F Sicherheit von Programmcode

F.1 Überblick

- Sicherheitsprobleme bei der Ausführung von Software
- Mobiler Code
- Sprachbasierter Schutz
- Schutz zur Laufzeit
- Java-Sicherheit
- Sicherheit der Systemschnittstelle
- Symbian OS

F.2 Sicherheitsprobleme bei der Ausführung von Software

- Schwachstellen und Bedrohungen bei der Ausführung von Software
  - Quelle des Codes unklar
  - manipulierter Code
  - fehlerhafter Code
  - unbekannte Funktionalität
- Gegenmaßnahmen
  - Quelle sicherstellen, Manipulationen erkennen
    - Hashes veröffentlichen
    - Code signieren
  - Sicherheit durch Softwareentwicklungsprozess garantieren
    - Code Reviews
    - Unterstützung durch Programmiersprachen und Entwicklungswerkzeuge

F.3 Mobiler Code

1 Überblick

- zunehmende Verbreitung verteilter Softwaresysteme
  - Agentensysteme
  - Applets und Servlets, ActiveX-Controls
  - Aktive Netzwerke
  - Plug-ins
  - aktive Inhalte von Web-Seiten und E-Mails
- ▲ Mobiler Code
  Software, die auf einem entfernten, potentiell nicht vertrauenswürdigen Rechner generiert wurde und die auf einem Gastrechner ausgeführt wird
  - grundlegender Unterschied zu Client-Server-Modell:
    Software ist auf den Knoten resident, es werden nur Daten (Aufrufparameter) transportiert
- Plattformabhängig (z. B. ActiveX) oder -unabhängig (z. B. Java)

2 Bedrohungen

- Unsicherheit über die tatsächliche Quelle des Codes
  - wurde der Code bereits an der Quelle verfälscht?
- Übertragung des Codes über ein Transportmedium
  - was passiert unterwegs mit dem Code?
- Angriffe auf den mobilen Code auf dem Gastrechner
  - sind die ausführenden Subjekte berechtigt?
  - sind ggf. lokale, sensible Daten des Codes (z. B. Kommunikationsschlüssel) gegen unberechtigte Zugriffe geschützt?
  - kann der Code vor oder bei der Ausführung verfälscht werden?
  - was tut die Ausführungsumgebung tatsächlich mit dem Code?
  - was erwartet der Absender des Codes von der Ausführung?
  - bewegt sich der Code evtl. danach weiter oder zurück zum Absender (Agentensystem) - und in welchem Zustand ist er dabei?

2 Bedrohungen (2)

- Angriffe des mobilen Codes auf den Gastrechner
  - ◆ was soll der Code tun, was tut er tatsächlich?
    - ➤ auf welche Daten kann er zugreifen?
    - ➤ welche Daten überträgt er wohin?
    - ➤ welche Ressourcen kann er belegen? (Denial-of-Service-Aktivitäten)
    - ➤ Maskierungsangriffe
      - – Vortäuschen einer falschen Identität
      - – Vortäuschen einer falschen Funktionalität

! Sicherheit über die Quelle des Codes bringt nicht unbedingt Sicherheit über die Funktionalität

- ➤ auch signierter Code kann bedrohliche Funktionalität beinhalten!

3 Beispiel: ActiveX-Controls

- Erweiterung von Microsofts OLE-Technologie
  - ➤ ab Mitte der 1990er Jahre
- Wiederverwendbare Softwarekomponenten in Binärform
  - ➤ Ausführung in spezieller Ausführungsumgebung z. B. Browser
  - ➤ Code wird über Netzwerk aus *Codebase* geladen Angaben z. B. im Inhalt einer Webseite
  - ➤ Ausführung des Codes mit umfangreichen Rechten
    - – Zugriff auf Systemschnittstelle
    - – Zugriff auf Dateisystem
    - – Zugriff auf Netzwerk

➥ Quelle massiver Sicherheitsprobleme

4 Schutz des mobilen Codes

- Verschlüsselungstechniken, Signieren
  - ➤ Sicherstellen der Quelle
  - ➤ Schutz bei Übertragung
  - ➤ Schutz auf dem Gastsystem bis zur Ausführung
  - ! kein Schutz in der Ausführungumgebung auf dem Gastsystem
- Ausführung nur in vertrauenswürdigen Umgebungen
  - ◆ Basis: Trusted-Computing-Platform Technologie
    - ➤ Konfiguration des Gastsystems kann zuerst über Remote-Attestation-Protokoll abgeprüft werden
    - ➤ spezielle Ausführungsumgebung wird vorab auf Gastsystem geschickt (→ virtuelle Maschine)
    - ➤ mobiler Code wird nur in der speziellen Ausführungsumgebung ausgeführt
  - ◆ Problem:
    - ➤ Qualität von Attestierungsaussagen

5 Schutz des Gastrechners

- Beschränkung der Zugriffsrechte des mobilen Codes
  - ➤ Kontrolle des Zugriffs auf die Systemschnittstelle
  - ➤ Kontrolle des Zugriffs auf Speicherbereiche (→ Sandboxing)
  - ➤ Ausführung in einer abgeschotteten virtuellen Maschine
- Eindeutige Identifikation des Codes
  - ➤ *Authenticode*-Techniken
    - – Hash über Code wird mit privatem Schlüssel des Code-Herstellers signiert → Zertifikat des Herstellers
    - – Echtheit des Hersteller-Zertifikats wird durch Signatur einer Zertifizierungsstelle nachgewiesen (z. B. VeriSign)
  - ➤ Problem: auch zertifizierter Code kann Fehler, Viren, etc. enthalten und damit Bedrohung darstellen
    - – falsches Sicherheitsgefühl beim Anwender!

5 Schutz des Gastrechners (2)

- Proof-Carrying-Code
  - ➤ Gastrechner legt Sicherheitsstrategie fest und veröffentlicht sie Beispiele:
    - – Zugriff nur auf bestimmte Speicherbereiche
    - – keine Pufferüberläufe, weil Feldgrenzen immer geprüft werden
    - – ausschließlich Typ-konforme Zugriffe
  - ➤ Mobiler Code wird vom Erzeuger mit Beweis ausgestattet, dass eine bestimmte Sicherheitsstrategie eingehalten wird
  - ➤ beim Laden des Codes überprüft der Gastrechner den Beweis
    - – spart aufwändige Kontrollen zur Laufzeit
- ◆ Problem: Erstellung der Beweis sehr aufwändig
  bislang nur wenig Unterstützung durch Entwicklungsumgebungen

F.4 Sprachbasierter Schutz

- Schutzkonzepte und Kontrollen auf der Ebene von Programmiersprache, Compiler und Binder
  - ➤ Möglichkeit, Sicherheitsprobleme sehr frühzeitig zu erkennen
  - ➤ feingranularer, Anwendungs-spezifischer Zuschnitt von Sicherheitseigenschaften
    - – im Gegensatz zu grob-granularen Konzepten des Betriebssystems oder der MMU
- Sicherheitskonzepte auf programmiersprachlicher Ebene unterstützen die Entwicklung sicherer Anwendungen
  - ➤ aber kein Ersatz für spezielle Entwicklungsmethoden und -Verfahren bei der Erzeugung sicherer Software
    (→ Security Engineering)

1 Programmiersprachliche Konzepte

- ★ Typisierung
  - ◆ strenge Typisierung
    - ➤ keine Typkonvertierung zwischen Objekten unterschiedlichen Typs
    - ➤ nur Typ-konforme Operationen und Aufrufe
  - ◆ statische Typprüfung
    - ➤ Compiler überprüft Typ-Konformität
    - ➤ Typ-Fehler führen zu Fehler zur Übersetzungszeit
  - ◆ dynamische Typprüfung
    - ➤ Laufzeit-/Betriebssystem überprüft Typ-Konformität
    - – wichtig bei dynamischem Binden von Software
    - ➤ Typfehler führt zu Abbruch der Operation
    - – Problem: Abbruch kann zu System-Inkonsitenzen führen
    - – Lösung: kontrollierte Behandlung von Abbrüchen

1 Programmiersprachliche Konzepte (2)

- ★ Ausnahmebehandlung
  - ◆ Fehlersituationen werden abgefangen und durch Behandlungsroutinen speziell bearbeitet
    - ➤ System-Inkonsistenzen können aufgeräumt werden
    - ➤ fehlende Rechte können evtl. dynamisch nachgefordert werden und die abgebrochene Bearbeitung kann fortgesetzt werden
- ★ Transaktionskonzept
  - ◆ Unterstützung zur Bereinigung von inkonsisteten Zuständen beim Abbruch von Bearbeitungen
  - ◆ in Datenbanksystemen gängige Technik (ACID-Transaktionen), in Programmiersprachen nicht etabliert

1 Programmiersprachliche Konzepte (3)

★ keine direkten Speicherzugriffe

- ◆ häufige Fehlerquelle: direkte Speicherzugriffe über Zeiger
  - – zusätzliches Problem: Zeiger-Arithmetik
- ◆ Lösung: statt direkte Speicherreferenzen nur typisierte Objektreferenzen

★ keine explizite Speicherverwaltung

- ◆ Problem: vergessene Freigaben → Speicherverlust
- ◆ Problem: Freigabe und Wiederverwendung von Speicher auf den noch Objektreferenzen zeigen
  → kann zu nicht-erkennbaren Typverletzungen führen
- ◆ Lösung: Allokation von Objekten und automatische Garbage Collection

2 Compiler und Binder

★ Informationsflusskontrolle

- ◆ Menge von Sicherheitsklassen
- ◆ Flussrelation beschreibt zulässige Informationsflüsse zwischen den Klassen
- ◆ jedes Datum X wird mit einer Sicherheitsklasse $l(X)$ versehen
- ◆ für jedes Sprachkonstrukt wird spezifiziert, welche Informationsflüsse durch seinen Einsatz entstehen können
  - ➤ explizite Informationsflüsse (z. B. durch Zuweisungen)
  - ➤ implizite Informationsflüsse (z. B. als Ergebnis einer if-Abfrage)

★ allgemeine Compilertechniken

- ◆ Prüfung der Einhaltung von Typregeln
- ◆ Strenge Typisierung erlaubt die Prüfung von Wertebereichen und die Erkennung von Bereichsüberschreitungen
  - ➤ häufige Ursache für Sicherheitsprobleme
- ◆ Einhaltung von Sichtbarkeitsregeln

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Compiler und Binder (2)

★ Generierung von Kontrollcode

- ◆ Erkennen von Speicherüberläufen, Verletzung von Feldgrenzen
- ◆ Bereitstellung von Attributinformationen (z. B.Typinformationen)
- ◆ Aufrufe über symbolische Namen und Aufruftabellen
  - ➤ keine direkten Sprünge auf Speicheradressen
  - ➤ Aufruftabelle ermöglicht dynamische Kontrolle der Aufrufe (z. B. Zwischenschalten von Kontrollobjekten)
- ◆ kontrollierte Speicherzugriffe und Sprünge

★ Binden

- ◆ Problembereich: dynamisches Binden
  - ➤ wer kontrolliert, welcher Code zur Laufzeit hinzugebunden wird?
  - ➤ ungeprüfter Binärcode kann Sicherheit komplett zerstören
  - ➤ nur wenn Übersetzer genügend Kontrollinformation für Binder zur Verfügung stellt, ist ausreichende Prüfung möglich!

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.5 Schutz zur Laufzeit

■ Eingrenzung der Programmausführung durch Hardwaremechanismen

- ➤ logische/virtuelle Adressräume
- ➤ MMU

■ Eingrenzung durch Softwaremechanismen: Sandboxing

- ◆ Aufteilung eines Adressraums in Regionen
  z. B. 256 Sandboxes à 16 MB in einem 32-Bit-Adressraum
- ➤ alle Adressen innerhalb einer Sandbox sind in den oberen 8 Bit gleich
- ➤ getrennte Sandboxes für Code und Daten
- ➤ Code darf nicht auf Adressen außerhalb der Code-Sandbox springen
  + Code darf nicht auf Daten außerhalb der Daten-Sandbox zugreifen
  - ➥ auch auf Binärcode-Ebene einfach statisch bzw. durch einfache Laufzeit-Tests abprüfbar
  - ➥ getrennte Sandboxen verhindern Code-Manipulationen
- ➤ Systemaufrufe werden über Referenzmonitor (spezielle Sandbox im Adressraum) umgeleitet und dort nach Sicherheitsstrategie überprüft

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.6 Java-Sicherheit

- Grundkonzepte
  - ◆ Plattformunabhängigkeit: Compiler erzeugt maschinenunabhängigen Bytecode
  - ◆ Bytecode enthält umfangreiche Typinformationen
  - ◆ Sicherheitsüberprüfung zur Ladezeit durch Bytecode-Verifier
  - ◆ Ausführung durch JVM - JIT oder Interpreter
  - ◆ Sicherheitskontrolle zur Laufzeit durch Security-Manger
- Java-Applets
  - ➤ mobile Java-Software-Komponenten, die innerhalb einer Anwendung (z. B. Browser) ausgeführt werden
  - ➤ eingeschränkte Rechte zur Ausführungszeit wichtig
- Java-Anwendungen
  - ➤ selbstständige Programme, die in einer JVM ausgeführt werden
  - ➤ Rechte analog zur Ausführung anderer Anwendungen

1 Schutzkonzepte der Programmiersprache

- Objektorientierung unterstützt Kapselung von Objekten
  - ➤ Kapselung durch Sichtbarkeitsregeln (private/public)
  - ➤ Zugriff auf Zustand nur über Methoden-Schnittstelle
- Typisierung stellt sicher, dass nur typkonforme Aufrufe erfolgen können
  - ➤ keine beliebigen Casts möglich
- keine Zeiger und keine Zeiger-Arithmetik
  - ➤ keine direkten Speicherzugriffe
  - ➤ keine Umgehung des Typsystems möglich
- kontrollierte Feldgrenzen
  - ➤ kein Zugriff über Feldgrenzen hinaus
  - ➤ keine unkontrollierten Speicherzugriffe ...

2 Sicherheitsarchitektur

- Strukturierter Bytecode
  - ◆ `.class`-Datei enthält Bytecode und umfangreiche Informationen aus dem Quellprogramm
    - ➤ Typinformationen
    - ➤ maximal benötigter Stack
    - ➤ benötigte Register
  - ◆ Basis für Überprüfungen zur Ladezeit (Bytecode-Verifier) und zur Laufzeit (dynamische Typprüfungen)
  - ◆ essentieller Unterschied zu traditionellen Sprachen (wie C)
    - ➤ Binärcode enthält keine oder nur sehr rudimentäre Informationen aus der Quelle

2 Sicherheitsarchitektur (2)

- Bytecode-Verifier
  - ◆ überprüft Bytecode auf syntaktische und strukturelle Korrektheit
  - ◆ Zugriffe auf undefinierte Register?
  - ◆ Illegale Kontrollflussoperationen?
  - ◆ Datenflussanalyse
    - ➤ keine Typverletzungen zur Laufzeit?
    - ➤ korrekte Initialisierung aller Variablen?
    - ➤ keine Stackgrenzenverletzungen?
- ClassLoader
  - ◆ lädt Code in Ausführungsumgebung der JVM
  - ◆ nicht-vertrauenswürdige Applets erhalten isolierten Namensraum und werden auf interne Methodenaufrufe und Aufrufe vorgegebener Systemfunktionen beschränkt (Sandboxing)
  - ◆ Systemklassen dürfen nicht überschrieben werden

2 Sicherheitsarchitektur (3)

- Security Manager
  - ◆ kontrolliert alle Zugriffe auf sicherheitskritische Systemressourcen innerhalb der JVM
    - ➤ Definition eines neuen ClassLoaders
    - ➤ Zugriffe auf Dateien
    - ➤ Zugriffe auf Netzwerk
    - ➤ Zugriffe auf Betriebssystemdienste
- Sicherheitsprobleme
  - ◆ Bytecode-Verifier führt keine wirkliche Code-Verfikation durch
    - ➤ lediglich Tests auf strukturelle Konformität und Einhaltung einiger Konventionen (z. B. Typisierung)
  - ◆ Bytecode kann manipuliert werden
    - ➤ Verifier akzeptiert auch Code, der nicht den Java-Sprachkonventionen entspricht und so nicht von einem Compiler erzeugt würde (→ trügerische Sicherheit)

2 Sicherheitsarchitektur (4)

- Sicherheitsmodelle
  - ◆ ursprünglich nur Unterscheidung zwischen lokalem (vertrauenswürdig) Code und Applets (nicht-vertrauenswürdig)
    - ➤ problematisch, um mit Applets sinnvolle Anwendungen aufzubauen
  - ◆ signierte Applets ab JDK1.1
    - ➤ auch vertrauenswürdiger Code im verteilten System möglich
    - ➤ Signaturen nur sehr grob-granular (pro Software-Paket = JAR-Datei)
    - ➤ volle Rechte für alle Komponenten eines korrekt signierten Pakets wiedersprechen dem Prinzip der minimalen Rechte
  - ◆ Sicherheitsstrategien ab Java 2

2 Sicherheitsarchitektur (5)

- Subjekte (lokaler oder entfernter Code) unterliegen einer Sicherheitsstrategie
  - ➤ Spezifikation durch Administrator oder Benutzer: Politik-Objekte
  - ➤ Standard-Strategie: Sandboxing
- Sicherheitsstrategie
  - ➤ beschreibt Berechtigungen des Codes in Abhängigkeit von Herkunft oder Signaturen
    (Herkunftsort = URL)
  - ➤ Berechtigungen für Datei-, Netzwerk- oder awt (Fenster)-Zugriffe
    - – anwendungsspezifisch erweiterbar
  - ➤ Schutzdomänen fassen Klassen mit gleicher Herkunft/Signatur zusammen
    - – Rechtevergabe an Schutzdomänen
  - ➤ kontrollierte Kommunikation zwischen Schutzdomänen
    - – über vermittelnden Systemcode oder explizite Rechtevergabe an beteiligte Domänen

2 Sicherheitsarchitektur (6)

- Access Controller
  - ◆ Security Manager leitet seine Überprüfungen an Access Controller weiter
  - ◆ Access Controller prüft auf Basis der Politik-Objekte
    - ➤ hat das ausführende Subjekt (dessen Schutzdomäne) die erforderlichen Rechte?
    - ➤ Aufrufstack enthält alle durchlaufenen Schutzdomänen
  - ◆ unzulässige Aufrufe lösen Exception aus
- Sicherheitsprobleme (2)
  - ◆ Subjekte = Java-Klassen, nicht Benutzer
  - ◆ Rechtevergabe auf Basis Herkunft/Signatur
    - ➤ Herkunft durch URL-Spoofing-Angriffe ggf. fälschbar
    - ➥ stärkere Authentifikationsmaßnahmen oder sichere Transportprotokolle könnten helfen
  - ◆ Prüfung der Signaturen auf Basis der rechnerlokalen Schlüsseldatenbank

F.7 Sicherheit der Systemschnittstelle

1 Motivation

- Angriffe auf Programmcode sind nur problematisch, wenn sie *Auswirkungen* haben
  - ➥ Auswirkung bedeutet Interaktion mit der Umgebung
    - ➤ Mitteilung verfälschter Resultate
    - ➤ falsches Verhalten (= falsche Interaktion)
- Interaktion mit der Umgebung erfolgt über Betriebssystemschnittstellen
  - ➤ Dateisystem, Geräte, Netzwerk
  - ➤ Interprozesskommunikation
- Fundamentales Schutzkonzept:
  Sicherung der Betriebssystemschnittstelle

3 Referenzmonitore

- Idee: alle sicherheitskritischen Systemaufrufe gehen zur Sicherheitsüberprüfung durch eine spezielle Systemkomponente
  - ➤ **Referenzmonitor**
  - ➤ siehe auch: Sandbox-Konzept
- Grundlegendes Konzept: **Trusted Computing Base (TCB)**
  = der minimale Teil des Systems, der notwendig ist, um alle Sicherheitsregeln durchzusetzen
  - ➤ Großteil der Hardware (vgl auch Kap. D7 Trusted Computing)
  - ➤ Teil des Betriebssystems (möglichst minimal)
  - ➤ privilegierte Anwendungsprogramme (möglichst wenige)
- Betriebssystemteil der TCB
  - ➤ Referenzmonitor
  - ➤ Prozesserzeugung/-umschaltung, IPC,
    Teile der Geräteschnittstelle

2 allgemeine Konzepte

- bekannte und weit verbreitete Konzepte
  - ➤ Zugriffsrechte / Access Control Listen
  - ➤ Capabilities
- Problem:
  Absicherung über eine Vielzahl von Systemkomponenten verteilt
  - ➤ Dateisystemschnittstelle (open, read/write, fcntl, …)
  - ➤ Interprozesskommunikation / Netzwerkschnittstelle (bind, IPC, …)
  - ➤ Prozessverwaltung (exec, Signale, …)
- Betriebssysteme enthalten meist viele umfangreiche Systemkomponenten

➥ konsistentes Schutzkonzept nur schwer zu erreichen
- ◆ und noch schwerer auf Dauer konsistent zu halten
  - ➤ wesentliche Ursache für viele Sicherheitslücken

F.8 Beispiel: Sicherheitskonzept in Symbian OS

1 Symbian-Überblick *(www.symbiansigned.com)*

- Betriebssystem für mobile Geräte
  - ➤ Nokia, Ericsson, Sony Ericsson, Panasonic, Siemens, Samsung
  - ➤ Weiterentwicklung aus EPOC-System von Psion
- Struktur
  - ➤ Minimalkern-Architektur
    (Scheduler, Speicherverwaltung, Gerätetreiber)
  - ➤ Basisdienste
    (Basis-Anwendungsschnittstelle, Dateisystem, Plug-In-Framework, kryptographische Funktionen, …)
  - ➤ höhere Betriebssystemdienste
    (Netzwerk, Telefonie, Multimedia, …)
  - ➤ Anwendungsunterstützung
    (Java ME)
  - ➤ User Interface Framework

1 Symbian-Überblick (2)

- Zielplattform
  - Mobiltelefone mittlerer "Größe"
  - Stückzahlen: 10 - mehrere 100 Mio.
  - unterstützte Prozessorarchitektur: ARM
  - Programmiersprache: vor allem C++ (Entwicklungsumgebung mit speziellem gcc)
- Systemziel
  - Plattform für Basissoftware + Plattform für Produkte unabhängiger Softwarehesteller (ISVs - Independent Software Vendors)

1 Symbian-Überblick (3)

- Herausforderung: Sicherstellen der Systemsicherheit
  - ◆ APIs sind dokumentiert, bis Version 8.1 war Softwareentwicklung für jedermann möglich
    - fehlerhafte Programme
    - bösartige Programme
    - Viren, Trojaner, …
  - ◆ System ist weit verbreitet und damit ideale Grundlage für Angriffe

➥ erster Wurm/Virus 2004: Cabir
  - erster verbreiteter Angriff auf Mobiltelfone
  - Ausbreitung über Bluetooth-Schnittstelle
  - Schwachstelle war nicht Symbian sondern der jeweilige Benutzer (Benutzer mussten der Installation zustimmen)

2 Plattform-Sicherheit ab Symbian 9.1

- Hauptschwachstelle des Systems = der Benutzer

➥ zur Verbesserung der Sicherheit muss vor allem die Hauptschwachstelle des Systems beseitigt werden :-)

➥ sicherheitskritische Entscheidungen können nicht "einfach" dem Benutzer überlassen werden
  - ◆ Zustimmung zur Installation "leicht sicherheitskritischer" Software expliziter gestalten
    - Software muss grundsätzlich signiert sein
  - ◆ Entscheidung über "erheblich sicherheitskritische" Software an den Gerätehersteller oder eine zertifizierende Stelle übertragen
    - Anwender kann nicht frei Software auf seinem Gerät betreiben

★ umstrittenes Konzept
  - sehr guter Schutz von Anwendungen und Inhalten
  - Eingriff in die Rechte der Endnutzer

3 Capabilities

- Vergabe an Prozesse auf Basis von Code-Autorisierung
- mehrere Klassen von Systemschnittstellen und Capabilities

▲ User Capabilities
  - erlauben Zugriff auf
    - lokale Netzwerkschnittstellen (Bluetooth, Infrarot)
    - Positionsdatenabfrage
    - kostenverursachende Netzwerkdienste (SMS, …)
    - vertrauliche Daten des Telefonnutzers (lesend und schreibend)
    - Benutzerumgebungsdaten (Audi, Video, biometrische Daten, …)
  - Anwendung muss "Symbian Signed" sein oder Benutzer muss bei Installation bzw. bei jedem Zugriff auf eien Schnittstelle explizit zustimmen

3 Capabilities (2)

- Extended Capabilities
  - ➤ Anwendung wird für diese Zertifizierung zusätzlichen Tests unterzogen
  - ➤ Anwendung muss "Symbian Signed" sein,
    Endnutzer kann die Capabilities nicht vergeben
  - ➤ erlauben Zugriff auf weitere, systemnähere Schnittstellen
    - Power Management (Ausschalten ungenutzter Peripherie, Stand-by-Mode, Telefon ausschalten)
    - Registrierung privilegierter Dienste (protected servers)
    - Zugriff auf Gerätedaten (lesend und schreibend)
    - Erzeugen von Tasten- und Stift-Ereignissen per Software
    - Erzeugen von vertrauenswürdigen Benutzerinteraktionen (UI sessions)

4 Entscheidungen des Benutzers

- User-Grantable Permissioning
  - ◆ Gerätehersteller kann einzelne Capabilies "user-grantable" markieren
  - ◆ Recht kann bei der Installation oder Ausführung durch Benutzer erteilt werden
    - ➤ Blanket permission
      - Benutzer wird bei Installation einer Anwendung gefragt, ob das Recht grundsätzlich erteilt werden soll
    - ➤ Single shot permission
      - Benutzer wird bei jeder Anwendungsausführung gefragt

3 Capabilities (3)

- Phone-manufacurer approved Capabilities
  - ➤ Anwendung muss vom Gerätehersteller zertifiziert sein
  - ➤ erlauben den Zugriff auf die komplette Systemschnittstelle
    - komplettes Dateisystem
    - alle Gerätetreiber (vor allem Kommunikation, Multimedia)
    - Disk Administration
    - Zugriff auf geschützte Inhalte (DRM)
    - TCB-Capability: Zugriff auf Programme und read-only Ressourcen
- ◆ Symbian TCB
  - ➤ Komponenten, die freien Zugriff auf die komplette Hardware und Software haben
    - Kern, Datei-Server, Software-Installer
- ◆ Symbian TCE (Trusted Computing Environment)
  - ➤ weitere sicherheitskritische Systemkomponenten
    (extended oder phone-manufacturer approved Capabilities erforderlich)

5 Identifikation von Programmen

- Capabilities spezifizieren Rechte für den Inhaber
  - ➤ Schnittstelle kann den Inhaber nicht mehr identifizieren
- Problem: wie kann man z. B. den Zugriff auf bestimmte Daten auf Software eines bestimmten Herstellers beschränken?

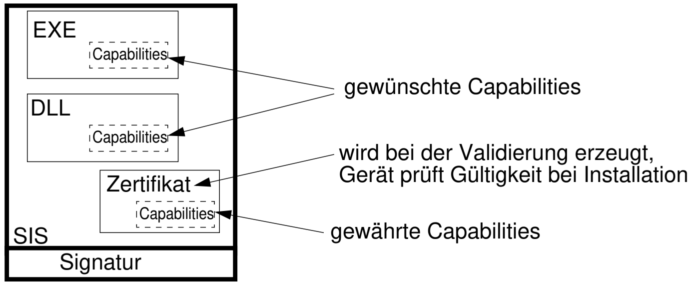
➥ UID-Konzept für Programme – zusätzlich zu den Capabilities
- ◆ SID (Secure Identifier)
  - ➤ eindeutige Id für ein Programm
  - ➤ erlaubt zusätzlichen Schutz von Systemschnittstellen (z. B. Dateisystem)
  - ➤ SID-Bereiche werden an Software-Entwickler vergeben und können beim Signieren überprüft und mit signiert werden (protected UID)
- ◆ VID (Vendor Identifier)
  - ➤ erlaubt Identifikation des Softwareherstellers
- ◆ SID und VID werden auch im Rahmen von IPC übermittelt

6 Einfluss von Capabilities auf Systemschnittstellen

- eingeschränkter Zugriff auf Systemschnittstellen des TCE
  - ➤ Anwendungen müssen bei der Erzeugung mit den notwendigen Capabilities versehen werden
  - ➤ Software-Installer prüft, ob bei signierten Anwendungen die angeforderten Capabilities mit den im Datei-Header festgelegten Capabilities übereinstimmen
    - – Zertifikat-Kette wird bis Wurzel zurückverfolgt
      Zertifikate können zurückgezogen werden
- Dienstprogramme prüfen Capabilities wenn ihre Schnittstellen aufgerufen werden
- Capabilities sind Programmen zugeordnet
  - ➤ Prozess erhält sie beim Laden des Programms (.EXE-Datei)
  - ➤ Bibliotheken (.DLL-Dateien) können nur in einen Prozess eingebunden werden, wenn sie für alle Capabilities des Prozesses autorisiert sind
  - ➤ Capabilities werden generell dem Quellcode zugeordnet und können nach dem Compilieren nicht mehr verändert werden

7 Zertifizierung

- Aufbau installierbarerer Dateien (.SIS-Dateien)



- Zertifikate
  - ➤ Publisher-Zertifikat: identifiziert den Software-Entwickler
  - ➤ Hash zur Verhinderung von Manipulationen
  - ➤ Unabhängige Prüfung → Content-Zertifikat (*chain of trust* zum Symbian Root-Zertifikat auf dem Gerät)

8 Probleme, Sicherheitslücken

- Symbian-Architekturkonzept erhöht die Sicherheit der Systeme grundsätzlich deutlich
  - ➤ es bleiben aber immer noch Schwachstellen

– Hersteller-zertifizierte Software
  - ➤ Sicherheitslücken in solcher Software kompromittieren das ganze System
  - ➤ Beispiel (2008): Nokia Debugger
    – erlaubte Zugriff auf den Speicher
    ➥ maximale Rechte konnten durch durch Setzen eines Bits in der Prozessdatenstruktur an beliebige Anwendungen vergeben werden
    – Lösung: Debugger schreibt nicht mehr in alle Speicherbereiche, neue Firmware verhindert Installation des alten Debuggers

– Betriebssystem selbst
  - ➤ unkontrollierte Pufferüberläufe erlaubten Angriffe und damit das Erlangen maximaler Rechte (2008)

F.9 Linux/Posix-Capabilities

- Rechte in UNIX-Systemen sehr grob-granular
- zu häufig Root-Rechte erforderlich
- Linux-Capabilities erhöhen die Flexibilität
  - ➤ zusätzliche Rechte für nicht-Root-Prozesse
    (z. B. Binden von Port-Nummern < 1024 oder Zugriff auf raw-Sockets)
  - ➤ eingeschränkte Rechte
    (z. B. Nutzung von Systemaufrufen wie setuid, kill, chown, ... verbieten)