Sicherheit von Betriebssystemen – VO 01: Grundlagen der Betriebssystemsicherheit

Paul Kalauner, Rafael Vrecar

Research Group for Industrial Software (INSO)

https://www.inso-world.com

RISE Research Industrial Systems Engineering

FACHHOCHSCHULE WIENER NEUSTADT University of Applied Sciences - Austria

Agenda

- Einführung
- Prozesse & Isolation
- Speicher & Adressisolation
- Kernel
- Rechte & Policies
- Storage & Dateisysteme
- Boot- & Update-Sicherheit

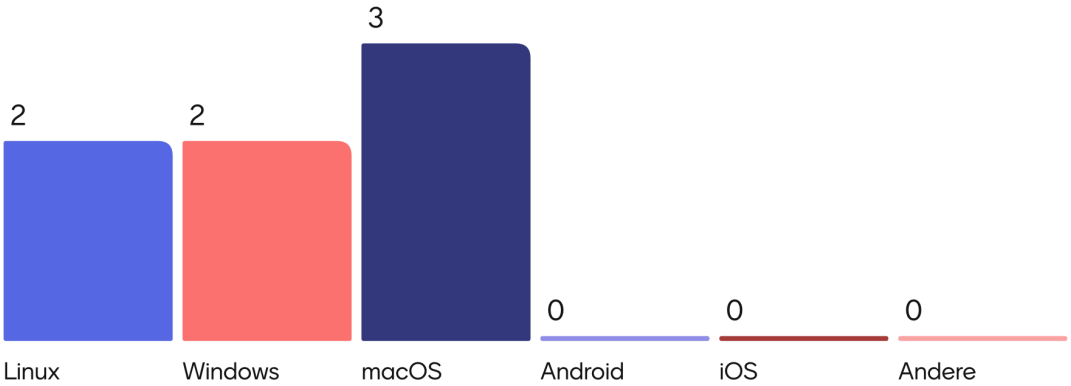
Einführung

Brainstorming:
Motivation Betriebssystemsicherheit

Warum Betriebssystem-Sicherheit? Denken Sie an Bedrohungen, Schutzmechanismen und Schutzziele.

zero-day exploits
token-stealing
over and underflows
unternehmens daten
dangerous usb devices
persönliche daten
datendiebstahl
botnet
heap corruption
malware
passwörter
keylogger
netzwerkangriffe
benutzerkonten
firewall
0 click exploits
cryptominers
datenverlust
unbefugter zugriff
manipulation verhindern
secure boot
signed executables

Welche Betriebssysteme verwenden Sie?



Operating System (OS)

- Betriebssystem als Vermittler zwischen Hardware und Software
- Kontrolliert Zugriff auf CPU, Speicher, Geräte und Dateien
- Erzwingt Sicherheitsgrenzen zwischen Programmen
- Stellt Abstraktionen für Prozesse, Speicher und I/O bereit
- Zentrale Vertrauensbasis (Root of Trust) des Systems
- Kompromittierung des OS = vollständige Systemkontrolle

CIA-Triade

- Confidentiality: Schutz vor unbefugtem Zugriff auf Daten
- Integrity: Schutz vor unautorisierter Veränderung von Daten
- Availability: Systeme und Daten müssen verfügbar bleiben
- Betriebssystem setzt technische Mechanismen für alle drei um
- Sicherheitsmechanismen sind oft Zielkonflikten ausgesetzt
- Grundlage vieler Sicherheitsmodelle und -analysen

Angreifer:innenmodelle

- Klassifikation von Angreifer:innen nach Zugriff
- Lokale:r Angreifer:in: Benutzerkonto oder Code auf dem System
- Remote-Angreifer:in: Zugriff über Netzwerk ohne lokalen Login
- Physische:r Angreifer:in: direkter Zugriff auf Hardware
- OS muss mit all diesen Szenarien umgehen können

Prozesse & Isolation

Prozesse vs. Threads

- Prozess = eigener virtueller Adressraum + Ressourcen
- Threads teilen Speicher innerhalb eines Prozesses
- Prozesse bieten starke Isolation gegeneinander
- Threads sind leichtergewichtig, aber weniger isoliert
- Fehler in Threads können gesamten Prozess kompromittieren
- Sicherheitsrelevanz bei Sandboxen und Exploits

Prozesszustände & Scheduling

- Typische Zustände: ready, running, blocked, terminated
- Scheduler entscheidet, welcher Prozess CPU-Zeit erhält
- Kontextwechsel zwischen Prozessen erfolgt regelmäßig
- Fairness vs. Performance vs. Sicherheit
- Scheduler-Fehler können zu Denial-of-Service führen
- Timing-Angriffe nutzen Scheduling-Effekte aus

User-Space vs. Kernel-Space

- User-Space: Anwendungen mit eingeschränkten Rechten
- Kernel-Space: Vollzugriff auf Hardware und Speicher
- Trennung verhindert Absturz des gesamten Systems
- System Calls als kontrollierte Übergänge
- Direkter Zugriff auf Kernel ist verboten
- Viele Exploits zielen auf Privilege Escalation

Privilege Levels / Protection Rings

- Hardware-gestützte Schutzstufen (CPU-Modi) (z.B. x86 Rings)
  - Ring 0: Kernel Mode, höchste Privilegien
  - Ring 1-2: meist ungenutzt
  - Ring 3: User Mode, eingeschränkte Rechte
- Hardware erzwingt Zugriffsbeschränkungen
- Übergänge nur über definierte Schnittstellen (System Calls)
- Reduziert Schaden durch kompromittierte Anwendungen
- Basis für Isolation und Sicherheit
- Prozessorarchitekturen setzen dieses Prinzip unterschiedlich um, z.B. ARM64: Exception Levels

Context Switch & Security

- Wechsel zwischen laufenden Prozessen/Threads
- CPU-Register und Speicherzustand werden gesichert
- Fehlerhafte Trennung kann Datenlecks verursachen
- Seitenkanalangriffe nutzen Kontextwechsel aus
- Performance-Overhead relevant für Sicherheitsfeatures
- Zentrale Rolle bei Isolation und Multitasking

Trust Boundaries & Attack Surface

- Trust Boundary = Grenze zwischen unterschiedlich privilegierten Komponenten
- Jede Grenze ist potenzielle Angriffsfläche
- System Calls als klassische Trust Boundary
- Kernel-Interfaces besonders kritisch
- Je größer die Attack Surface, desto höher das Risiko
- Prinzip: möglichst kleine und einfache Schnittstellen

Welche Prozesse dürfen niemals root sein und weshalb?

Welche Prozesse dürfen niemals root sein und weshalb?

Bash Skripte

Open Source mit unbekannten Quellen

User Applicationen, externe Apps, Files über externen Netzwerke.

Prozesse die von außen erreichbar sind und Code Exekution ausführen können

Kommt immer darauf an aber grundsätzlich einfach Code den man nicht vertraut oder aus unseriösen Quellen hat Untrusted Code (Downloads, Random Shell Script, etc.) Und Code der nichts am System ändert

Jeder Prozess der die Privilegien nicht unbedingt braucht

Installer Skripte aus dem Web

Benutzernahe Anwendungen (Man hat direkt Kontakt zu "untrusted" Input). Generell Programme die Daten aus dem Internet holen und Daten verarbeiten

Welche Prozesse dürfen niemals root sein?

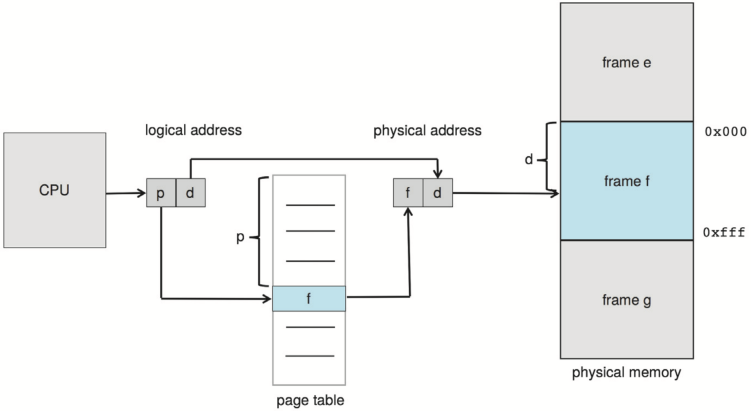
- Normale Anwendungen (z.B. Browser, Editor, Spiele)
  → Root unnötig, erhöht Risiko bei Fehlern oder Exploit
- Dienste ohne Systemzugriff (z.B. einfache Hintergrundprozesse, die keine Hardware oder Systemressourcen benötigen)
  → Root unnötig, erhöht Risiko bei Fehlern oder Exploit
- Ausnahme: Systemdienste (z.B. `sshd`, `systemd`)
  → Müssen root sein, um Systemfunktionen auszuführen

Speicher & Adressisolation

Virtueller Speicher / Address Space

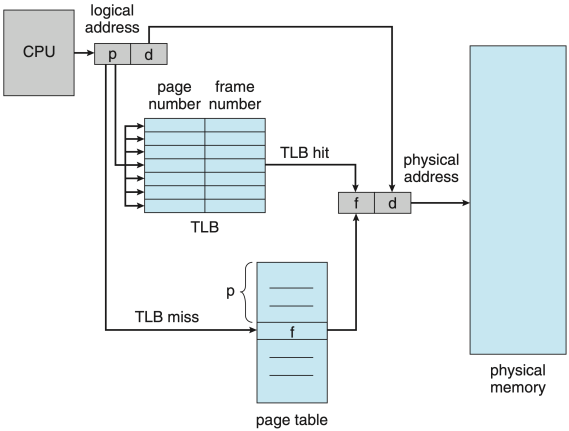
- Jeder Prozess erhält eigenen virtuellen Adressraum
- Abbildung auf physischen Speicher durch Memory Management Unit (MMU)
- Prozesse sehen sich als alleiniger Speicher-Nutzer
- Verhindert direkten Zugriff auf fremde Speicherbereiche
- Grundlage für Isolation und Stabilität
- Fehlerhafte Speicherzugriffe führen zu Segmentation Faults

Übersetzung virtuelle auf physische Adressen



(Vergleiche Silberschatz et al. (2019))

Übersetzung virtuelle auf physische Adressen mit Translation Lookaside Buffer



(Vergleiche Silberschatz et al. (2019))

Speicherisolation

- Jeder Prozess als eigener “Speichercontainer”
- Keine Pointer auf fremden Speicher
- Kernel als vermittelnde Instanz
- Hardware erzwingt Isolation
- Verletzungen führen zu sofortigem Abbruch

Warum kann Prozess A nicht auf Prozess B zugreifen?

Warum kann Prozess A nicht auf Prozess B zugreifen?

Unbefugter Zugriff

Rechte und Sicherheitsmodelle

Kernel händelt System Calls

Speicherisolation

Aufgrund von Speicherisolation. --> jeder Prozess hat eigenen virtuellen Raum für Adressen

Damit man keine 'fremden' Daten verändern kann. Paradigma der Prozessisolation. Man könnte sonst zB buffer overflows etc forcen.

Theoretisch eh möglich (Root oder bei IPC? shared mem) Grundsätzlich sollte aber Prozess A nicht Prozess B verändern oder kaputt machen können. Auch sensible daten könnten ausgelesen werden.

Verletzung der Integrität des Prozesses

Warum kann Prozess A nicht auf Prozess B zugreifen?

- Eigener virtueller Adressraum pro Prozess
- Eigene Page Tables
- MMU erzwingt Übersetzung bei jedem Zugriff
- Kein Mapping $\rightarrow$ Page Fault
- Isolation wird von der Hardware erzwungen

Kernel- vs. User-Space Memory

- Kernel-Speicher nur im privilegierten Modus zugreifbar
- User-Prozesse dürfen Kernel-Speicher nicht lesen/schreiben
- Hardware erzwingt Zugriffsbeschränkungen
- Schutz vor Manipulation des Betriebssystems
- Viele Exploits versuchen Kernel-Speicherzugriff (zentrale Rolle bei Privilege Escalation)

Logische Ressourcenisolation (Beispiel: Linux Namespaces)

- Logische Isolation von Ressourcen
- Prozesse sehen unterschiedliche Systemansichten
- Typische Namespaces:
  - PID
  - Mount
  - Network
  - User
- Kein Ersatz für Speicherisolation, sondern Ergänzung zu Adressräumen
- Grundlage von Containern

Speicherzugriffsrechte (R/W/X)

- Speicher kann lesbar, schreibbar und/oder ausführbar sein
- Trennung von Code und Daten als Sicherheitsprinzip
- Schreib- und Ausführrechte sollten nicht kombiniert sein
- Erzwingt Kontrolle über Programmfluss
- Grundlage für moderne Exploit-Mitigation
- Durch Hardware (Page Tables) umgesetzt

Address Space Layout Randomization (ASLR)

- Zufällige Anordnung von Speicherbereichen
- Erschwert Vorhersage von Adressen für Angreifer
- Schützt vor Return-Oriented Programming (ROP)
- Wirksamkeit steigt in Kombination mit NX
- Kein vollständiger Schutz, aber wichtige Hürde
- Heute Standard in modernen Betriebssystemen

Data Execution Prevention / No-Execute (DEP / NX)

- Verhindert Ausführung von Code in Datenseiten
- Stoppt klassische Buffer-Overflow-Exploits
- Hardware-Feature moderner CPUs
- Erzwingt Trennung von Code und Daten
- Umgehung oft nur mit komplexen Techniken

Sichere Enklaven

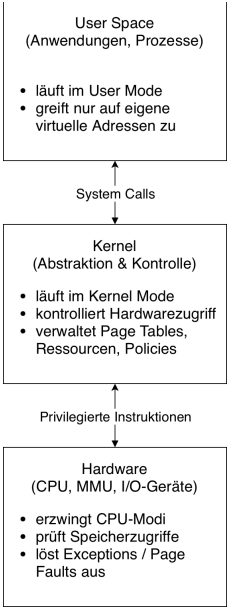
- Geschützte Ausführungsumgebungen innerhalb des Systems
- Isolieren Code selbst vor Kernel oder Hypervisor
- Beispiel: Intel SGX, ARM TrustZone
- Schutz sensibler Schlüssel und Berechnungen
- Erhöhen Vertrauenswürdigkeit in unsicheren Umgebungen
- Komplexität bringt neue Angriffsflächen

Kernel

Kernel

- Herz des Betriebssystems
- Verwaltet CPU, Speicher, Geräte und Prozesse
- Läuft mit höchsten Privilegien
- Zentraler Teil der Trusted Computing Base (TCB)
- Fehler im Kernel sind besonders kritisch
- Ziel vieler Angriffe und Exploits
- Muss klein, robust und geprüft sein

Schichtenmodell



Schichtenmodell

- Hardware als unterste Schicht
- Kernel abstrahiert Hardwarezugriffe
- User-Space nutzt Kernel-Schnittstellen (System Calls)
- Jede Schicht vertraut der darunterliegenden
- Sicherheitsverletzungen propagieren nach oben
- Klare Trennung erleichtert Analyse

Beispiel: Rootkits

- Rootkit manipuliert Kernel oder Systemfunktionen
- Versteckt Prozesse, Dateien oder Netzwerkverkehr
- Läuft oft mit höchsten Privilegien
- Sehr schwer zu erkennen
- Schutz durch Secure Boot und Integritätsprüfungen
- Prinzip: Vertrauen in Kernel absichern

Rechte & Policies

Authentifizierung vs. Autorisierung

- Authentifizierung: Wer bist du?
- Autorisierung: Was darfst du tun?
- Authentifizierung erfolgt vor Autorisierung
- Beispiele: Login vs. Dateizugriff
- Fehlerhafte Trennung führt zu Sicherheitslücken
- OS setzt beide Mechanismen zentral um

Discretionary Access Control (DAC)

- Zugriffskontrolle durch Eigentümer von Objekten
- Klassisches Unix-Rechtemodell (rwx)
- Benutzer kann Rechte weitergeben
- Flexibel, aber anfällig für Fehlkonfiguration
- Malware kann Rechte des Nutzers ausnutzen
- Standard in vielen Betriebssystemen

Demo: Linux Permissions

Mandatory Access Control (MAC)

- Zentrale, systemweite Sicherheitsrichtlinien
- Benutzer können Regeln nicht überschreiben
- Beispiele: SELinux, AppArmor
- Erzwingt strikte Isolation von Prozessen
- Erschwert Schadsoftware-Ausbreitung
- Höherer Konfigurationsaufwand

MAC – Beispiel anhand von SELinux

File Context-Regel

```
/var/www/html(/.*)?   system_u:object_r:httpd_sys_content_t:s0
```

SELinux-Regel

```
allow httpd_t httpd_sys_content_t:file { read open };
```

Least Privilege

- Prozesse erhalten nur minimal notwendige Rechte
- Reduziert Schadenspotenzial bei Kompromittierung
- Kurzlebige Privilegien bevorzugt
- Root-Rechte nur wenn unbedingt nötig
- Zentrales Prinzip sicherer Systemarchitektur
- Oft verletzt aus Bequemlichkeit

Over-privileging: Risiken

- Zu viele Rechte erhöhen Angriffsfläche
- Exploits haben größere Wirkung
- Häufige Ursache realer Sicherheitsvorfälle
- Besonders kritisch bei Netzwerkdiensten
- Verstößt gegen Least Privilege
- Oft schwer nachträglich zu korrigieren

Storage & Dateisysteme

Dateisystemrechte

- Steuern Zugriff auf Dateien und Verzeichnisse
- Lesen, Schreiben, Ausführen getrennt
- Besitzer, Gruppe und andere
- Grundlage für Vertraulichkeit und Integrität
- Fehlerhafte Rechte häufige Schwachstelle
- Zentral für Mehrbenutzersysteme

Verschlüsselung

- Schutz ruhender Daten (Data at Rest)
- Vollverschlüsselung von Datenträgern
- Schlüssel oft an Benutzer oder Hardware gebunden
- Schutz bei Diebstahl oder Verlust
- OS verwaltet Schlüsselmaterial
- Kein Schutz bei kompromittiertem System

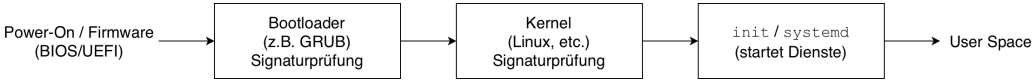
Journaling / Schutz vor Datenverlust

- Protokolliert Änderungen vor Ausführung
- Erhöht Konsistenz nach Abstürzen
- Schutz vor Datenkorruption
- Indirekter Beitrag zur Verfügbarkeit
- Kein Ersatz für Backups
- Standard in modernen Dateisystemen

Boot- & Update-Sicherheit

Bootkette

- Start mit Firmware (BIOS/UEFI)
- Laden des Bootloaders
- Initialisierung des Kernels
- Jeder Schritt vertraut dem vorherigen
- Schwachpunkt für persistente Angriffe
- Basis für Secure Boot



Secure Boot & Trusted Boot

- Secure Boot prüft Signaturen beim Start
- Verhindert Laden manipulierter Komponenten
- Trusted Boot misst Integrität für spätere Prüfung
- Root of Trust in Hardware verankert
- Schutz vor Bootkits und Rootkits
- Vertrauenskette vom Einschalten an

Updates & Supply Chain Security

- Updates schließen bekannte Sicherheitslücken
- Signierte Pakete verhindern Manipulation
- Supply-Chain-Angriffe kompromittieren Quellen
- Vertrauen in Update-Infrastruktur entscheidend
- Rollback-Mechanismen wichtig
- Automatische Updates vs. Kontrolle

Zusammenfassung

- Betriebssystem ist zentrale Sicherheitsinstanz
- Isolation von Prozessen und Speicher essenziell
- Hardware und Software greifen ineinander
- Rechte- und Privilegienmodelle entscheidend
- Sicherheit ist ein kontinuierlicher Prozess

Literaturverzeichnis

- Abraham Silberschatz, Peter B Galvin, und Greg Gagne. *Operating system concepts*.
  John Wiley & Sons, 2019
- William Stallings. *Operating systems: internals and design principles*.
  Prentice Hall Press, 2011
- Andrew Tanenbaum. *Modern operating systems*.
  Pearson Education, Inc., 2009
- Jerome H. Saltzer und Michael D. Schroeder. The protection of information in computer systems.
  In *Proceedings of the IEEE*, Band 63, Seiten 1278–1308, 1975

Vielen Dank!

https://establishing-security.at/