Security Concepts: EROS

Seminar on Trusted Computing

Winter-Semester 2008

David Hobach

Matr.-Nr.: 108006224913

8. Februar 2009

Zusammenfassung

EROS: The Extremely Reliable Operating System zeichnet sich durch seine konsequente Implementierung von Capabilities aus. Jeder Zugriff durch ein Subjekt, d.h. einen Nutzer oder ein Programm, auf ein Objekt benötigt der zugehörigen Capabilities für das Objekt. Es handelt sich bei Capabilities also um eine Umsetzung von "Discretionary Access Control". Um die Idee der Capabilities z.B. auch auf den Hauptspeicher erweitern zu können, wurde im Fall von EROS das sogenannte Diminish-Take-Modell, welches mathematisch nachweisbar keine unerwünschten Informationsflüsse zulässt, in einer Virtual Machine implementiert. Innerhalb dieser Virtual Machine läuft dann das eigentliche EROS Operating System.
Die vorliegende Seminararbeit beschreibt neben diesen Grundkonzepten weitere für EROS typische Mechanismen wie die Prozessverwaltung, Persistenz und das EROS Window System.

Inhaltsverzeichnis

1 Einleitung **3**

2 Definitionen **4**

3 Theorie: Capabilities **5**
3.1 Vor- und Nachteile 5
3.2 Modelle 6
3.2.1 Take-Grant-Modell 6
3.2.2 Diminish-Take-Modell 7

4 Praxis: Umsetzung der Capabilities in EROS **9**
4.1 EROS Virtual Machine 9
4.1.1 Objekte/“Datentypen“ 9
4.1.2 Prozesse 10
4.2 EROS Operating System 11
4.2.1 Persistenz 11
4.2.2 Prozess-Erstellung 11
4.2.3 Speicherverwaltung/Virtual Copy Spaces 12

5 EROS Window System (EWS) **13**
5.1 Notwendigkeit 13
5.2 Ziele 13
5.3 Beispiele der Implementierung 14

6 Zusammenfassung **15**

Literatur **16**

1 Einleitung

Im Jahre 1991 startete Jonathan Strauss Shapiro das Projekt "EROS" als rein akademisches Open-Source-Betriebssystem und Nachfolger des Betriebssystems KeyKOS. Zunächst entwickelte Shapiro an der University of Pennsylvania, im Jahre 2000 dann an der Johns Hopkins University als "Research Assistant Professor". [SHABIO] Die Ziele, welche er mit EROS zu erreichen versuchte, waren die Implementierung von Capabilities, Effizienz/Schnelligkeit, Unterstützung aktueller Prozessorarchitekturen (x86) und Persistenz. Diese Ziele wurden laut [SHAP99] auch erreicht. Im Jahre 2000 beteiligte sich Shapiro zudem an der Entwicklung des EROS Window System (EWS), welches komplett auf Capabilities basierend versuchte, die sicherheitskritischen Probleme von Ambiguitäten von Darstellungen für den Nutzer zu addressieren. Nach Abschluss der Arbeiten wurde 2003 allerdings klar, dass die Architektur von EROS sich nicht mit der damaligen Entwicklung der Kommunikation zwischen Prozessen (IPC) in Punkto Sicherheit vereinen ließ. [SHIP03] Daher brach Shapiro die Arbeit an EROS zugunsten des Nachfolgerprojektes "Coyotos" ab. Des Weiteren entstand 2005 ein weiterer Open-Source-Ableger von EROS namens "CapROS" mit dem Ziel, aus EROS ein stabiles System von kommerzieller Qualität zu machen. CapROS wird von der "Strawberry Development Group" unter Leitung von Charles Landau mit Budget von der amerikanischen "Defense Advanced Research Projects Agency", also dem Militär, entwickelt. Während CapROS seit 2005 nicht mehr weiterentwickelt zu werden scheint, arbeitet Shapiro offenbar noch an Coyotos[1].

Die Sicherheit von EROS und seinen Nachfolgern beruht hauptsächlich auf der Sicherheit durch Capabilities. Daher werde ich mich in dieser Ausarbeitung auf das Capability-Konzept konzentrieren und nur am Rande Konzepte wie z.B. das EWS behandeln.

[1] vgl. die entsprechenden Webseiten und SVN-Server

2 Definitionen

Capabilities: Eine Capability beschreibt die Zugriffsrechte eines Subjekts auf ein Objekt. Die Capability befindet sich im Besitz des Subjekts. Die Verwaltung der Capabilities, welche für gewöhnlich in Form von Listen implementiert werden, muss dabei geschützt durch das Betriebssystem erfolgen. Ein Nutzer kann Zugriffsrechte auf Objekte vergeben, indem er Kopien seiner Capabilities an andere Nutzer weitergibt. Das Zurückziehen dieser Capabilities sollte vom Betriebssystem auch ermöglicht werden.

Discretionary Access Control: (dt.: "Zugriffskontrolle nach freiem Ermessen") Allgemeiner Begriff für subjektbasierte Zugriffskontrollsysteme. Üblicherweise kann ein Subjekt, welches ein Objekt besitzt ("owner"), die Rechte für dieses Objekt vergeben. Oft implementiert in Form von Access Control Lists (ACLs, z.B. Windows oder Linux) oder auch Capabilities. [SADH08] Im Gegensatz zu Capabilities handelt es sich bei ACLs um eine Beschreibung der Zugriffsrechte aus Sicht der Objekte (vgl. Abb. 1: Spalten $\widehat{=}$ ACLs, Zeilen $\widehat{=}$ Capabilities).

Mandatory Access Control: (dt.: "verbindliche Zugriffskontrolle") Allgemeiner Begriff für systembasierte Zugriffskontrollsysteme. Die Zugriffskontrolle für ein Objekt liegt in diesem Fall einzig und allein beim Betriebssystem. Das Betriebssystem handelt dabei nach anfänglich festgelegten Regeln. Häufig werden innerhalb dieser Regeln mehrere Sicherheitsstufen eingeführt ("multilevel security"). [SHAP03] EROS enthält keine Methoden des mandatory access control, unterstützt sie aber theoretisch. Somit ist es entgegen des ersten Eindrucks möglich, eine Kombination aus mandatory und discretionary access control zu implementieren.

Confinement property: Falls ein Programm innerhalb eines Systems keinen nicht-authorisierten Zugriff auf Informationsflüsse erhalten kann, so erfüllt dieses System die confinement-Eigenschaft. Diese Eigenschaft ist nötig, um multilevel security implementieren zu können, und allgemein immer erwünscht. [SHAP99]

Decidability: "Entscheidbarkeit" der Sicherheit eines Betriebssystems. Ein Betriebssystem sollte stets in linearer Zeit untersuchen können, ob die confinement-Eigenschaft noch gültig ist. Falls dies möglich ist, so spricht man von "decidability". Das Take-Grant-Modell sowie das abgeleitete Diminish-Take-Modell unterstützen die Entscheidbarkeit. [SHAP03]

$$\text{ACM} = \begin{pmatrix} & \textbf{O0} & \textbf{O1} & \textbf{O2} & \dots & \textbf{Om} \\ \textbf{S0} & \{r\} & \{r,w\} & \{w,e\} & \vdots & \{r,w,e\} \\ \textbf{S1} & \{r,e\} & \{e\} & \{w,e\} & \vdots & \{w,e\} \\ \textbf{S2} & \{w\} & \{r,w\} & \{r,w\} & \vdots & \{r,w,e\} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \textbf{Sn} & \{r,w\} & \{r\} & \{r\} & \dots & \{r,e\} \end{pmatrix}$$

Abbildung 1: Beispiel einer Access Control Matrix

3 Theorie: Capabilities

[SHAP99] nennt folgende von Capabilities unterstützte Eigenschaften:

Least privilege: Jedes Programm soll nur so wenige Rechte wie möglich bekommen.

Selective Access Right Delegation: Rechte sollen von einem Programm an ein zu ihm gehörendes Unterprogramm weitergegeben werden können. Diese Rechte sollen auch weiter beschnitten werden können.

Right Transfer Control: Der Akt der Rechtevergabe soll kontrolliert werden können/darf nur über einen authorisierten Kanal erfolgen.

Information Transfer Control: Der Informationsfluss muss kontrolliert werden können.

Endogenous Verification: Das System selbst muss verifizieren können, ob gewisse Restriktionen/Regeln bezüglich "Right Transfer Control" und "Information Transfer Control" eingehalten werden.

Auch wenn diese Eigenschaften von Capabilities unterstützt werden, so bleibt es die Aufgabe des Betriebssystems, diese auch auszuschöpfen. Will man dem Betriebssystem auch nicht vetrauen, so besteht die Möglichkeit, diesem die Benutzung der Capabilities durch Implementierung einer Virtual Machine aufzuerlegen. Das ist bei EROS der Fall.

3.1 Vor- und Nachteile

Im Bereich der Discretionary Access Control-Systeme gibt es als direkte Konkurrenz von Capability-basierten Systemen vornehmlich die Systeme, welche auf Access Control Lists (ACLs) basieren wie z.B. Windows oder Linux. Warum diese die obigen Eigenschaften nicht erfüllen, soll kurz erklärt werden (Szenario: Nutzer A und Nutzer B eines Betriebssystems besitzen je einige Dateien): [SHAP99]

Least privilege: Jedes Programm besitzt bei ACL-basierten Systemen die Rechte seines Nutzers.

Selective Access Right Delegation: Rechte können nicht reduziert werden, wenn sie an Unterprogramme weitergegeben werden.

Right Transfer Control: Nutzer A kann jedem beliebigen Nutzer, z.B. Nutzer B, Rechte auf sicherheitskritische Daten o.ä. geben.

Information Transfer Control: Dadurch dass Rechte beliebig vergeben werden können, kann der Informationsfluss auch nicht kontrolliert werden.

Endogenous Verification: Da die bisherigen Eigenschaften nicht gegeben sind, würde eine Verifikation ihrer Existenz immer fehlschlagen.

Warum diese Eigenschaften von Capability-basierten-Systemen erfüllt werden, kann erst nach der Vorstellung der Modelle verstanden werden.
Ein Nachteil Capability-basierter Systeme liegt wohl in dem organisatorischen Aufwand: Pro Nutzer muss für jedes Objekt innerhalb des Betriebssystems eine Liste aufrecht erhalten werden, in welcher sämtliche Capabilities des Nutzers enthalten sind. Da ein Nutzer auf ein Objekt mehrere Capability-Typen besitzen kann, ergibt sich ein beträchtlicher Aufwand

in Speicher und Rechenzeit. Bei ACL-basierten Systemen werden die Zugriffsrechte dagegen meist in wenigen Bits direkt bei der Datei gespeichert, so dass nur dann ein Aufwand entsteht, wenn ein Zugriff auf die Datei erfolgt. Den Geschwindigkeitsvorteil ACL-basierter Systeme versucht EROS durch Persistenz auszugleichen.

3.2 Modelle

Die beiden für das Thema "Capabilities" ausschlaggebenden Modelle sind das Take-Grant-Modell sowie das Diminish-Take-Modell, eine vom Take-Grant-Modell abgeleitete Version. Sie lassen sich als gerichtete Graphen aus Subjekten und Objekten darstellen. Subjekte (Nutzer, Prozesse o.ä.) werden als Kreise dargestellt, Objekte als Quadrate. Besitzt ein Subjekt eine oder mehrere Capabilities auf ein Objekt oder Subjekt, so wird dies durch eine gerichtete Kante dargestellt, an welcher die Capabilities aufgeführt werden.
Um Sicherheitsaspekte der Modelle untersuchen zu können, muss man davon ausgehen, dass ein Subjekt jede mögliche Schwachstelle, die es im Modell gibt, auch ausnutzt. [SHAP03]

3.2.1 Take-Grant-Modell

Beim Take-Grant-Modell unterscheidet man 4 Capability-Typen: {read,write,take,grant} bzw. {r,w,t,g}. Jeweils in die Richtung der Kante erlaubt "read" das Lesen von Daten, "write" das Schreiben, "take" das Kopieren von Capabilities und "grant" das Übergeben von Capabilities. Offensichtlich gilt Transitivität, was im Falle von take vom Modell aus erwünscht ist (Subjekt A bekommt über take auf Subjekt B alle Rechte von B auf Objekt C, "de jure access", vgl. Abb. 2), im Falle von read bei Kollaboration von Subjekten jedoch z.B. auch beachtet werden muss (Subjekt A hat bloß eine read-Capability auf Subjekt B, Subjekt B eine read-Capability auf C → wenn A und B zusammen arbeiten, so kann A von C lesen, "de facto access", vgl. Abb. 2). Dass ein Subjekt innerhalb des Modells keine Rechte auf sich selbst vergeben darf, erscheint auch klar.

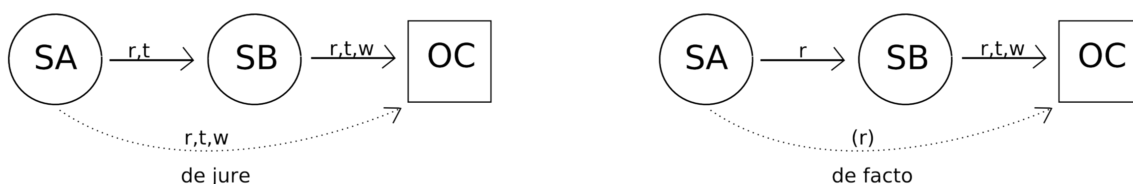

Abbildung 2: Transitivität

Nun soll betrachtet werden, inwiefern das Modell die oben genannten Eigenschaften unterstützt:

Least privilege: Erstellt ein Subjekt/Prozess P1 ein anderes Subjekt/einen anderen Prozess P2, so besitzt P2 zunächst keinerlei Capabilities, d.h. Rechte. Nur P1 besitzt Rechte auf P2.

Selective Access Right Delegation: P1 kann daraufhin via Grant beliebig viele bzw. wenige Capabilities an P2 weitergeben.

Right Transfer Control: Zwei Subjekte ohne Take- oder Grant-Capabilities besitzen keine Möglichkeit, Rechte auf Objekte des anderen zu bekommen, d.h. wenn gewollt, kann der Rechtetransfer über diese beiden Capabilities kontrolliert werden.

Information Transfer Control: Analog können zwei Subjekte ohne jegliche Capabilities auf den jeweils anderen keinerlei Informationen austauschen.

Endogenous Verification: Der Akt der Rechtevergabe sowie der Informationsfluss sind in dem Maße kontrollierbar, dass er entweder möglich ist oder eben nicht. Allerdings gibt es Szenarien, in denen ein begrenzter Informationsfluss möglich sein soll. Ein Beispiel (vgl. Abb. 3): Ein Subjekt S1 besitzt sämtliche Capabilities bezüglich eines anderen Subjekts S2 und dessen Untersubjekten oder -objekten SU. Möchte es nun einem dritten Subjekt S3 lesenden Zugriff auf S2 *und* den zugehörigen Untersubjekten/objekten gewähren, so besteht diese Möglichkeit mit einer Read-Capability von S3 auf S2. Zusätzlich wird aber noch eine Take-Capability von S3 auf S2 benötigt, da der Zugriff auf die untergeordneten Subjekte oder Objekte von S2 möglich sein soll. Damit hat S3 aber faktisch dieselben Rechte wie S2, was von S1 gerade nicht gewünscht war, d.h. die in der Definition so allgemein formulierten "gewissen Restriktionen" (vgl. Abschnitt 3) konnten hier nicht eingehalten werden. Folglich unterstützt das Take-Grant-Modell diese letzte gewünschte Eigenschaft nicht, wenn gewisse Zugriffsformen möglich sein sollen.

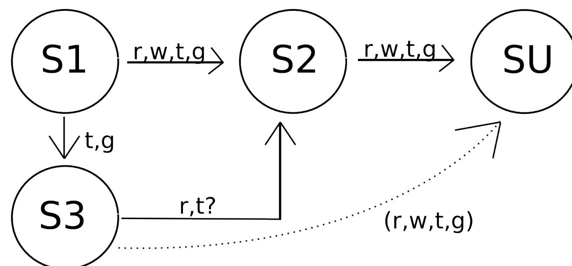

Abbildung 3: Fehlerhaftes Teilen

Das Take-Grant-Modell weist zwei Schwächen auf: Erstens ist es einem Subjekt nicht möglich, Teilgraphen mit anderen Subjekten zu teilen, ohne diesen volle Rechte auf diese Teilgraphen zu geben (s.o.). Zweitens wird in der Praxis häufig die Möglichkeit des indirekten Zugriffs benötigt, d.h. es soll z.B. beim Anwenden von Capabilities auf Hauptspeicherzugriffe unmöglich sein, eine Speicherzelle zu schreiben, in welcher eine Adresse steht, welche auf eine zu beschreibende Speicherzelle verweist. Kurz: Es ist mit dem Take-Grant-Modell unmöglich, indirekt zu lesen oder zu schreiben. [SHAP99, SHAP03]

3.2.2 Diminish-Take-Modell

Um die beiden Schwächen des Take-Grant-Modells zu beheben, wurde das Diminish-Take-Modell entwickelt [SHAP99]:
Zunächst wurde zu jeder Capability {r,w,t,g} ein indirektes Analogon eingeführt - {ri,wi,ti,gi}.
Mathematisch ausgedrückt gilt beispielsweise für ri:
($S \stackrel{\alpha}{\rightarrow} O$ heiße: Subjekt S besitzt Capability α auf Objekt O.)

$$S \stackrel{ri}{\rightarrow} O_0, \quad O_i \stackrel{ri}{\rightarrow} O_{i+1} \quad \forall i \quad 0 \leq i < n, \quad O_n \stackrel{r}{\rightarrow} O \Leftrightarrow S \stackrel{r}{\rightarrow} O$$

In Worten: Hat ein Subjekt S über beliebig viele Objekte indirekten Lesezugriff auf ein Objekt, welches direkten Lesezugriff auf ein Objekt O hat (vgl. Abb. 4), so hat auch S

Lesezugriff auf O. Die Objekte dazwischen kann S *nicht* lesen, d.h. der indirekte Lesezugriff erlaubt kein Lesen des Inhalts, nur der Referenz auf das nächste Objekt.

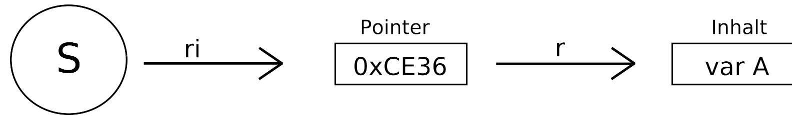

Abbildung 4: indirektes Lesen

Des Weiteren führte man zwei weitere Capabilties namens "dimtake" und "dimgrant" {dt,dg} ein. Besitzt ein Subjekt eine dt-Capability auf ein Objekt, so kann es eine Menge von Capabilities α vom Objekt kopieren, wenn auf diese vorher der sogenannte Diminish-Operator ausgeführt wurde. Genauso kann ein Subjekt mit dg-Capability auf ein Objekt seine Capabilities α auf ein anderes Objekt O nur nach Anwendung des diminish-Operators weitergeben. Jener ist folgendermaßen definiert:

$$diminish(\stackrel{\alpha}{\rightarrow} O) = \stackrel{\alpha \cap \{r,ri,dt,dti\}}{\rightarrow} O$$

Soll heißen: Es können nur die Rechte {r,ri,dt,dti} erhalten oder weitergegeben werden. Es ist einem Subjekt damit möglich, anderen Subjekten lesenden Zugriff auf Teilgraphen zu gewähren. [SHAP03]

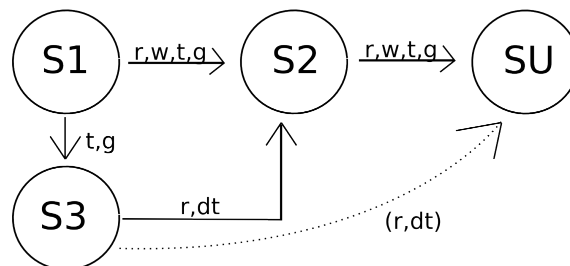

Abbildung 5: Korrektes Teilen

So zeigt beispielsweise Abb.5 ein Subjekt S1, welches einem anderen Subjekt S3 lesenden Zugriff auf S2 *und* die S2 untergeordneten Subjekte SU (z.B. Kinderprozesse) gewähren möchte. Dies ist nun dank des diminish-Operators durch die Vergabe einer dt-Capability möglich.

4 Praxis: Umsetzung der Capabilities in EROS

Das “Extremely Reliable Operating System“ setzt sich aus einer Virtual Machine und dem eigentlichen Betriebssystem in dieser VM zusammen (vgl. Abb. 6).

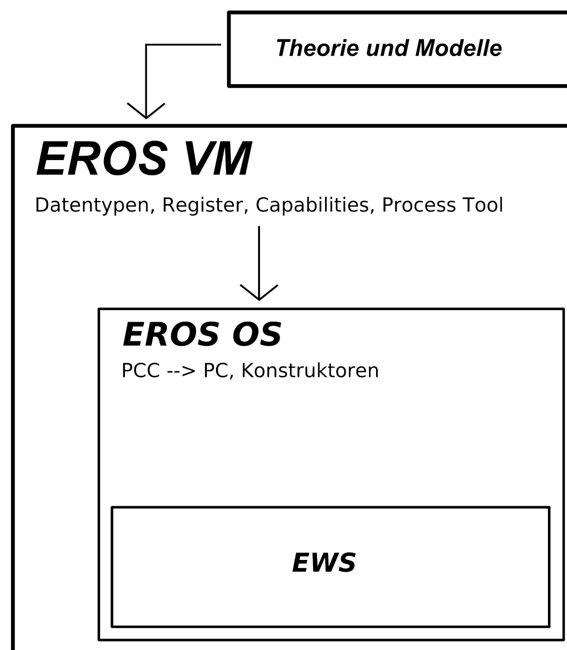

Abbildung 6: Gesamtüberblick

4.1 EROS Virtual Machine

Die Notwendigkeit einer Virtual Machine wird sehr schnell deutlich, wenn man bedenkt, dass Capabilities ein reines Software-Produkt sind und sie damit auch nur auf Software-Ebene erzwungen werden können. Will man einem Betriebssystem aber nicht vertrauen, so besteht - wie bei EROS gemacht - die Option, sicherheitskritische Aspekte des Kernels in eine VM auszulagern. Somit wird das Betriebssystem quasi vor “vollendete Tatsachen“ gestellt: Bei jedem Zugriff auf ein Objekt überprüft die VM die dafür nötigen Capabilities. Shapiro geht sogar so weit, dass er von einem “Aufruf“ (invocation) der Capability spricht; er stellt die Capability also über das eigentliche Objekt, da nur die Capability den Zugriff gewährt. Die EROS VM stellt durch eine logische Unterteilung des Speichers in Teile zum Speichern von Capabilities und Teile zum Speichern von Daten sicher, dass Capabilities unter keinen Umständen aus Daten “generiert“ oder auf irgendeine andere Weise gefälscht werden können. [SHAP99]

4.1.1 Objekte/“Datentypen“

Die Unterscheidung von Capabilities und gewöhnlichen Daten spiegelt sich vor allem in der Implementierung der Datentypen durch die EROS VM für das Betriebssystem wider: [SHAP99]

Number: Ein einfacher Datentyp, der eine positive Zahl zwischen 0 und $2^{96} - 1$ enthalten muss. Interessant ist einzig die Tatsache, dass selbst für diesen Datentyp eine Zugriffskontrolle über Capabilities erfolgt, so dass z.B. sicherheitskritische Registerwerte gespeichert werden können.

Data Page: Enthält eine feste Anzahl von Bytes an Daten. Bei Besitz der zugehörigen "page capability" können einige Basisoperationen auf die Data Page wie beispielsweise "data load" oder "data store" ausgeführt werden. Es besteht die Möglichkeit einer read-only Data Page.

Capability Page: Capability Pages enthalten Capabilities von der Größe einer Data Page. Wichtig ist hierbei, dass sich die Basisoperationen auf die Capability Pages, auf welche nur bei Besitz sogenannter "Capage Capabilities" zugegriffen werden kann, von denen der Data Pages unterscheiden, so dass auf Capabilities nicht wie auf Daten zugegriffen werden kann. Auch hier besteht die Möglichkeit, eine read-only Capability Page zu erstellen. Dabei wurde auch an den Fall gedacht, dass Capability Pages weitere Capabilities enthalten können, mit welchen ggf. auch nur lesend zugegriffen werden können soll. Daher ist es möglich, solche Capabilities vorher zu transformieren ("weaken transformation"), so dass sie nur noch lesenden Zugriff erlauben.

Node: Bei einem "Node" handelt es sich um einen Array fester Größe, der aus Capabilities besteht. Mit einer "Node Capability" werden jedoch andere Operationen zur Verfügung gestellt als mit einer Capage Capability, da Nodes hauptsächlich benutzt werden, um Capabilities auf Data Pages, Capability Pages oder andere Nodes im Hauptspeicher abzuspeichern, d.h. ihr Einsatzgebiet ist sehr speziell.

Abb. 7 gibt einen Überblick über die von der Virtual Machine implementierten Datentypen und deren Zusammenspiel am Beispiel eines Adressraums eines Prozesses im Hauptspeicher. Die Pfeile sind in diesem Fall als Capabilities zu verstehen, die in dem übergeordneten Datentyp gespeichert werden.

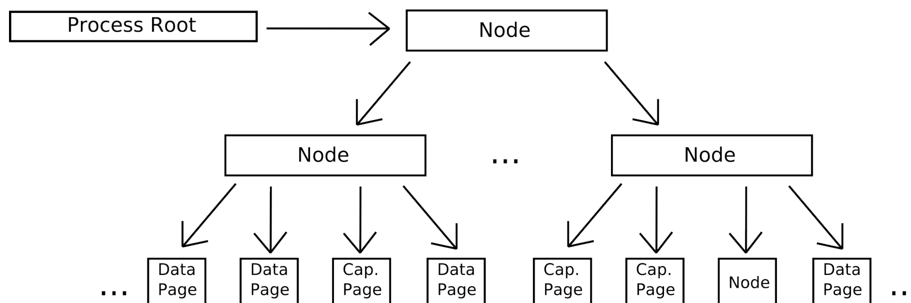

Abbildung 7: Adressraum eines Prozesses

4.1.2 Prozesse

Ein EROS-Prozess wird über einen Satz von Registern, einer Node-Capability auf den ihm zugehörigen Adressraum sowie einer "schedule capability" beschrieben, auf die hier nicht näher eingegangen werden soll.
Der Registersatz setzt sich aus Datenregistern und Capability-Registern zusammen. Die

Datenregister wiederum sind die Prozessorregister der Hardware sowie einige virtuelle Register mit Flags bezüglich des Zustandes des Prozesses etc. Die 32 Capability-Register werden nicht in einem einzigen Node verwaltet, sondern in einer Baumstruktur: Der oberste Knoten (node) wird dabei als "Process Root" bezeichnet. Er enthält die Node Capability auf den dem Prozess zugeordneten Adressraum (vgl. Abb. 7), eine Node Capability auf den "registers annex", der zum Prozess gehörende Daten enthalten kann, sowie eine Node Capability auf den "capabilities annex", der die Capabilities auf die besagten 32 Capability-Register enthält.
Bei der Capability auf den Prozess-Wurzelknoten ("process root node") handelt es sich an sich um eine gewöhnliche Node Capability, die jedoch mithilfe des innerhalb der VM implementierten "Process Tool" zu einer Process Capability aufgewertet wird. Dies hat einerseits Performance-Gründe, andererseits ist es auch nötig, da gewöhnliche Node Capabilities beispielsweise nicht die Zugriffsmöglichkeit bieten, um Unterprozesse zu kreieren. Weiterhin bietet das "Process Tool" Methoden an, um festzustellen, welche Unterprozesse zu einem übergeordneten Prozess gehören. [SHAP99]

4.2 EROS Operating System

Das EROS OS bestand im Jahre 1999, dem Jahr seiner erstmaligen Dokumentation durch [SHAP99] nur aus einem "trusted storage manager", d.h. einem Mechanismus, mit Festplattenspeicher umzugehen, Mechanismen zur Prozesserstellung sowie Methoden zur Fehlerbehandlung. Mechanismen wie Nutzerauthentifikation, Netzwerkprotokolle oder einige Anwendungsprogramme fehlten. Selbst die Implementierung des Terminals sah Shapiro noch nicht als ausgereift an. Daher beschränke ich mich im Folgenden auf einige wenige Details des EROS OS.
Im Jahre 2004 war dagegen sogar ein Window System implementiert, welches separat behandelt werden soll (s. Abschnitt 5), da nur ein beschränkter Zusammenhang zum Thema "Capabilities" besteht.

4.2.1 Persistenz

Um das Sicherheitsproblem der Initialisierung eines Betriebssystems durch sich selbst, also aus einer eventl. unsicheren Umgebung heraus, zu umgehen, ist EROS persistent. Soll heißen: Es existiert nur ein anfängliches, von Shapiro konfiguriertes Image des Betriebssystems, welches alle Zustände der Register, die Capabilities etc. beschreibt. Dieses wird von der Virtual Machine, d.h. aus einer sicheren Umgebung heraus, gestartet. Das erklärt auch die Abwesenheit von Security Policies, Nutzerabfragen zur Vergabe von Capabilities o.ä. - derartige Konfigurationen wurden ja "von Hand" vorgenommen. [SHAP99]
Um trotzdem Änderungen am Betriebssystem zur Laufzeit zuzulassen, wird das komplette OS in einem gewissen Intervall auf die Festplatte geschrieben (snapshot). Um sicherzustellen, dass kein Fehler durch einen Hardware-Reset entsteht, geschieht dies asynchron und das geschriebene Image wird erst dann als gültig markiert, wenn der Schreibvorgang abgeschlossen wurde. Im Falle eines Hardware-Fehlers kehrt das OS also zu dem letzten gültigen Zustand zurück.

4.2.2 Prozess-Erstellung

Das OS bietet zwei Möglichkeiten, um einen Prozess zu erstellen: Durch einen "Process Creator" oder einen "Constructor".

Bei dem "Process Creator" handelt es sich um die einfachere Variante: Process Creators sind die einzigen Objekte im EROS OS, die eine Capability auf das Process Tool der VM besitzen und bieten zunächst schlichtweg die Methoden des Process Tools innerhalb des Betriebssystems an. Zusätzlich besitzt jeder Process Creator eine "start capability", auch "brand" genannt, welche er den von ihm erstellten Prozessen als Identifikationsmerkmal zuweist. Um jeden Prozess eindeutig identifizieren können, ist folglich ein Process Creator (PC) pro Prozess nötig. Daher existiert der Process Creator Creator (PCC), welcher für die Erstellung der PCs zuständig ist. Der Nachteil dieser schlichten Methode der Prozess-Erstellung liegt darin, dass keinerlei Initialisierungsvorgänge durchgeführt werden, dem Prozess u.a. auch kein Hauptspeicher zum Arbeiten zugewiesen wird. Weiterhin kann die confinement-Eigenschaft (vgl. Abschnitt 2: Definitionen) des Prozesses nicht nachgewiesen/überprüft werden.
Daher wurden Konstruktoren ("constructors") implementiert, die zunächst mit den Capabilities des zukünftigen Prozesses "gefüllt" werden und daraufhin die Initialisierung des Prozesses vornehmen. Zu diesem Zweck benutzt ein Konstruktor wieder je einen Process Creator. Da es nach dem Erstellen des Prozesses durch den Konstruktor nicht mehr möglich ist, Capabilities hinzuzufügen, kann der Konstruktor, welcher ja auch im Besitz der Capabilities des Prozesses ist, permanente Aussagen über die confinement-Eigenschaft des Prozesses machen. Diese wird als gegeben angesehen, wenn der Prozess keine unauthorisierte Möglichkeit besitzt, Daten (eventl. indirekt) zu schreiben oder Capabilities auf Konstruktoren existieren, die wiederum behaupten, die confinement-Eigenschaft zu unterstützen. Das Besondere ist, dass die Entscheidung, ob die confinement-Eigenschaft eingehalten wird, in linearer Zeit nur unter Benutzung lokaler Informationen getroffen werden kann. Da der Konstruktor allerdings Zugriff auf den Adressraum des von ihm verwalteten Prozesses hat, könnte der Prozess rein theoretisch Informationen in einen Bereich schreiben, der vom Konstruktor ausgelesen werden kann. Der eben genannte und auch so eingesetzte Test der confinement-Eigenschaft würde also immer fehlschlagen. Jenes Problem wurde durch den Einsatz von "Virtual Copy Spaces" behoben. [SHAP99, SHAP03]

4.2.3 Speicherverwaltung/Virtual Copy Spaces

Die Verwaltung des Hauptspeichers läuft in der VM ab und wurde bereits kurz erwähnt: Sie erfolgt als Baumstruktur in Form von Node Capabilities, die Capabilities auf Data oder Capability Pages enthalten.
Der einzige Mechanismus, welcher auf Seiten des OS bezüglich der Speicherverwaltung durchgeführt wird, ist die Erstellung von "virtual copy spaces". Immer dann, wenn ein Prozess nur lesenden Zugriff auf Daten erhalten hat (z.B. aufgrund der confinement-Eigenschaft), aber schreibenden Zugriff benötigt, wird beim ("imaginären") Schreiben eine Kopie der benötigten Daten erstellt, auf welche dann geschrieben wird. Somit ist es z.B. möglich, das o.g. Problem zu umgehen. [SHAP99, SHAP03]

5 EROS Window System (EWS)

Das im Jahre 2004 fertig gestellte EROS Window System soll im Folgenden kurz angesprochen werden:

5.1 Notwendigkeit

Moderne Nutzer-Frontends sind aus Gründen der Bedienbarkeit meist grafikorientiert. Folglich werden auch sicherheitskritische Operationen, wie z.B. das Abfragen von Passwörtern, über grafische Eingabefelder gelöst. Selbst wenn man von einer fehlerfreien Implementierung des Window Systems (WS) ausgehen würde, so kann auf keinen Fall davon ausgegangen werden, dass Nutzer sich fehlerfrei verhalten, d.h. nicht unbewusst schädliche Programme (Malware) o.ä. ausführen. Somit ist auch die Sicherheit eines Betriebssystems wie EROS gefährdet, wenn ein Trojaner ein Passwort-Eingabefenster nachahmt und der Nutzer gutgläubig sein Passwort eingibt.
Weiterhin entstehen über ein WS viele Möglichkeiten, die es unmöglich machen können, den Informationsfluss nachzuvollziehen oder gar mithilfe von policies zu kontrollieren (Bsp.: Copy & Paste). [SVNC04]

5.2 Ziele

Die Ziele der Implementierung des EWS-Display-Servers waren von Anfang an klar abgegrenzt:

- Isolation von Nutzer-Sessions und deren Auswirkungen auf den Server
- Unterstützen von mandatory access control policies
- Kommunikation zwischen Applikationen über den Display-Server sollte nur mit expliziter Erlaubnis des Nutzers geschehen
- Minimalisierung von Codelänge, algorithmischer Komplexität und Resourcentypen, um eine Evaluierung (Common Criteria) zu ermöglichen; auch die Kommunikation zwischen Prozessen ("Interprocess Communication", IPC) sollte in Menge und Art (d.h. wie sie möglich ist) beschränkt werden
- Minimalisierung der Anzahl der Antworten durch den Display-Server; das Ziel dieser Antworten sollte auch immer nur eine Applikation sein, die Antwortzeit dabei konstant (Maßnahme gegen "covert channels")
- Erhalten des allgemein gewohnten "Look and Feel"

Diese Ziele wurden laut [SVNC04] mit nur 4500 Zeilen Code bis auf 3 Ausnahmen, in welchen Rücksicht auf die Benutzbarkeit des Display-Servers genommen wurde, erreicht.

5.3 Beispiele der Implementierung

Da hier nicht auf die Details der Implementierung eingegangen werden kann, möchte ich nur einige Beispiele aus [SVNC04] nennen:

Copy & Paste: In gewöhlichen Window Systems wird bei jeder "Copy"-Operation *jede* beim Server registrierte Applikation darüber informiert, dass der zugehörige Puffer gefüllt wurde. Bei einer "Paste"-Operation kann dann die zugehörige, aber auch jede andere Applikation den Puffer leeren. Der Grund für diese Vorgehensweise liegt darin, dass für das WS nicht feststellbar ist, für welche Applikation das "Paste" durchgeführt wurde. Bei einer Tastaturkombination wie Strg-C/Strg-V wäre dies zwar noch möglich, spätestens bei applikationsabhängigen Kopieren-/Einfügen-Buttons jedoch nicht mehr ("Paste"-Ereignis durch das Programm).
Das EWS dagegen gibt das "Copy"-Ereignis gar nicht und das "Paste"-Ereignis nur an die Ziel-Applikation weiter. Diese kann bei einer Tastaturkombination einfach ermittelt werden, da es sich um ein Ereignis durch den Nutzer handelt (s.o.); andere "Paste"-Ereignisse sind nur erlaubt, wenn eine Applikation ein vom Server definiertes unsichtbares Fenster verwendet, über welchem sich das Aussehen der Maus ändert, um eine mögliche "Paste"-Operation anzuzeigen. Die Position dieser Fenster muss natürlich vom Server verfolgt werden.

Titelleiste: Es ist ebenfalls usus, die Darstellung der Titelleiste, welche normalerweise die Optionen Minimieren, Maximieren und Schließen bietet, der Applikation zu überlassen. Dass dies unerwünschte Folgen haben kann, sieht man derzeit an der auf vielen Internetseiten vorhandenen Javascript-Overlay-Werbung, die auch nach Klick auf den mit Javascript dargestellten Schließen-Button noch die Werbeseite im Browser öffnet. Das EWS verwaltet deshalb die Operationen der Titelleisten sämtlicher laufender Applikationen, so dass der Nutzer sicher gehen kann, dass seine Instruktionen auch wie gewünscht ausgeführt werden. Des Weiteren erlaubt der EROS Display Server nur eine vordefinierte Schrift zur Darstellung der Titelleiste, um einer Applikation, die in der Lage ist, systemweit die Schrift zu verändern, nicht zu ermöglichen, darüber den Inhalt der Titelleisten anderer Programme zu modifizieren.

Kennzeichnung von Fenstern: Um mandatory access controls grundsätzlich zu unterstützen, wurden farbliche Markierungen zur Unterscheidung der Sicherheitsstufen von Fenstern implementiert. Auch das Fenster, welches aktuell im Vordergrund ist, wird speziell markiert.

Es gilt zu beachten, dass EROS wegen seines Einsatzes von Capabilities viele Wünsche für das WS schon von sich aus unterstützt, so z.B. als "trusted path" einen vertrauenswürdigen Dialog zum Abspeichern von Dateien ("Save As"): Weil nur dieser Dialog die nötigen Capabilities besitzt, kann er von keiner anderen Applikation gefälscht werden.

6 Zusammenfassung

Insgesamt fällt bei der Betrachtung von EROS auf, dass viele Fragen nicht beantwortet werden, beispielsweise: Wie verläuft die Authentifikation von Nutzern? Wie werden neue Nutzer erstellt? Gibt es so etwas wie "policies", nach denen Rechte vergeben werden? Gibt es einen Urprozess (à la "init" in Linux)?
Einerseits ist es schwer, diese Fragen zu beantworten, weil EROS ein "Ein-Mann-Projekt" von Jonathan S. Shapiro bezüglich Implementierung und Dokumentation war und die Dokumentation leider nur in Form von wissenschaftlichen Abhandlungen vorliegt, die auch nur einen Zustand (Jahr) des Betriebssystems widerspiegeln. Andererseits umgeht EROS durch Einführung von Persistenz viele der o.g. Probleme: Sämtliche Rechte für Nutzer und Prozesse wurden von Shapiro vermutlich einmalig statisch vergeben, die Nutzer einmalig erstellt. Da EROS als reiner Forschungsprototyp anzusehen ist, darf eine solche Form der "Umgehung" von Problemen auch nicht verwundern: Shapiro wollte zunächt nur beweisen, dass der Einsatz von Capabilities durchaus auf effiziente Art und Weise geschehen kann. Wie Capabilities vergeben werden müssen, ohne den Benutzer übermäßig zu belästigen, interessierte ihn dagegen vorläufig nicht, auch wenn er in seinen Schriftstücken stets die theoretische (nicht implementierte) Unterstützung von mandatory access control-Mechanismen betont.
Schritte hin zu einem in der Praxis benutzbaren Betriebssystem geschahen mit der Entwicklung des EWS, einem grafischen Frontend für eventuelle Nutzer. Das Problem der Persistenz jedoch blieb z.B., weswegen es das erklärte Ziel des Ablegers CapROS war, EROS praxistauglich zu machen. Dieses Vorhaben scheint (bislang) gescheitert zu sein.
Dass Capabilities jedoch in einer Vielzahl aktueller Sicherheitsprobleme grundlegend helfen könnten, erscheint sofort klar, wenn man z.B. die Thematik Trojaner/Viren betrachtet. Ohne die nötigen Capabilities kann ein Trojaner weder Informationen ausspähen, noch weitergeben; ein Virus kann das System nicht zerstören. So gesehen ist und bleibt das Thema aktuell.

Literatur

[SHAP99] Jonathan S. Shapiro. EROS: A Capability System. A dissertation in computer and information science. 1999.

[SHAP03] Jonathan S. Shapiro. The Practical Application of a Decidable Access Model. SRL Technical Report. November 2003.

[SVNC04] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, David Chizmadia. Design of the EROS Trusted Window System. 2004.

[SHHA02] Jonathan S. Shapiro, Norm Hardy. EROS: A Principle-Driven Operating System from the Ground Up. 2002.

[SHWE00] Jonathan S. Shapiro, Sam Weber. Verifying the EROS Confinement Mechanism. 2000.

[SHSF99] Jonathan S. Shapiro, Jonathan M. Smith, David J. Farber. EROS: a fast capability system. 1999.

[SHIP03] Jonathan S. Shapiro. Vulnerabilities in Synchronous IPC Designs. 2003.

[SADH08] Prof. Dr.-Ing. Ahmad-Reza Sadeghi. Vorlesung: Betriebssystemsicherheit. Kapitel 2. Wintersemester 2008

[SHABIO] Jonathan S. Shapiro. Biographie. `http://srl.cs.jhu.edu/~shap/`

[HOME] EROS Hompage. `http://www.EROS-os.org/`