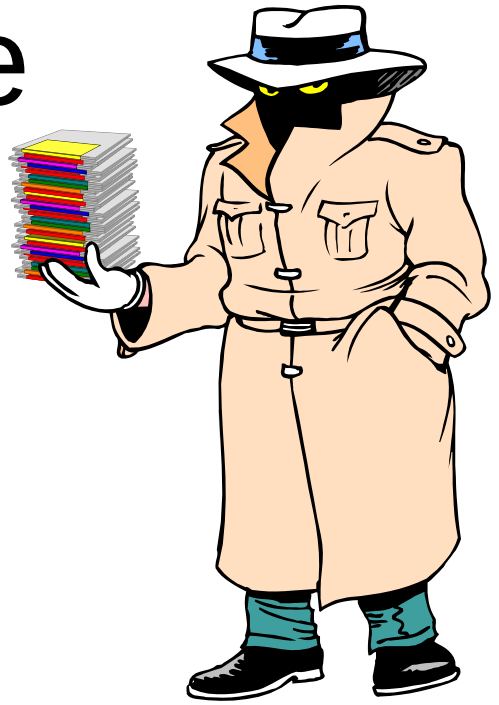
Verteilte Systeme

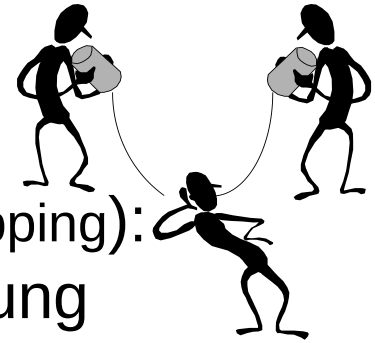


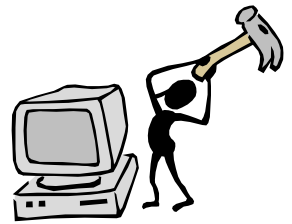
Sicherheit

„It is easy to run a secure computer system. You merely have to disconnect all dial-up connections and permit only direct-wired terminals, put the machine and its terminals in a shielded room, and post a guard at the door.“

Problem Sicherheit

- Das Senden von Daten von einem zu einem anderen Computer ist **immer ein Risiko**.
- **Gefahren**:
  - **Mithören** (*Schnüffeln* Sniffing/*Lauschen* Eavesdropping): Versuch, ohne die entsprechende Berechtigung Nachrichten mitzuhören
  - Vorgabe **falscher Identitäten** (*Parodieren* Spoofing/*Maskieren* Masquerading): Senden und Empfangen von Nachrichten unter einer anderen Identität (ohne die Erlaubnis dieser Identität)
  - **Unterbrechen**: Ein Teil des Systems, d.h. des gesamten Informationskanals, wird zerstört oder unbrauchbar.





Problem Sicherheit

- **Änderung** von Nachrichten (*Verfälschen* Tampering): Abfangen von Nachrichten und Veränderung ihres Inhalts, bevor sie an den eigentlichen Empfänger weitergegeben werden (schwierig in Broadcast-Netzen, leicht bei Store-and-Forward)
- **Wiederholung** von Nachrichten (*Wiederholung* Replay): abgefangene Nachrichten werden abgespeichert und zu einem späteren Zeitpunkt erneut gesendet
- **Verweigerung** von Diensten (*Ablehnung von Diensten* Denial of Service): Eingeschleuste Komponenten verweigern die Dienstleistung, oder durch Überfluten eine Dienstverweigerung bewirken

Definitionen von Sicherheit

- **Funktionssicherheit** (*safety*)
  - Übereinstimmung der realen Ist-Funktionalität eines Systems mit der spezifizierten Soll-Funktionalität
  - *Korrektheit und Zuverlässigkeit des Systems*

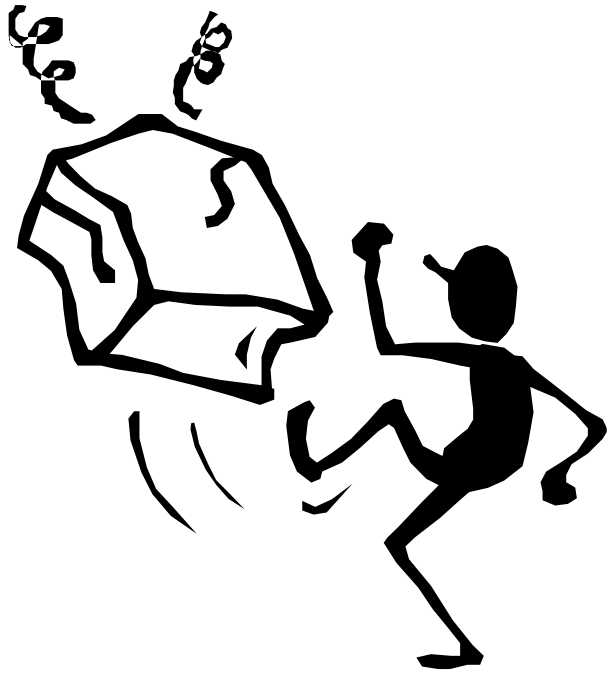
- **Datensicherheit** (*protection*)
  - Eigenschaft eines funktionssicheren Systems, nur solche Zustände anzunehmen, die zu keinem unautorisierten Zugriff auf Daten oder andere Systemressourcen oder zum Verlust von Daten führen.
  - *Schutz der Daten (u.a. durch Datensicherungsmaßnahmen)*

- **Informationssicherheit** (*security*)
  - Eigenschaft eines funktionssicheren Systems, nur solche Zustände anzunehmen, die zu keiner unautorisierten Informationsveränderung oder Informationsgewinnung führen
  - *Schutz der Informationen*

Ziele von Sicherungsmaßnahmen

- **Vertraulichkeit** (Confidentiality)
  Schutz der Informationen vor unautorisierter Einsichtnahme (Geheimhaltung!)
- **Unversehrtheit** (Integrity)
  Schutz der Daten vor unautorisierter Veränderung (Verhindern von Modifikation oder Löschung!)
- **Authentizität** (Authenticity)
  Die Daten wurden wirklich von der Person gesendet, die behauptet, der Sender zu sein.
- **Verantwortlichkeit** (Responsibility)
  Jede sicherheitsrelevante Aktion im System kann eindeutig einem Urheber zugeordnet werden.
- **Verfügbarkeit** (Availability)
  Schutz des Systems vor (beabsichtigter) Störung - Verhindern von Abstürzen oder Performanceverlusten!
- **Einbruchssicherheit** (Intrusion Protection)
  Schutz der Endsysteme vor unautorisiertem Zugang
- **Gefährdet durch** Konzeptionsfehler, Programmierfehler, Konfigurationsfehler

Angriffe + Verteidigungen



Angriffe

- **Angriff:**
  Ein nicht autorisierter Zugriff bzw. Zugriffsversuch auf ein IT-System
- **Passiver Angriff:**
  Zugriff auf vertrauliche Informationen (→ Verlust der Vertraulichkeit)
  Beispiele: Abhören von Leitungen, Lesen von geheimen Daten
- **Aktiver Angriff:**
  Modifikation von Datenobjekten oder Systemressourcen (→ Verlust der Integrität / Verfügbarkeit)
  Beispiele: Verändern / Löschen von Dateien oder IP-Paketen, Überschwemmen mit TCP-Verbindungsanfragen („Denial-of-Service“)

Angreifer-Typen

Bezeichnung	Charakterisierung	Ziele	Motive
Hacker („White Hats“)	Sicherheitsfachleute	auf Schwachstellen („Exploits“) aufmerksam machen	Wissens-erwerb
Cracker („Black Hats“)	Technisch versiert, kriminelle Energie	• Diebstahl von Geld (Kreditkartennummern, Dialer, …) oder Informationen • Ruhm in der Szene • Zerstörungslust	• Bereich-erung • Eitelkeit • Bosheit
Skriptkids	jugendlich, technisch unbedarft, nutzt im Internet veröffentlichte Schwachstellen und Tools	• Ruhm in der Szene • Spiellust • Faszination	• Eitelkeit • Neugier

Angreifer-Typen

Bezeichnung	Charakterisierung	Ziele	Motive
Geheim-dienste	Technisch versierte Geheimdienst-mitarbeiterInnen	• Wirtschaftspionage • Militärische Spionage • Terrorbekämpfung	• Wirtschaftliche Vorteile für Firmen • „Nationale Sicherheit“
Interne Mitarbeiter-Innen	Personen mit internen Kenntnissen und Zugriffsrechten	• Sabotage • Sammeln interner Informationen • Wirtschaftsspionage für Konkurrenzfirmen	• Frust und Wut • Neugier • Bereicherung

Funktionsweise von Angriffen

- Für einen Angriff, muß ein **Zugang zu dem System** bestehen.
- **Meist** über die **Kommunikationskanäle** des verteilten Systems.

- In den meisten Fällen werden **Angriffe von rechtmäßigen Benutzern** gestartet, die ihre Autorität mißbrauchen.

- **Nicht-zugangsberechtigte Angreifer** müssen Methoden wie das Raten oder Knacken von Passwörtern einsetzen.
- Außer diesen direkten Formen des Angriffs werden Programme eingesetzt, die das System von außen **infiltrieren**. (Passwort knacken, Virus, Wurm, ...)

Beispiel: Angriffstaktik eines Cracker-Angriffs

- Angriffsziel festlegen und **Informationen sammeln**
- Erstzugriff durch Ausnutzen von Schwachstellen
  z.B. Erzeugen eines Pufferüberlaufs, Maskierung, …
- Ausbau der Zugriffsberechtigungen
  z.B. Knacken von Passwortdateien, Ausnutzen von Vertrauensbeziehungen
- Spuren verwischen
  z.B. Manipulation von Protokolldateien, Verstecken von Dateien
- Hintertür offen lassen
  z.B. Manipulation der Startdateien

Beispiel: Buffer Overflow

- **Problem**:
  - Nachlässige Programmierung
  - Unsichere Programmiersprache (meist C)
  - ➔ Unzureichende Längenprüfung / Absicherung von Eingabedaten
- **Angriffstechnik**:
  - Durch Eingabedaten mit Überlänge (→ lokale Variablen, Parameter) werden Teile des Stacks überschrieben
  - Überschreiben der echten Rücksprungadresse
  - Platzieren von eigenem Assemblercode auf dem Stack oder einer gefälschten „Rücksprungadresse“ mit Aufruf einer Bibliotheksprozedur (LoadLibrary, Shell, ..)!

Beispiel: Buffer Overflow

```
cmd = lies_aus_netz();

do_something(cmd);

.....

int do_something(char* InputString) {
      char buffer[4];
      strcpy (buffer, InputString);
      ...
      return 0;
}
```

`strcpy` kopiert ohne Prüfung solange in den Speicher, bis NULL gelesen wird!!!

Beispiele: Angriffe aus dem Netzwerk

- **TCP SYN Flooding**
  Verweigerung von Diensten durch Erzeugung vieler halboffener TCP-Verbindungen
- **IP Spoofing**
  Einbruch in bestehende Verbindungen durch Vorgabe falscher Identitäten (IP Absenderadresse)
- **DNS Spoofing**
  Einpflanzung einer falschen IP-Adressauflösung zur Servicevortäuschung
- **Ping Flooding (SMURF-Attacke)**
  Verweigerung von Diensten durch echo-requests nach IP-Spoofing
- **D**istributed **D**enial **o**f **S**ervice – **DDoS**
  Überfluten des Opfers durch Pakete von sehr vielen Rechnern – gepaart mit Würmern oder regulären Kommunikationsdiensten (Botnets)

Häufigste Angriffe (Schnappschuss)

- Angriffe auf Integrität
  - Cross-site scripting, Cross-site request forgeries,
  - Bsp: Stuxnet, Hackers Gamers Brazil
- Angriffe auf Verfügbarkeit
  - Amplification DDoS attacks (botnets, DNS, server-side scripting)
  - Bsp: SpamHouse, Operation Ababil, Operation 'Semana de Pagamento‘ (Brazil)
- Angriffe auf Vertraulichkeit
  - Kryptographische Brüche $\rightarrow$ RC4, TLS
  - Routing Redirections

Das Schutzsystem von Java

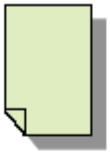
Java ist eine sichere Sprache (safe language) im Gegensatz z.B. zu C oder Assembler:

- keine undefinierten Effekte (wie z.B. Pufferüberlauf), die von Angreifern gezielt ausgenutzt werden könnten
- Bevor der Klassenlader (class loader) eine Klasse in die JVM lädt, werden etwaige Manipulationen an der Klasse, die zu syntaktisch fehlerhaftem Bytecode führen (aber auch nur diese!), vom Bytecode Verifier entdeckt.

Zusätzlich spezielles Java Schutzsystem

- schützt vor **Trojanischen Klassen**,
  lokal oder (mit Applets) übers Netz geladen,
- gemäß einer vorgegebenen **Schutzstrategie** (security policy),
- zusätzlich zum Schutz durch das Betriebssystem.
- 3 Sicherheitsstufen:
  1. keine zusätzliche Einschränkung
  2. differenzierte **Rechtevergabe**
  3. Programm läuft in **Sandkasten** (sandbox):
     keine Rechte (außer dass Applet Verbindung zum Herkunftsort aufnehmen darf)

Beispiel: Dateizugriff



gewünschter Zugriff

Strategie

zugreifende *Klasse*

JVM

gemäß *Strategie* und *Klasse*
erlaubter Zugriff

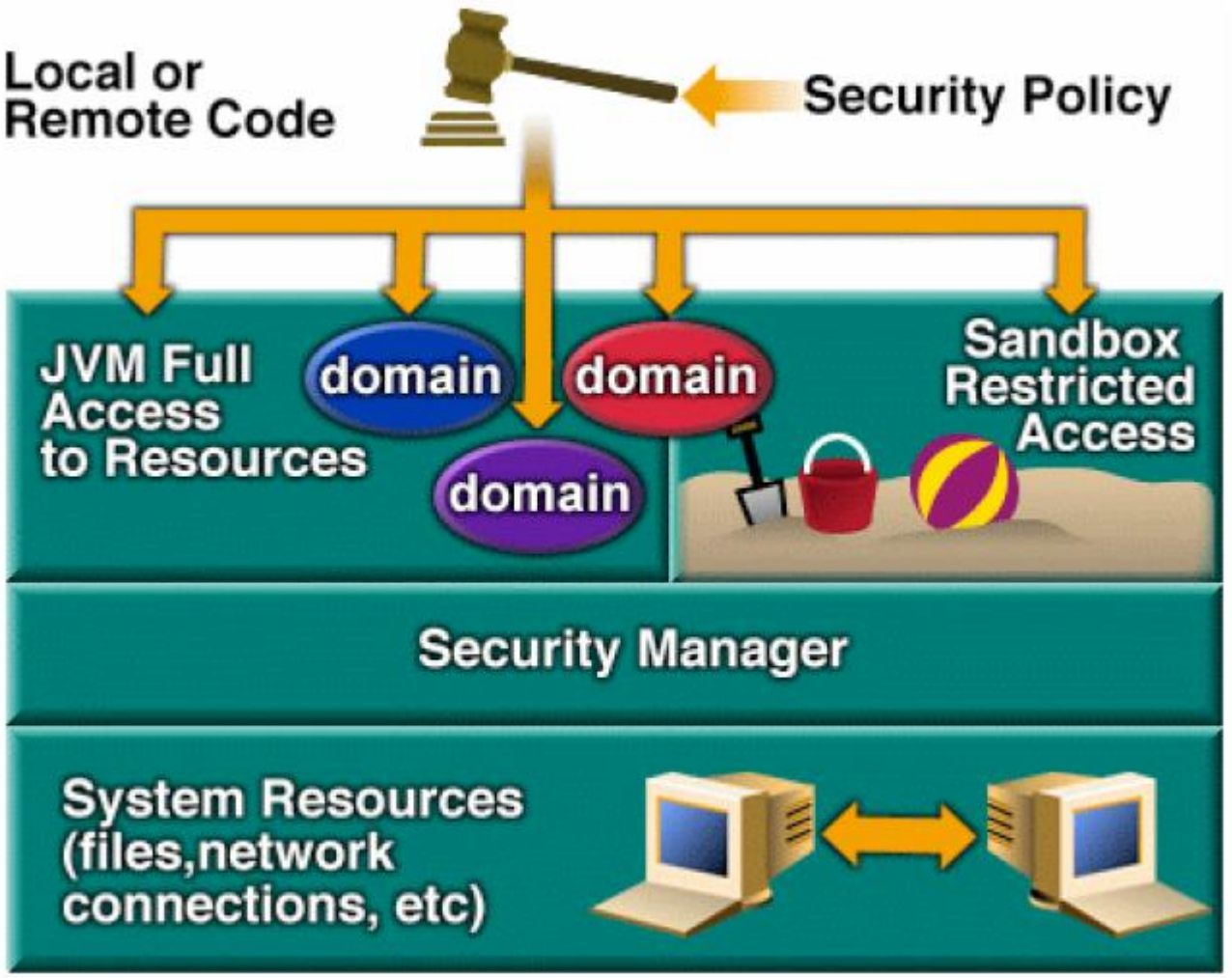
BS

gemäß *Privilegien* und
Schutzstatus erlaubter Zugriff

Zugriffsschutzmodell

- **Herkunft** einer Klasse: gewisse URL oder lokales Verzeichnis
- **Schutzbereich** (protection domain): Menge von Klassen gleicher Herkunft
- **Berechtigung**: Objektbezogen oder objektungebunden, überwacht werden
  - Manipulation von Threads
  - Dateizugriffe
  - Netzzugriffe
  - Benutzung von Systemaufrufen und Programmen
- ***grant*** definiert die Schutzrechtsstrategie

Java Policy Konzept



Formulierung der Schutzstrategie

in Strategie-Dateien (policy files) .policy :

- globale (Standard-) Strategie
  unter `.../java.home/lib/security/java.policy`
- persönliche Strategie unter
  `<myhome>/.java.policy`

- Keine Beschränkung für `java Prog`
  wohl aber für `java -Djava.security.manager Prog`
- Effektive Strategie
  ergibt sich aus beiden (und eventuell weiteren) Strategien
- Sandkasten, falls keine Strategie-Dateien angegeben

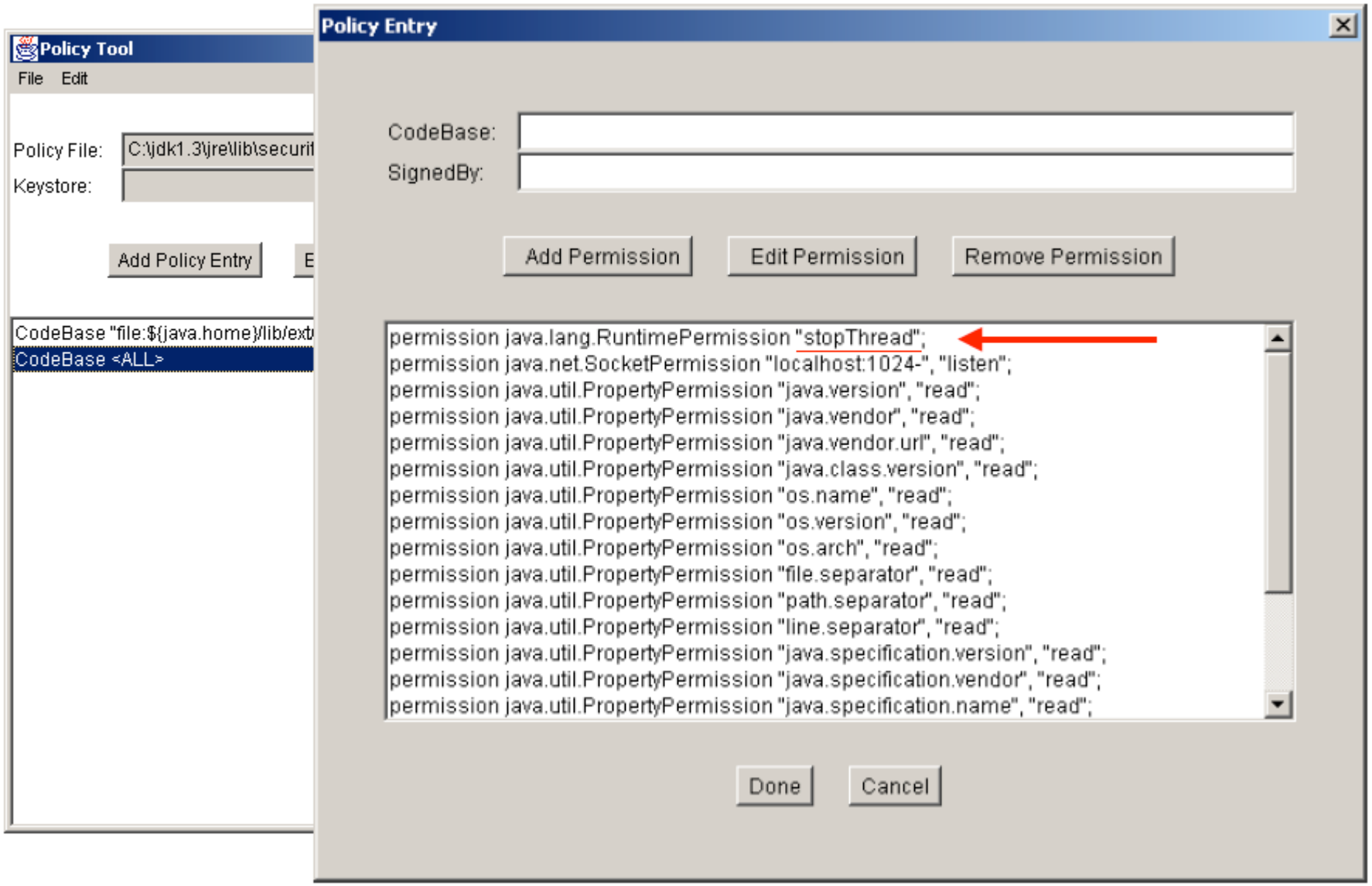
Typische Strategie-Datei

```
grant  codeBase "http://www.bsi.de/trusted/classes/*" {
        permission java.io.FilePermission "/*", "read";
        permission java.io.FilePermission "/tmp/*", "read,write";
};
grant  codeBase "file:/home/buddy/classes/*" {
        permission java.io.FilePermission "/usr/charly/*", "read";}

    // Standard extensions get all permissions by default:
grant  codeBase "file:${java.home}/jre/lib/ext/*" {
        permission java.security.AllPermission;
};

    // allows anyone to listen on un-privileged ports:
grant {
        permission java.net.SocketPermission "localhost:1024-","listen";
};
```

Verwaltung von Strategie-Dateien: policytool



Permission Klassen & Weiteres

- Abstrakte Klasse: `java.security.Permission`
- Mit Unterklassen:

```
java.lang.RuntimePermission
java.io.FilePermission
java.security.AllPermission
......
```

- Weiteres:
  - **JAAS** - Java Authentication and Authorisation Service
  - **JCE** - Java Cryptographic Extensions
  - **JSSE** - Java Secure Socket Extensions

Angriffe auf das Passwortsystem

Ziel: Unautorisierter Zugang zu Systemen/Accounts

Hintergrund:

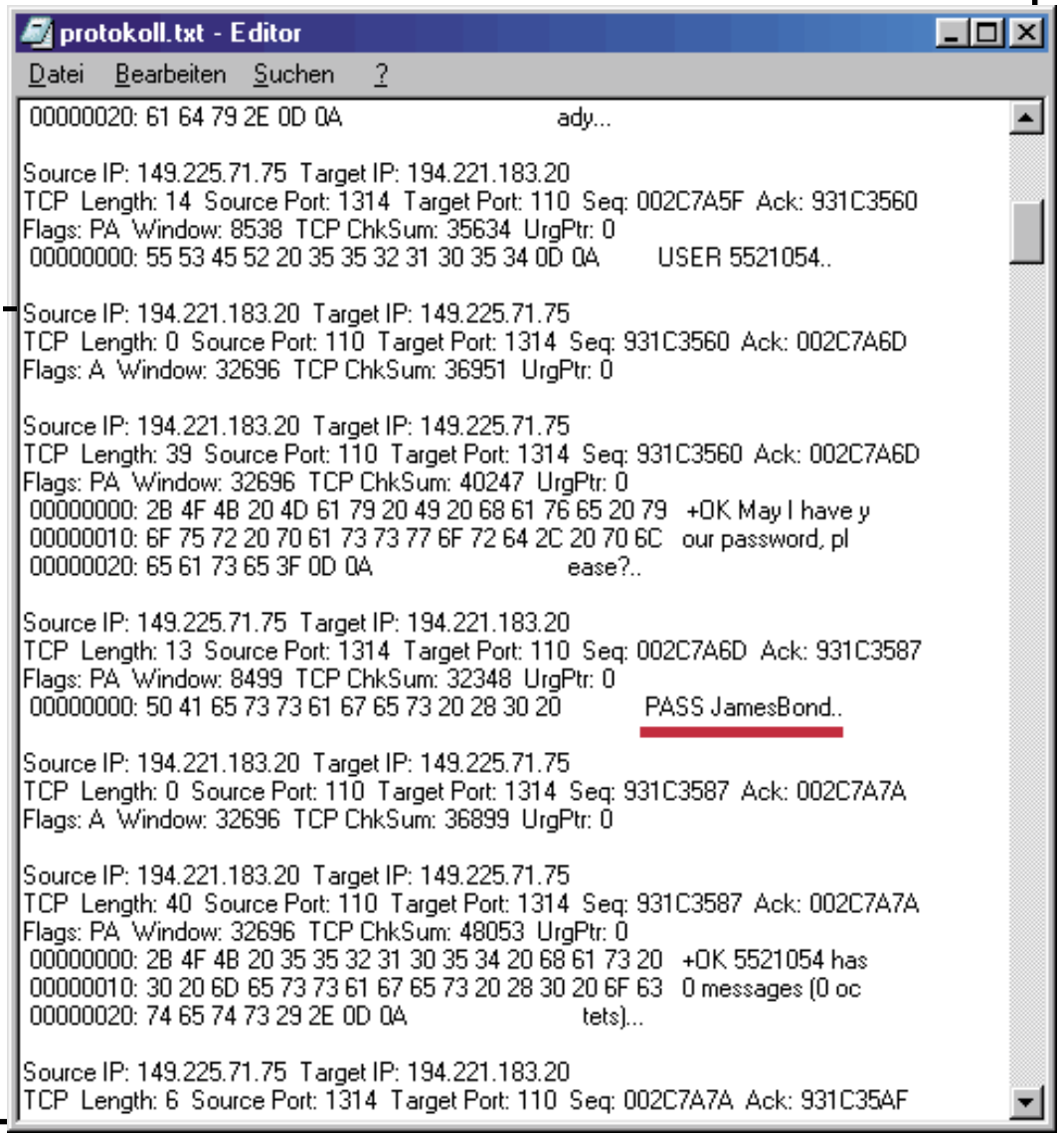
- Passworte häufigste Art der Authentifizierung
- Unterstützung in fast allen Systemen
- flexibel, kostengünstig

Alternativen Biometrie, Chipkarten:

- teilweise bereits praxistauglich
- wenig verbreitet
- Zusatzkosten

Angriffe auf das Passwortsystem

- Offene Zugänge ohne Passwort (Gast-account)
  - Kein Angriff notwendig
- schwache Passwörter und Standard-passwörter
  - Angriff durch
    - Brute-Force-Attacken,
    - Dictionary Attacken
- Klartextübertragung
  - Angriff durch
    - Network Sniffing
- Diebstahl (und entschlüsselung) von Passwort-Dateien
- „Social Engineering“



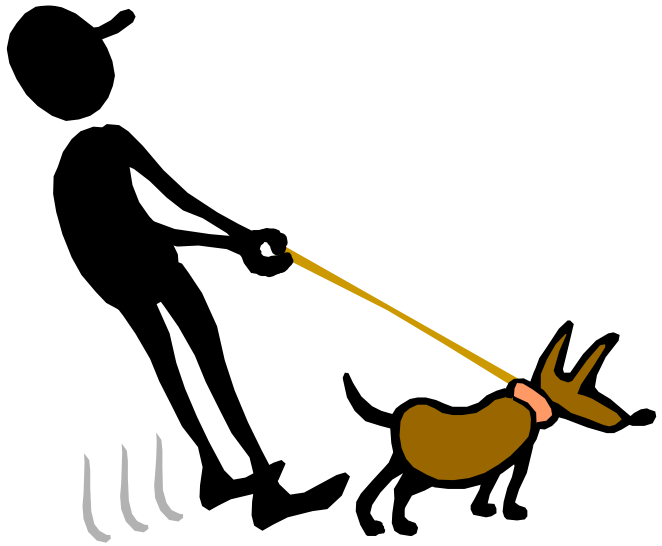
Schwachstellen in Passwortsystemen

- **Benutzerverhalten**:
  - Benutzer wählen schwache Passwörter
  - starke Passwörter sind schwer zu merken
  - viele Dienste - ein Passwörter (Single-Sign-On)
- **Implementation**:
  - Klartextübertragung
  - Schwache Verschlüsselung
  - Passwörter-Dateien für alle lesbar
- **Systemumgebung**:
  - Benutzernamen über Netzdienste feststellbar (finger)
  - Login-Versuche und Passwörter-Überprüfungen werden nicht dokumentiert

Beispiele für fehlerhafte Konfigurationen

- **Fehlendes Administrator-Passwort** bei Microsoft SQL-Server 7
- **automatische Freigabe von Laufwerken** bei der Installation von Netz-Software (Win 95)
- **Standardmäßige Aktivierung** (unsicherer) Dienste (in Linux-Distributionen)
- **automatisches Anzeigen** von aktiven Inhalten in Mailprogrammen (Outlook)

Beispiel Kerberos



Kerberos - Authentifikationssystem

- Am MIT (in Kooperation mit IBM und Sun) Mitte der 80er Jahre entwickelt
- Basiert auf **Needham-Schroeder Protokoll** für symmetrische Kryptosysteme, erweitert um **Zeitstempel**
- Aufgaben
  - Sichere **Authentifikation** von Benutzern und Computern (Principals genannt) in einem (lokalen) Netz
  - Sicherer **Austausch** von Sitzungsschlüsseln
- Realisierung eines **Single-Sign-On** Service für Benutzer
- Sowohl als **Open Source** als auch in kommerzieller Software verfügbar

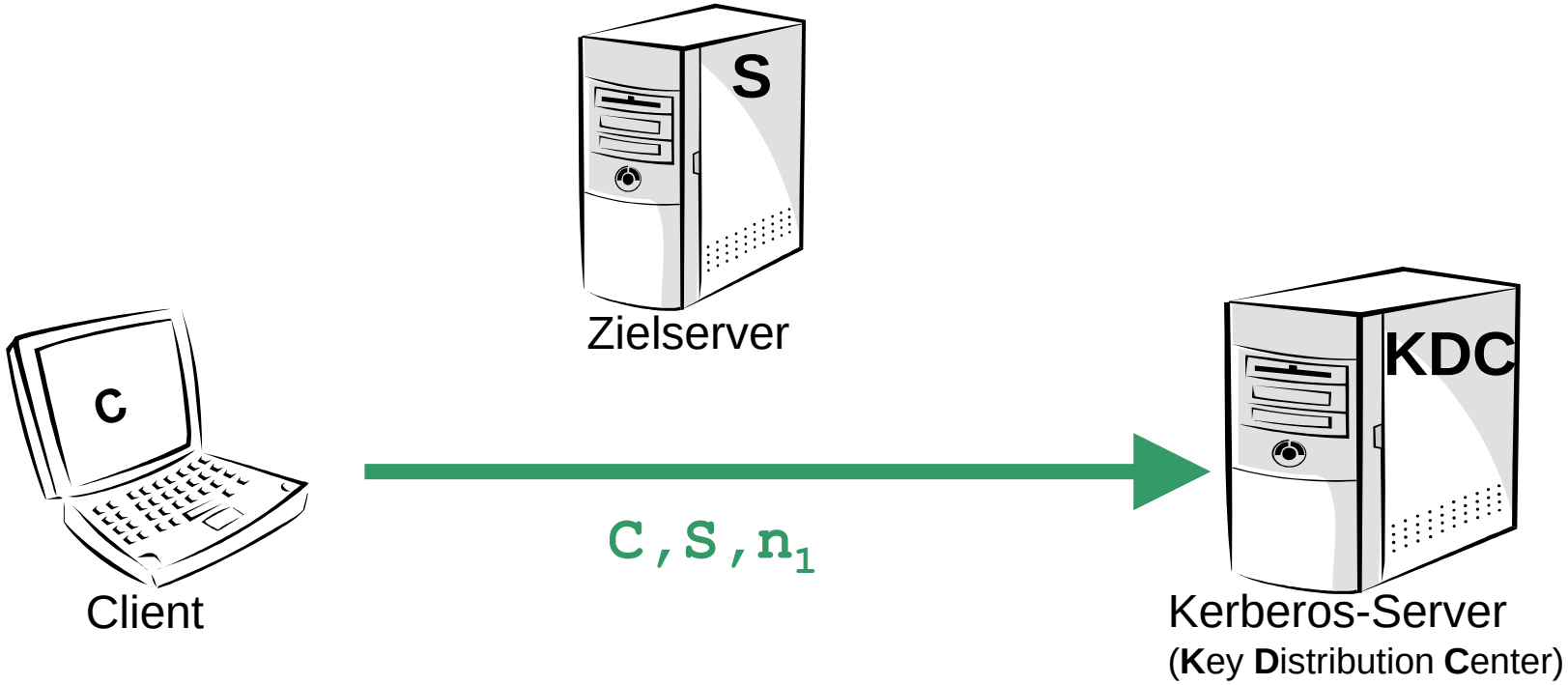
Design

- Der Benutzer muss beim **Einloggen** einmalig seine Authentizität durch Angabe von **Kennung und Passwort** beweisen
- **Passwörter** werden nie als Klartext, sondern immer **verschlüsselt** über das Netzwerk versendet
- Jeder Benutzer und jeder Serverdienst (Principal) hat einen **eigenen geheimen Schlüssel** (bei Benutzern aus dem Passwort abgeleitet)
- **Verschlüsselt** wird symmetrisch mit **DES (D**ata **E**ncryption **S**tandard), in Kerberos Version 5 gibt es auch andere Verschlüsselungsmöglichkeiten
- Die einzige Instanz, die **alle Schlüssel** (Passwörter) **kennt**, ist der **Kerberos Server**, auch **K**ey **D**istribution **C**enter (KDC) genannt

Begriffe

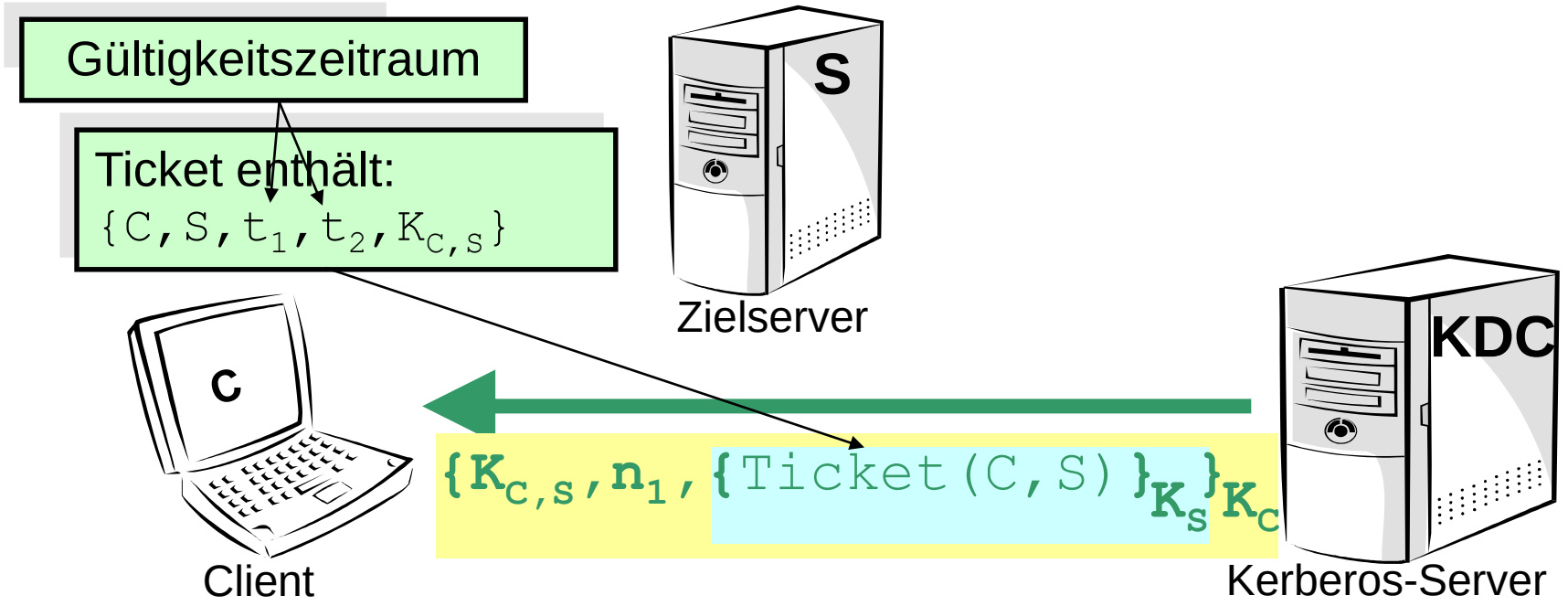
- **Principal**: Eindeutig benannter Benutzer oder Server(dienst), der an einer Netzwerkkommunikation teilnimmt
- **Session key** (Sitzungsschlüssel): Eine Zufallszahl, die vom KDC erzeugt und zeitlich befristet als geheimer Schlüssel zwischen einem Client und einem Server genutzt wird
- **Ticket**: Eine mit einem Serverschlüssel verschlüsselte Nachricht, die dem Server beweist, dass sich der Sender (Client) vor kurzem gegenüber dem KDC authentifiziert hat (beinhaltet einen Sitzungsschlüssel)
- **Nonce** (Einmalstempel): Neu generierte Zufallszahl, die einer Nachricht hinzugefügt wird, um ihre Aktualität zu beweisen; Notation: `n`
- **Time stamp** (Zeitstempel): Eine Zahl, die das aktuelle Datum und die genaue Zeit darstellt; Notation: `t`

Grundprinzip (1) [vereinfacht]



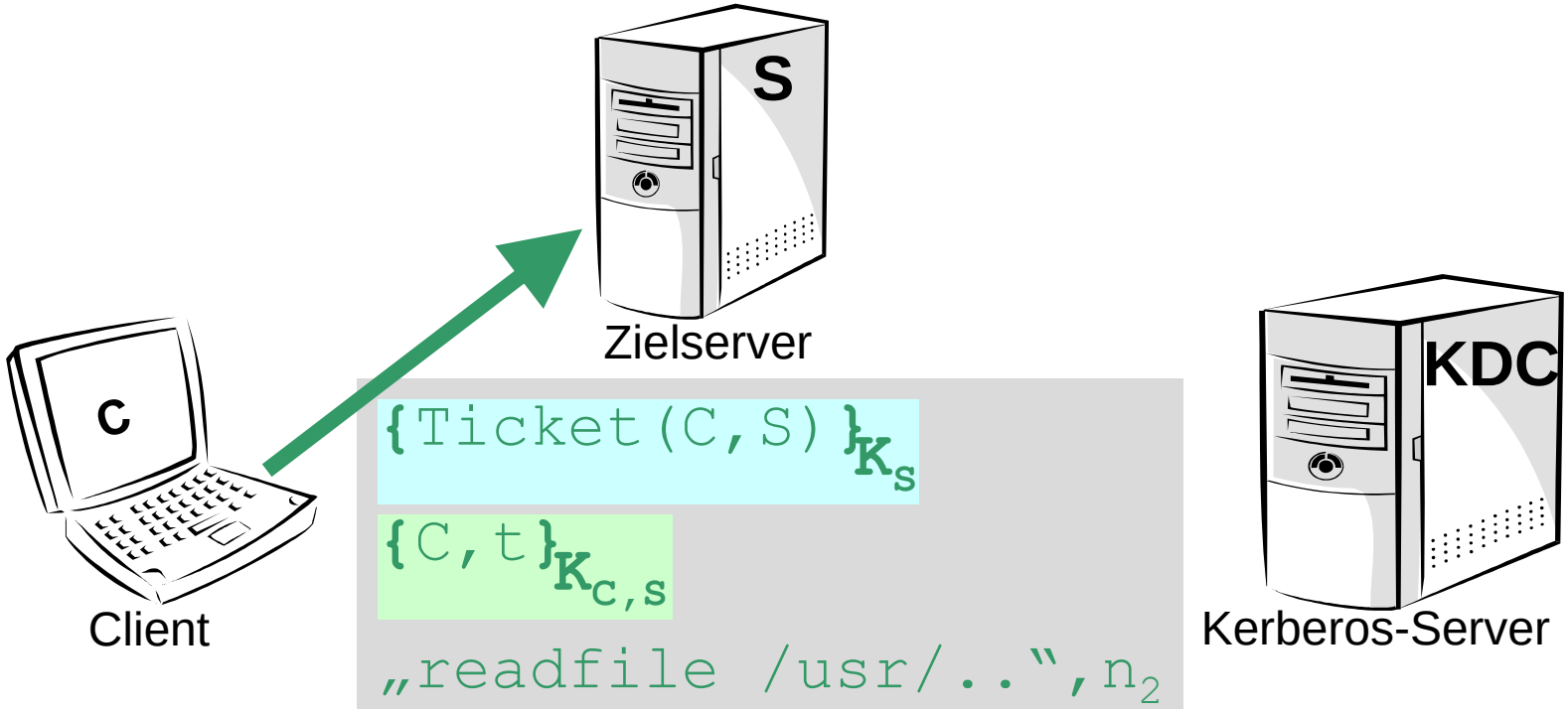
1. Der **Client** `C` sendet eine **Anforderung** für die Kommunikation mit dem Zielserver `S` an den `KDC` (inkl. erstem Nonce-Wert):
   `Benutzerkennung, Zielservername, Nonce`$_1$

Grundprinzip (2)



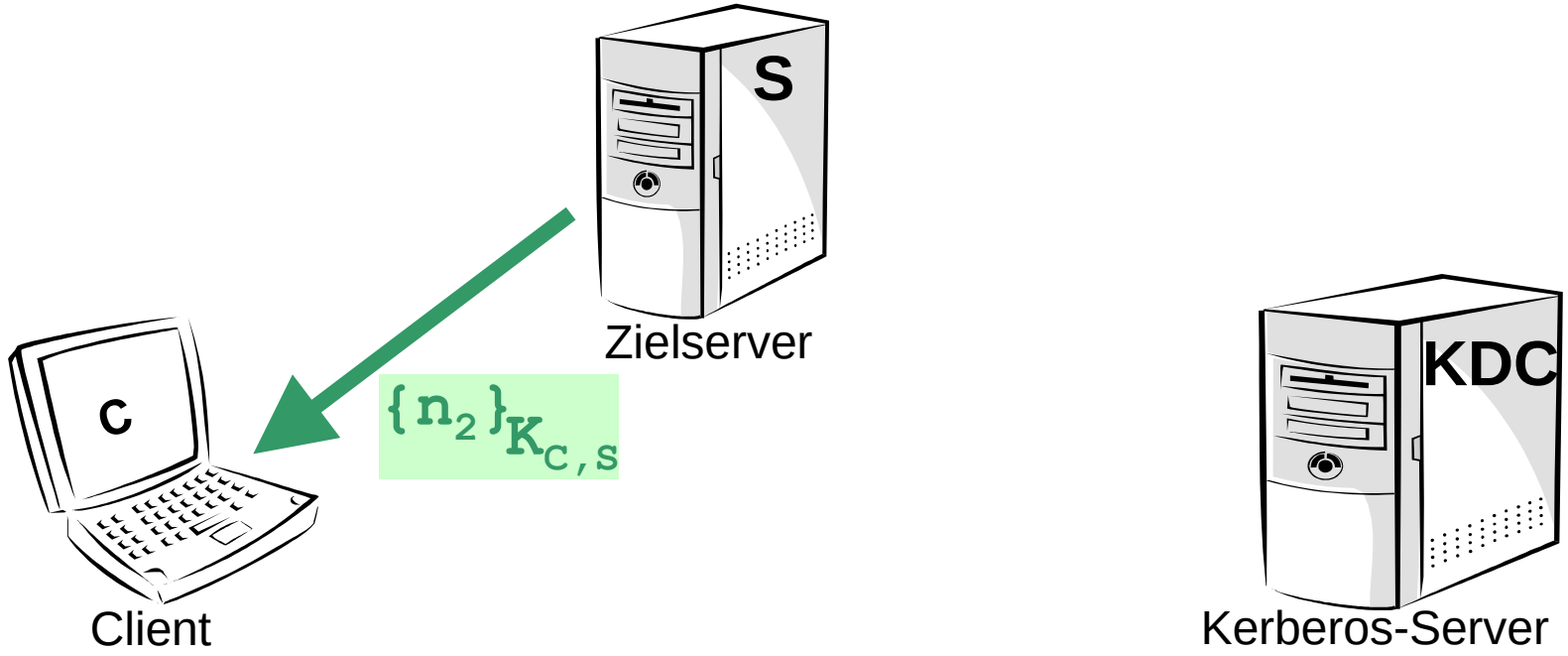
2. Der **KDC** gibt eine mit dem **geheimen Schlüssel von C** verschlüsselte Nachricht zurück, die einen **neu erzeugten Sitzungsschlüssel** $K_{C,S}$ für C und den Zielserver S enthält, ebenso wie ein **Ticket**, das mit dem **geheimen Schlüssel** K_S **von S** verschlüsselt ist.

Grundprinzip (3)



3. Der Client sendet das mit K_S verschlüsselte Ticket mit einer **neu erzeugten Authentifizierungsnachricht** (Name und Zeitstempel, verschlüsselt mit dem gemeinsamen Sitzungsschlüssel $K_{C,S}$) sowie eine Dienstanforderung an den Zielserver S (inkl. zweitem Nonce-Wert)

Grundprinzip (4)



4. Der Zielserver S sendet den mit dem gemeinsamen Sitzungsschlüssel $K_{C,S}$ **verschlüsselten Nonce-Wert** zurück.

Beide sind gegenseitig **authentifiziert**!!

Problem der vereinfachten Lösung

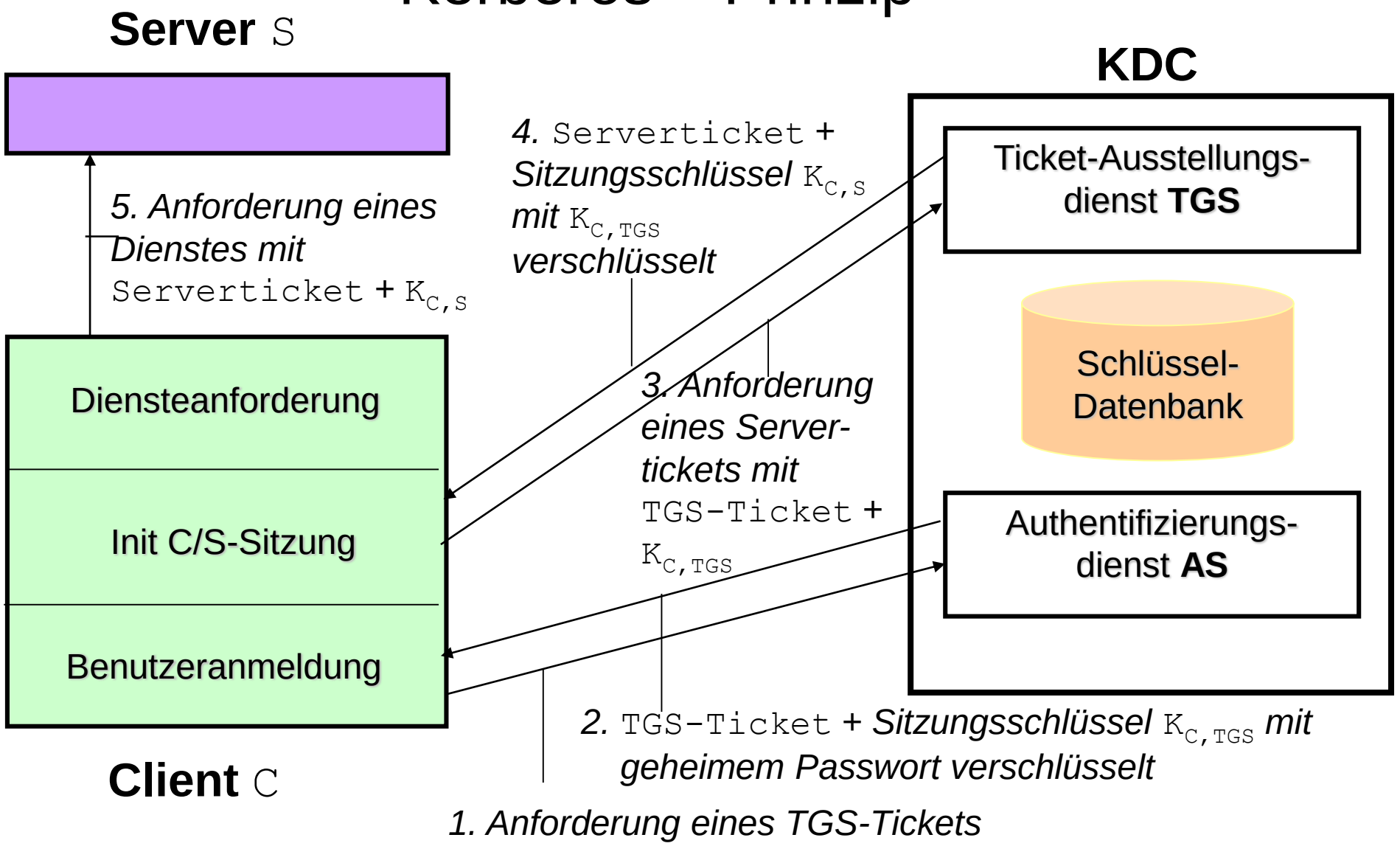
- **Problem**: Da für jeden Serverdienst ein eigenes Ticket nötig ist, muss das **Benutzer-Passwort** (zur Ableitung des Benutzerschlüssels)
  - vom Benutzer **mehrfach eingegeben** werden (lästig, nicht zumutbar)

oder

  - das Passwort im Speicher des Client **gehalten werden** (zu gefährlich!!)

- **Lösung**:
  - Erweiterung des KDC um einen **Ticket-Granting-Service** (TGS) zur Ausstellung von Tickets unabhängig von der Passwort-Authentifizierung
  - Statt des Passworts muss nun nur noch das **TGS-Ticket** und der Sitzungsschlüssel zur Kommunikation mit dem TGS im Client-Speicher gehalten werden, d.h. **einmaliges Eingeben des Passwortes** reicht aus!!

Kerberos – Prinzip



Protokollbeschreibung Version 4

	Von	An	Nachricht
1	Client C	KDC (AS)	C, TGS, n_1
2	KDC (AS)	Client C	$\{K_{C,TGS}, n_1, \underbrace{\{C, TGS, t_1, t_2, K_{C,TGS}\}_{K_{TGS}}}_{\text{TGS-Ticket}} \}_{K_C}$
3	Client C	KDC (TGS)	$\underbrace{\{C, TGS, t_1, t_2, K_{C,TGS}\}_{K_{TGS}}}_{\text{TGS-Ticket}}, \{C, t\}_{K_{C,TGS}}, S, n_2$
4	KDC (TGS)	Client C	$\{K_{C,S}, n_2, \underbrace{\{C, S, t_1, t_2, K_{C,S}\}_{K_S}}_{\text{Serverticket}} \}_{K_{C,TGS}}$
5	Client C	Server S	$\underbrace{\{C, S, t_1, t_2, K_{C,S}\}_{K_S}}_{\text{Serverticket}}, \{C, t\}_{K_{C,S}}, Command, n_3$

Grenzen und Einsatzgebiet von Kerberos

- Alle TGS-Tickets sind mit dem gleichen Schlüssel chiffriert, dem **Kerberos Master Key**
- Kein Schutz vor **Systemsoftwaremodifikationen**
- **Alles** muss „**kerberorisiert**“ werden
  (Angriff auf Client genügt!)
- Kerberos Server muss funktionieren (**single point of failure**)
- Einsatz in **homogenen Umgebungen**
  - Firmennetz / Campusnetz
  - im Rahmen eines Verzeichnisdienstes

Beispiel OAUTH

OAUTH Delegationssystem

- Ziel: Autorisierung durch Dritte (3rd Party Authorisation) mit kontrolliertem Zugang zu Ressourcen
- 2006 initiiert von Twitter
- In der IETF seit 2008 zu Standardisierung entwickelt
- OAUTH 2.0 Framework ist RFC 6749 (2012)
- Aufgaben:
  - Sicheres Teilen von Autorisierungsinformation
  - Ohne die Authentifizierungsdaten zu teilen
- Anwendung:
  - Dienste-Kopplung, z.B. PayPal im Webshop
  - ID-Verifikation für Dritte (z.B. bei Github)

OAUTH Entities

User

User Agent
(Web Browser)

Authorization Request

Resource Consumer
(LinkedIn)

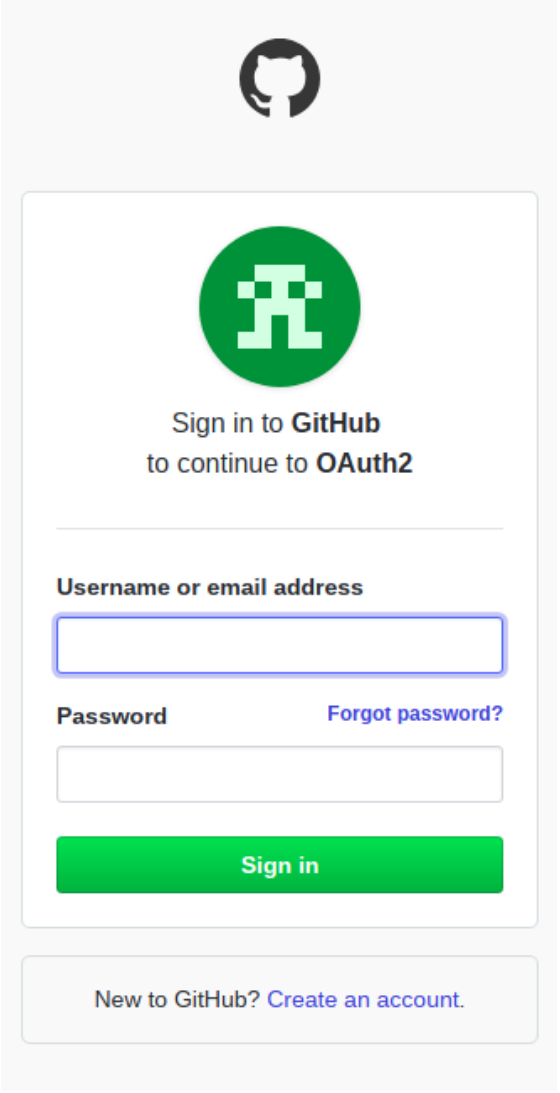
Token request

Authorization Server
(Github)

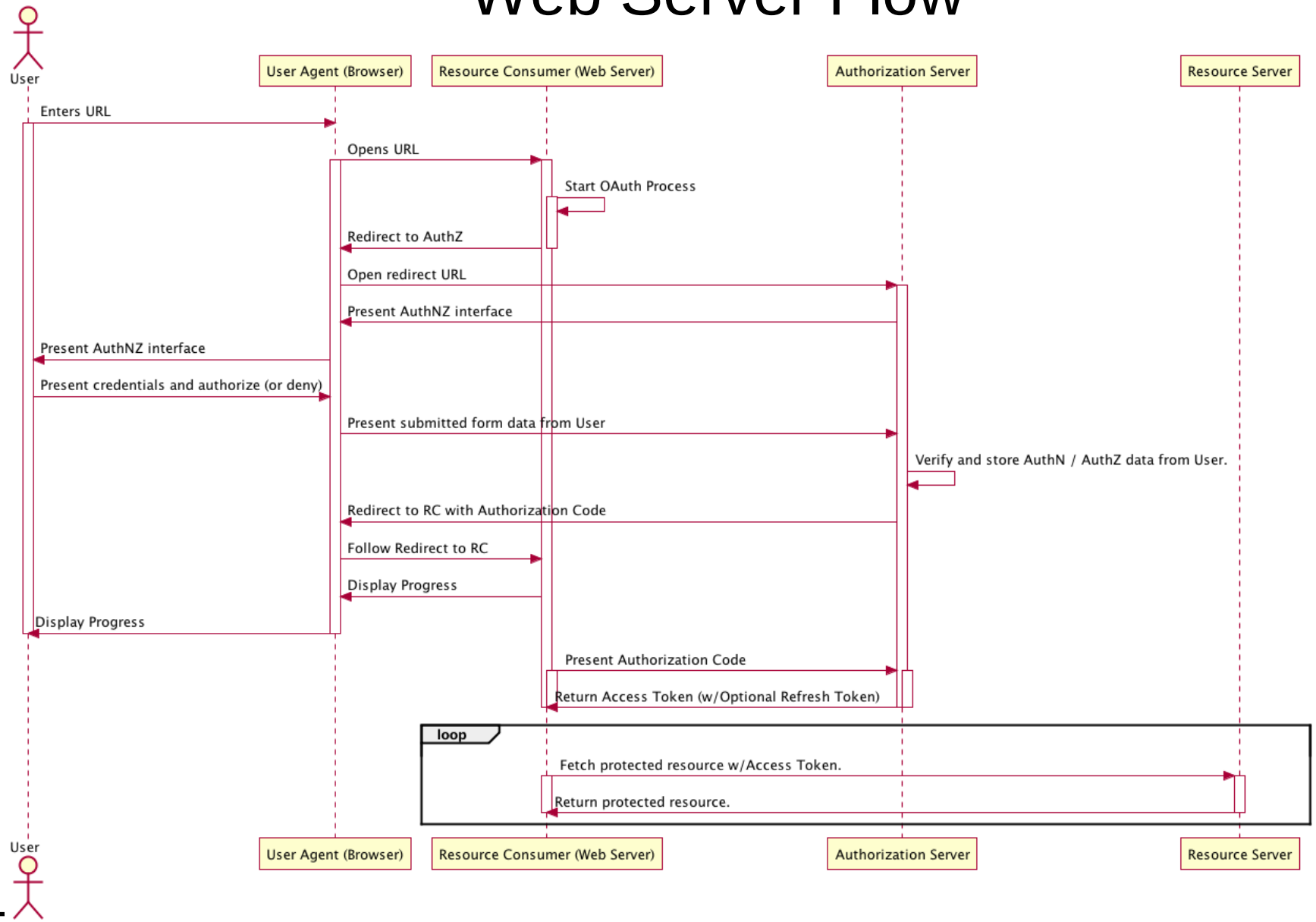
Access Request
(incl. Token)

Resource Server
(Github)

Embedded Login Interface



Web Server Flow



OAuth2 – Authentifizierung

- Authentifizierung gegenüber Github im Browser
  https://**github.com**/login/oauth/authorize?**client_id=1a887169659081acdbbe**

- Die **client_id** identifiziert die Anwendung, welche Github als Oauth Service Provider nutzt (z.B. LinkedIn)
- Nach erfolgreicher Authentizifierung wird zur Anwendungs-URL weitergeleitet
  http://localhost:8080/?**code=1a3e85ffae24a3838b3b**

- Der Queryparamerter **code** kann temporär zum Erstellen eines **access_token** verwendet werden

OAuth2 – Token Request

- Erstellung eines **access_token** mittels **curl**
  curl -X POST
  https://**github.com**/login/oauth/**access_token?code=1a3e85ffae24a3838b3b**&**client_secret=37ee8cd6f8149d380bc5df5e012c388f17834a42**&**client_id=1a3e85ffae24a3838b3b**
- Das **client_secret** stellt sicher, das der Request von der registrierten Anwendung stammt
- Als Antwort erhält man den **access_token** und den **token_type**
  access_token=28c3ff68702a68b2857281e33f95de9ae6d89193&scope=&**token_type=bearer**

OAuth2 – Resource Access

- Mittels des Tokens ist der Benutzer authentifiziert und es können z.B. Benutzerinformation abgerufen werden
  curl
  https://**api.github.com**/user?**access_token=28c3ff68702a68b2857281e33f95de9ae6d89193**
- { "login": "Username",
  "id": 77777777,
  "node_id": "xxxxxxxxxxxxxxxxxxxxxxxxxxx",
  "avatar_url": "https://avatars3.githubusercontent.com/u/111111?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/Username" ,
  "html_url": "https://github.com/Username",
  "followers_url": "https://api.github.com/users/Username/followers",
  "following_url": "https://api.github.com/users/Username/following{/other_user}",
  "gists_url": "https://api.github.com/users/Username/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/Username/starred{/owner}{/repo}",
  ….
  }

Probleme

- **Sicherheitsanforderungen** und -modelle sind **vielfältig**:
  - Uni-Netzwerk vs. Buchungssystem einer Bank
  - Authentisierung durch Passwort, Chipkarte, Iris-Scanner...
  - Autorisierung basierend auf Benutzer, Rolle, Sicherheitseinstufung, Zugriffslisten...
  - Zugriffskontrolle pro Methodenaufruf, pro Objekt , pro Server ...
  - Nachrichtenübermittlung im Klartext, symmetrisch verschlüsselt, asymmetrisch verschlüsselt...
- **Konsequenz**: Sicherheitsdienst stellt im Wesentlichen **Mechanismen** bereit, mit denen verschiedene Sicherheitspolitiken durchgesetzt werden können.