlüsselt* den Hash in der Signatur mit dem dazugehörigen *Public Key*
- *Berechnet den Hash* der empfangenen Daten
- *Vergleicht* die beiden Hashwerte

==> *Jeder* kann die Signatur *prüfen*!

Signatur (3)

Der entschlüsselte Hash aus der Signatur
und der 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 den Daten: *Zu einer beliebigen Signatur wären "dazupassende" Daten berechenbar, ohne den Private Key zu kennen!*
- *Hoher Rechenaufwand* bei langen Daten

Signatur (4)

	Verschlüsselung	Signatur
Senden (Verschlüsseln / Signieren)	***Öffentlicher*** *Schlüssel des* ***Empfängers***	***Privater*** *Schlüssel des* ***Senders***
Empfangen (Entschlüsseln / Signatur prüfen)	***Privater*** *Schlüssel des* ***Empfängers***	***Öffentlicher*** *Schlüssel des* ***Senders***

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*

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

*Auch die **CA darf keine Kopie des Private Key** haben!*

Voraussetzungen Signatur (3)

Zu 2.):

PKI = "Public Key Infrastructure"

Ein *technisches und organisatorisches System* zur

- Sicherstellung der ***Vertrauenswürdigkeit*** von Public Keys
- ... und ihrer ***Zuordnung*** zu Personen, Firmen, Domains, ...
- ... und zur 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*** *des Zertifikats*
- ***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:

Eine übergeordnete CA *zertifiziert den Key der CA*

==> Für jedes Zertifikat muss gelten:

Lückenlose Kette von CA's und ihren Zertifikaten

bis zu einer ***"Root CA"***

(die Root CA hat ein *"self-signed-Zertifikat"*
= signiert ihr Zertifikat selbst mit dem *eigenen Key*)

PKI (4)

Weltweit gibt es *ein paar hundert* Root CA's

==> ***Explizite Liste***
von Root CA's und deren Zertifikaten
fix *im Betriebssystem / im 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 *unverfälscht*
- 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 die *Korrektheit* der Daten im Zertifikat,
  Identität und Berechtigung des Beantragers!!!
- CA erzeugt ein neues *Schlüsselpaar*
- CA stellt das *Zertifikat für den Public Key* aus,
  übermittelt den Private Key geheim an den Besteller
- CA führt eine *Liste* aller von ihr ausgestellten Zertifikate (Beantrager, öffentl. Schlüssel, ...)
- CA verwaltet eine *Sperrliste ihrer Zertifikate*

Sperren von Zertifikaten (1)

Wenn:
- *Private Key* geklaut
- Daten des Beantragers *stimmen nicht* mehr
- *Kriminelle Aktivitäten* des Beantragers oder der CA

==> 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)

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 die 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, Karten usw. ausstellen
  ==> Hoher organisatorischer *Aufwand*, *dauert*!
- Behebung bei *signierten Dokumenten*: ***Keine!!!***
  (weil die Signatur *nicht nachträglich änderbar* ist!)

Probleme CA-System (2)

Angriffspunkt lokale Root-CA-Liste:

Dort selbstgebastelten *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 Virenscanner*, 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 Beziehungen zwischen Teilnehmern:

- ***X kennt Y*** (persönlich)
  ==> *X signiert den 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: ***Schlüsselserver***

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 / Client
  ==> 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)

DKIM usw. (1)

Standardisierter grundlegender E-Mail-Schutz:

SPF & DKIM & DMARC

Verschlüsselung: *nein*

Signatur und Authentizität: *Ja, aber* ...

- Nicht ab Client, *erst ab sendendem Server*
- **Nicht** *personenbezogen*,
  sondern nur *Domain-bezogen*

Hauptziel ist **nicht** *"Mail sicherer machen"*,
sondern *"Spam vermeiden / erkennen / verwerfen"*

DKIM usw. (2)

SPF = "Sender Policy Framework"

Eintrag im DNS der *Domain des Senders* als **`txt`**-Record:

*"**Welche Mailserver** dürfen Mail*
*mit einer **"From"-Adresse aus dieser Domain***
versenden?" *(Whitelist, nicht Blacklist)*

==> Erkennbar,

- wenn eine Mail ein *gefälschtes "From"* hat
- wenn eine Mail von einem für den Absender *"unzuständigen" Mailserver* kommt

DKIM usw. (3)

DKIM = "Domain Keys Identified Mail"

Der ***sendende Mailserver***:

- ***Berechnet Hashes*** des *Mail Bodies*
  und *ausgewählter Header-Zeilen*,
  die sich beim Transport *nicht ändern* dürfen

- ***Signiert*** die Hashes mit dem *private Key der Domain*

- ***Codiert*** Hashes und Signatur *Base64* als ASCII
  (weil Mail nicht 8-bit-transparent ist)

- Fügt das als ***neue Header-Zeile*** oben in die Mail ein:
  "**`DKIM-Signature:`** …"

DKIM usw. (4)

Technisch: ***RSA + SHA-256***

*Der **Public Key der Domain** für DKIM*
steht als **`txt`***-Record*
*im **DNS der Sender-Domain***

==> *Jeder Zwischen-Server* einer Mail und
der *empfangende Server / Client* können ***prüfen***

- ob die Mail *vom Server dieser Domain* kommt
- und ob der Body und die wesentlichen Header
("From:", "Subject:", "Date:", "To:", "Reply-To:", ...)
unverfälscht sind

DKIM usw. (5)

DMARC = "Domain-based Message Authentication, Reporting and Conformance"

Noch ein **`txt`**-Record im *DNS der Absende-Domain*:

"Ist SPF und/oder DKIM
für Mails von dieser Domain Pflicht?"

"Wie ist auf fehlendes oder falsches
SPF bzw. DKIM zu reagieren?"

Möglichkeiten u.a.:
- Mail *verwerfen*?
- *Fehlermeldung* an den Postmaster senden?

DKIM usw. (6)

- Noch ein paar *offene konzeptionelle Fragen*,
  z.B. bei der Umsetzung für *Mailing Lists*:

  Der sendende Server einer Mailing-List-Mail
  ist unabhängig und verschieden
  von der "From:"-Domain der Mail!

- Die *praktische Umsetzung* hängt von *Providern* ab:

  - Bei **SPF und DKIM** relativ **hoch**
    (u.a. weil von Google usw.
    für jede einlangende Mail gefordert!)
  - Bei **DMARC** noch eher **dürftig** ...

PGP (1)

"Pretty Good Privacy"

Standard: **RFC4880**

Implementierungen (frei) u.a.:

- **OpenPGP**
- **GPG** = "Gnu Privacy Guard"

Primärer Einsatz:

Mail-Verschlüsselung & -Authentizität

Z.B. auch:

Absicherung automatischer SW-Verteilung

PGP (2)

Kombination **symmetrischer & asymmetrischer Verfahren**:

- Falls gewünscht: Fügt eine *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 Elgamal ist lizenz- und patentfrei, arbeitet ähnlich Diffie-Hellman-Verfahren)
- *Base64-Kodierung* für die gesamte 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) nur optional
- Zusätzliche *Sende- und Empfangs-Nachweise*
  (wie *"Einschreiben"*), vom Dienstanbieter signiert

Organisatorisch:

- Nur von *zertifizierten Anbietern*
- Durch *Ausweisprüfung* identifizierte *Benutzerkonten*

De-Mail (2)

Sicherheit & Verschlüsselung:

- 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*, dafür *zentraler Key-Server*)
- *Hashwert für Integrität*, optional: *Signatur*
  (beides am *ersten / letzten Server*, *nicht* am Client!)

De-Mail (3)

Neben ***fehlender End-to-End-Verschlüsselung,***
fehlender End-to-End-Signatur und
sehr ***isoliertem, eingeschränktem Benutzerkreis***
(deutsche Sonderlösung, nicht EU-Behörden-kompatibel)

sind viele ***Datenschutz-Aspekte offen***:

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

==> Alles andere als "vertraulich" oder "geheim"!

De-Mail (4)

Trotz Start schon 2012:

==> *Fehlende Akzeptanz bei Benutzern und IT*
(einige 1000 Mails pro Jahr...)

- Ursprünglich *verpflichtende Benutzung* in Bundesbehörden wurde *2024 beendet*
- Die meisten *zugelassenen Anbieter* haben 2022-2025 mangels wirtschaftlicher Perspektiven ihren *Dienst eingestellt*
- ***Offizielles Betriebsende Ende 2026***

==> Soll durch eine EU-Lösung ersetzt werden?

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 nachdem eine gewisse Anzahl von Blöcken dahinter erzeugt ist)

- Jeder Block enthält *zusätzlich* ein *paar Bits*, die durch "*Mining*" bestimmt werden

Blockchain (4)

==> Bis jetzt wäre alles *einfach!*

Bei Krypto-Währungen usw.:

- Die 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 3,125
- ***Transaktionsgebühren***
  der damit bestätigten Transaktionen (weniger)

Mining-Belohnungen sind der einzige 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: Die **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!

TPM (1)

TPM = "Trusted Platform Module"

Auf den *ersten* Blick: Ähnlich Smartcard usw.

- ***Generiert und speichert Schlüsselpaare*** *(weltweit eindeutige!)*, prüft Schlüssel
- ***Verschlüsselt und entschlüsselt*** "on Chip" (Schlüssel braucht TPM nicht verlassen)
- 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 & Keys*** (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
(zumindest in Version 1.x, nicht zwingend bei 2.x)

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 (1)

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)

Status (2)

Nach "Abkühlung" der Diskussion

- ***2014*** Standardisierung von ***TPM 2.0***
  (ganz heimlich, still & leise)
- Erstmals Implementierung *direkt im Prozessor*
  (ein bisschen HW, vor allem ***Firmware***): ***"fTPM"***
- Seit Windows 11 *Pflicht*

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 AIK-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***)
- ***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)

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