JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Rollenbasierte Zugriffskontrollsysteme und deren praktische Umsetzbarkeit auf Betriebssystemebene unter Microsoft Windows 7 und Windows Server 2008

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

NETZWERKE UND SICHERHEIT (911)

Eingereicht von:
Joel Valek, B.Sc.

Angefertigt am:
Institut für Informationsverarbeitung und Mikroprozessortechnik

Beurteilung:
Mühlbacher Jörg R., o. Univ. Prof. Dr.

Mitwirkung:
Christian Praher, DI

Linz, April 2010

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 19. April 2010

Joel Valek

Danksagung

Ich möchte mich herzlich bei allen Leuten und Institutionen, die mich bei dieser Arbeit und dem vorausgehenden Studium unterstützt haben, bedanken.

In erster Line geht mein Dank an die Voest-Alpine Stahlstiftung, ohne deren finanzelle Unterstützung ich heute nicht an diesem Punkt angelangt wäre. Weiters geht mein Dank an das Institut für Mikroprozessortechnik, das mein Interesse an Netzwerk- und Sicherheitstechnischen-Aspekten motiviert und mir in dieser Hinsicht eine hervorragende Ausbildung vermittelt hat, sowie Herrn Univ.-Prof. Mühlbacher und Herrn DI Praher, die mich bei der Diplomarbeit betreut haben. Zu guter letzt möchte ich meiner Familie für die mentale Unterstützung in den vergangen 5 Jahren danken.

Abstract

Role Based Access Control Systems (RBAC) existieren als Variante der Zugriffskontrollsysteme neben den Discretionary Access Control Systems (DAC) und Mandatory Access Control Systems (MAC) und finden oft im Bereich der Enterprise Software ihre Anwendung. Grund hierfür ist die bessere Abbildbarkeit der Unternehmens-Strukturen und Verantwortlichkeiten im Bezug auf die Zugriffsregelung für Ressourcen bzw. Systemobjekte.

Grundlage der Masterarbeit ist es, die Entwicklung der RBAC Systeme von den ersten Ansätzen, bis hin zum ANSI/INCITS Standard [8] zu evaluieren. Dabei sind eine detaillierte Analyse des Standards und seiner Konzepte, ebenfalls Bestandteil, wie auch ein umfassender Überblick über die Vielfalt der Zugriffskontrollsysteme und das jeweilig sinnvolle Einsatzgebiet.

Das Betriebsystem Windows ist seit je her mit einem DAC ausgestattet. Allerdings ist dieses zusätzlich mit einem Gruppenkonzept verbunden. In Kombination damit ergibt sich bereits eine erste Ähnlichkeit zu RBAC. Allerdings ist das DAC in seiner vorgesehenen Verwendung viel zu statisch, um den Anforderungen an ein RBAC-System gerecht zu werden und die Rechte-Vergabe erfolgt durch den Besitzer der Objekte, also benutzerbestimmt, was dem RBAC Konzept grundlegend widerspricht. Ziel dieser Arbeit ist, zu evaluieren, ob es möglich ist, das DAC in Windows 7 so zu verwenden bzw. umzugestalten, dass es das Konzept eines RBAC Systems implementiert. Es wird versucht ein RBAC0 System nach [13] umzusetzen.

Ein Rechteverwaltungs-Tool, der sogenannte Autorization Manager von Microsoft, wurde bereits in einer vorhergehenden Praktikumsarbeit [14], hinsichtlich seiner Verwendbarkeit für ein RBAC-System evaluiert. Allerdings ist die Lösung über den MS Authorization Manager nur auf Applikationsebene anwendbar. Das darunter liegende Zugriffskontrollsystem des Betriebsystems bleibt dabei DAC. Die angestrebte Lösung sollte also auf OS-Ebene und nicht lediglich auf Applikationsebene funktionieren.

Da das Konzept für Enterprise Software gedacht ist, ist es natürlich unerlässlich, dass die Lösung auch unter Verwendung von Domaincontrollern im Active Directory funktio-

niert. Es wird dazu das NTFS-Dateisystem hinsichtlich der Rechte-Vergabe genauestens analysiert und die Konzepte, die im Active Directory zur Verfügung stehen, in Betracht gezogen, um die Rechteverwaltung entsprechend umzugestalten bzw. anzupassen. Zum Schluss wird ein im Zuge dieser Arbeit implementierter Prototyp namens *RBACSystem* beschrieben, der das geforderte RBAC0 Verhalten unter Windows 7 und Server 2008 umsetzt.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACL Access Control List

AD Active Directory

ANSI American National Standards Institute

API Application Programming Interface

CC Common Criteria

CLR Common Language Runtime

CPU Central Processing Unit

CTCPEC Canadian Trusted Computer Product Evaluation Criteria

CV Capabililty Vector

DAC Discretionary Access Control

DC Domain Controller

DLL Dynamic Link Library

DoD United States Department of Defense

DoS Denial of Service

DSD Dynamic Separation of Duty

FAT File Allocation Table

FDP User Data Protection Class in CC

FRU Ressource Utilization Class in CC

GC Global Catalog

GPO Group Policy Objects

GUI Graphical User Interface

IIS Internet Information Services

I/O Input/Output

IFS Installable File-System Drivers

INCITS InterNational Committee for Information Technology Standards

IOCTL Input/Output Control

IRP I/O Request Package

ITSEC Information Technology Security Evaluation Criteria

LAN Local Area Network

MAC Mandatory Access Control

MER Mutual Exclusive Role

NIST National Institute of Standards and Technology

NTFS New Technology File System

OS Operating System

OU Organizational Unit

PII Privacy Identifiable Information

PoLP Principle of Least Privilege

PP Protection Profile

RAID Redundant Array of Independent Disks

RBAC Role Based Access Control

RM Reference Monitor

SID Security Identifier

SMB Server Message Block

SoD Separation of Duty

SQL Structured Query Language

SSD Static Separation of Duty

ST Security Target

TCB Trusted Computing Base

TCSEC Trusted Computer System Evaluation Criteria

TOE Target of Evaluation

UNC Universal Naming Convention

WAN Wide Area Network

WebDAV Web-based Distributed Authoring and Versioning

WDK Windows Driver Kit

XML Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Vorwort

Als die ersten Rechenmaschinen entwickelt wurden, ging es primär darum, die Arbeit, welche zumeist im wissenschaftlichen Bereich angesiedelt war, zu unterstützen und zu automatisieren. Die riesigen Rechenmaschinen wurden über ein Terminal bedient, in welches man Lochkarten einschob und anschließend oft Stunden, wenn nicht sogar Tage lang, auf das Ergebnis wartete. Die Idee einer Zugangsbeschränkung belief sich damals – wenn überhaupt vorhanden – auf den physikalischen Zutritt zum Lochkartenterminal. Auch das Militär erkannte von Beginn an die Nützlichkeit der elektronischen Datenverarbeitung und förderte deren massiven Einsatz. Als die Rechenmaschinen leistungsfähiger, effizienter und kleiner wurden, fanden sie auch Einzug in die Wirtschaft und unser Alltagsleben. Durch die vermehrte Nutzung der elektronischen Datenverarbeitung sammelten sich in den Unternehmen bald immer mehr Betriebsgeheimnisse auf den Rechnern und auch bei Heimanwendern wurde der Computer mehr und mehr Ablageplatz für persönliche Informationen aller Art.

Das Militär handelt von Natur aus fast ausschließlich mit Daten, die aus taktischen Gründen streng vertraulich sind. Inspiriert durch die militärische Sichtweise wuchs auch in der Wirtschaft das Verständnis, dass Daten in den falschen Händen schnell einen enormen Nachteil zu Folge haben können, seien dies bekannt gewordene Betriebsgeheimnisse, Preisgestaltung oder Ähnliches. Daraus ergab sich ein Bedürfnis, das für jedermann bis heute Bestand hat – die Vertraulichkeit der Daten.

In Bereichen wo zum Schutz vor Schaden sichergestellt sein muss, dass die Daten korrekt sind, wie Banken, Krankenhäuser, etc., herrscht zusätzlich die Notwendigkeit zur Sicherstellung der Integrität. So wäre es beispielsweise fatal, wenn einem Patienten eine falsche Behandlung aufgrund einer inkorrekten Patientenakte widerfahren würde.

Im Bankenbereich geht es zwar nicht um Leben oder Tod, jedoch liegen die Schätze längst nicht mehr im Tresor sondern auf Datenträgern der Server abgespeichert und so kann eine kleine Manipulation am Konto einem Angreifer bedeutsamen Kapitalzuwachs bescheren.

Die Vernetzung der Computersysteme und der massive Durchbruch des Internets verschärften dann einerseits die Bedingungen an Vertraulichkeit und Integrität, andererseits warfen sie eine völlig neue Art von Schutznotwendigkeit auf. War früher die Gefahr, dass jemand in ein Unernehmen einbricht und die Computer dort für seine eigenen Interessen nutzt doch sehr gering, so ist durch die Automation und der fehlenden Notwendigkeit einer physikalisch-lokalen Präsenz, die Möglichkeit Rechenleistung an beliebig vielen Orten der Welt gleichzeitig abzuziehen doch sehr evident. Die Rede ist von unautorisierter Nutzung der Rechnerressourcen und macht sich in Sicherheitsproblemen wie Bot-Netze, Fast-Flux-Networks und unautorisierter Nutzung von Grid Ressourcen bemerkbar.

Aber auch Ressourcen müssen nicht immer durch den Angreifer direkt gewinnbringend missbraucht werden. Was bei den Daten durch Störung der Integrität als destruktive Attacke angesehen wird, ist auch bei physikalischen Ressourcen möglich. So können sie zum Nachteil eines Unternehmens derartig überlastet werden, dass dadurch ein finanzieller Schaden für das Unternehmen entsteht. Die Erpressung von Unternehmen mit derartigen DoS-Attacken war in der Vergangenheit kein Einzelfall.

Wie bereits erwähnt, war sich das Militär immer schon bewusst, dass der Vertraulichkeitsverlust von Daten einem taktischen Nachteil gleichzusetzen ist. Daher war der Schutz von Daten schon immer Aufgabenbestandteil des amerikanischen Verteidigungsministeriums, dem Deparment of Defense, kurz DoD. Aus deren Feder stammt das sogenannte „Orange Book“, auch bekannt unter dem „Namen Trusted Computer System Evaluation Criteria“, kurz TCSEC, welches 1983 erstellt wurde. In diesem Standardwerk finden sich wichtige Termini und Definitionen. Das bekannteste ist das Referenzmonitor Konzept, das in zahlreichen wissenschaftlichen Arbeiten für Zugriffskontrollsysteme immer wieder zitiert wird. Es handelt sich dabei um eine Entscheidungsinstanz deren Aufgabe es ist, den Zugriff eines Subjektes auf ein Objekt mit dem Wissen aus einer Entscheidungsdatenbank zu gewähren oder zu verweigern. Der Referenzmonitor muss dabei „tamperproof“ sein. Das bedeutet, er darf für keine Einflüsse von Außen empfänglich sein, muss also in jedem Fall seine Aufgabe korrekt erfüllen. Der Referenzmonitor ist Teil der Trusted Computing Base TCB, welche alle sicherheitsrelevanten bzw. sicherheitskritischen Bestandteile, sprich Hard-, Soft- und Firmware eines Computersystems umfasst. Sicherheitsprobleme innerhalb der TCB können aufgrund von Nebeneffekten zu weiteren Sicherheitsproblemen führen, die letztlich wiederum das gesamte System betreffen. Daher muss sichergestellt sein, dass die TCB bestmöglich abgesichert wird

und die Implementierung eine niedrige Fehlerrate aufweist. Weiters sollte sie, um eine geringe Angriffsoberfläche zu aufzuweisen, als auch die Verifikation des Codes zu ermöglichen, möglichst klein gehalten werden.

TCSEC unterteilt Computersysteme in 4 Schutzklassen.

- Schutzklasse D qualifiziert ein System mit minimalem Schutz
- Schutzklasse C Discretionary Protection
- Schutzklasse B Mandatory Protection
- Schutzklasse A Verified Protection

TCSEC wurde 1998 zusammen mit den „Information Technology Security Evaluation Criteria“, kurz ITSEC, welcher im europäischen Raum vorherrschend war und dem kanadischen „CTCPEC Standard“ zu den „Common Criteria“ zusammengefasst.

1.2 Common Criteria

Die Common Criteria definieren gemeinsame Kriterien für die Prüfung und die Bewertung der Sicherheit in der Informationselektronik und sind in 3 Teilen untergliedert.

- Teil 1 Introduction und General Model
- Teil 2 Security Functional Requirements
- Teil 3 Security Assurance Requirements

Es existieren 11 Funktionale Sicherheitsklassen, welche den Anspruch stellen, die Trusted Computing Base aus TCSEC vollständig abzudecken. Die für Zugriffskontrollsysteme am bedeutsamsten sind die Klasse FDP (User Data Protection Class) als auch die Klasse FRU (Ressource Utilization Class). Diese Klassen sind in Teil 2 – Functional Requirements – der Common Criteria beschrieben. Sie stellen die Basis für die Definition eines Protection Profiles (PP) und den Security Targets (ST) dar. Die funktionalen Anforderungen aus den jeweiligen Klassen beschreiben das gewünschte Verhalten eines TOE (Target of Evaluation). Während Protection Profiles eine allgemeine produktneutrale Spezifikation der Anforderungen darstellen, zielen Security Targets immer auf ein konkretes Produkt ab. Ein PP wird also immer auf eine TOE-Type ausgelegt, z.B. auf

die Gruppe der Zugriffskontrollsysteme, ein ST bezieht sich immer auf ein konkretes TOE, sprich einer konkreten Implementierung eines Zugriffskontrollsystems. Protection Profiles können also auch als Templates angesehen werden, die ein ST dann erfüllen kann oder nicht. Ein Security Target erfüllt ein Protection Profile, wenn es für ein bestimmtes TOE die funktionalen Anforderungen besser oder gleich gut wie das PP erfüllt und von gleichen oder schlechteren operationalen Umgebungsbedingungen ausgeht. Ein Protection Profile besteht im Wesentlichen aus

- *Security Objectives* Diese leiten sich aus der Security Problem Definition und den Functional Requirements her.

  Security Objectives for the TOE

  Security Objectives for the Operational Environment

- *Security Problem Definition* Die Erhebung der Security Problem Definition entzieht sich den CC, da jedes System individuell ist und daher ein Standardschema keinen Sinn machen würde. Die Unterstützung durch die CC erfolgt – wenn überhaupt – lediglich axiomatisch. Es ist aber essentiell, gerade in diesem Bereich erhebliche Zeit und Ressourcen zu investieren, da nur wenn man das Problem kennt, eine Verbeserung im Sinne der Security Objectives erfolgen kann.

1.3 Zugriffskontrollsysteme

Zugriffskontrollsysteme schützen Informationen und Ressourcen vor unerlaubter Verwendung. Damit spielen sie in der TCB eine entscheidende Rolle und stellen die Vertraulichkeit und Integrität der Daten sicher. Weiters verhindern sie den Diebstahl oder die unsachgemäße Nutzung von Rechenleistung. Wo die Grenzen der TCB gezogen werden, ist System und Policy abhängig wie auch letzten Endes auch eine Definitionsfrage. Das Zugriffskontrollsystem eines Webservers ist beispielsweise nicht Bestandteil der OS-TCB, sehr wohl aber der Webapplikation TCB, welche ein Gesamtsystem aus OS und Webserver bildet. In den CC werden die Grenzen der TCB durch die TOE und damit durch die PP und ST beschrieben.

Ein Computersystem ist – ähnlich einer Kette – nur so stark wie das schwächste Glied. Zugriffskontrollsysteme sind also neben Maßnahmen wie Kryptographie, Authentifizierung und Autorisierung, Sicherheitsmanagement, usw. nur eine von vielen, die optimal umgesetzt werden müssen, um ein starkes Gesamtkonzept bieten zu können.

1.3.1 Discretionary Access Control Systems

Bei den DAC handelt es sich um ein Zugriffskontrollsystem, das auf benutzerbestimmten Rechten basiert. Das bedeutet, dass jeder, der Besitzer eines Systemobjektes ist, anderen Individuen den Zugriff darauf erlauben oder verweigern kann, ohne dass dabei die Interaktion eines Administrators von Nöten ist. In Zugriffskontrollsystemen werden die Entitäten durch Subjekte und Objekte repräsentiert. Subjekte sind im Regelfall Menschen, können aber auch digitale Agenten und deren Prozesse sein. Die Subjekte werden oft auch als Prinicpals, sprich Teilnehmer bzw. User, bezeichnet. Objekte stellen wiederum den Überbegriff für Hardware-, Softwareressourcen und Datenobjekte dar. Subjekte und Objekte interagieren über Operationen, ein Subjekt greift also operativ auf ein Objekt zu. Welche Auswirkungen dieser Zugriff auf das Objekt hat, ist von der Art der Operation abhängig. In konventionellen Betriebssystemen können die Operationen beispielsweise „read“, „write“ und „execute“ umfassen. In Datenbanksystemen hingegen meist „insert“, „delete“ und „update“. All diese Operationen haben in der Regel unterschiedliche Auswirkungen auf die Objekte zufolge. Wenn auch für die Rechtevergabe die Interaktion eines Administrators nicht notwendig ist, so macht es doch Sinn, dass jemand als Verwaltungsinstanz über dem DAC steht. Dies erfolgt durch einen Administrator, der nicht zwangsläufig außerhalb des DAC als Überwachungsinstanz stehen muss sondern auch selbst ein Subjekt innerhalb DAC-Systems sein kann, jedoch mit speziellen Privilegien ausgestattet ist, die unveränderlich sind. Der Besitzer einer Ressource, bzw. Objektes, ist meist der Erzeuger des selbigen. Ein User besitzt also automatisch die Objekte, die er auch selbst als Folge seiner Arbeit im System erzeugt hat. Er kann jedoch auch Objekte veranlasst durch das System oder den Administrator in seinen Besitz zugewiesen bekommen, ohne dass er der tatsächliche Erzeuger des Objektes ist. Wie er auch immer in den Besitz eines Objektes gelangt ist, spielt letztlich keine Rolle. Er kann für diese Objekte Zugriffsrechte vergeben also bestimmte Operationen darauf erlauben oder verweigern. In manchen Systemen betreffen diese Einschränkungen sogar den Administrator, wenn er selbst als Principal im DAC auftritt. Um diesem nicht vollständig sein Mitspracherecht an den Systemobjekten abspenstig zu machen, wird ihm meist eine System-Operation gewährt, mit der er einem User in letzter Konsequenz den Besitz eines Objektes entziehen und neu vergeben kann. Damit unterliegt der Administrator derselben Rechtsgewalt wie auch die konventionellen User, behält aber im Notfall immer das letzte Wort.

Das DAC-System wird im mathematischen Sinne immer als Anordnung von $S\ X\ O$ angesehen, wobei S die Menge aller Subjekte und O die Menge aller Objekte darstellt. Diese Access Control Matrix stellt selbst ein Objekt dar und kann somit von den Subjekten operativ verändert werden. Möchte ein User auf eine Ressource operativ zugreifen, so erfolgt zunächst eine Identitätsbestimmung des Prinicpals, sprich Teilnehmers bzw. Users. Ist das zugehörige Subjekt in der Zeile gefunden, so kann in der Spalte nach

dem betreffenden Objekt, welches die Ressource repräsentiert gesucht und die Matrix nach den Rechten ausgewertet werden, die sich in der entsprechenden Zeilen/Spalten Kombination befinden. Die Menge dieser Rechte entscheidet letztlich darüber, ob die betreffende Operation vom System gewährt wird oder nicht.

Obwohl die Anordnung über eine Matrix einen pädagogischen Anschauungswert besitzt, ist sie in der Praxis nicht praktikabel. In realen Systemen existieren zu viele Subjekte und Objekte, die zusammen mit den Rechten einen Berechtigungsraum schaffen, der zum einen unüberschaubar und damit fehleranfällig wird, zum anderen die Performance negativ beeinflusst, da immer die komplette Matrix in den Hauptspeicher geladen werden muss. Um diesem Problem entgegenzuwirken, muss man sich entscheiden, ob die Rechte beim Subjekt gespeichert werden sollen (Capability-Vector) oder beim Objekt besser aufgehoben sind (Access Control List). Beide Varianten haben ihre Vor- und Nachteile, wenn es darum geht, die systemweiten Rechte zu bestimmen. Bei den Capabilities ist zwar sehr schnell erfasst, welche Rechte der User besitzt, jedoch nicht welche User Rechte auf einer bestimmten Ressource besitzen. Gegengleich verhält es sich bei den Access Control Lists.

1.3.2 Mandatory Access Control Systems

MAC verlässt sich nicht alleine auf die Rechtevergabe durch die Subjekte sondern installiert zusätzlich systemweite Bedingungen, die den Informationsfluss kontrollieren. MAC gilt daher als Zugriffskontrollsystem, das auf systembestimmten Rechten basiert. Es findet neben dem Gesundheits- und Behördenwesen hauptsächlich im militärischen Bereich Anwendung, wo Multi-Level-Sicherheitsanforderungen aufgrund der dort herrschenden Hierarchie und den unterschiedlichen Vertraulichkeits- und Geheimhaltungsstufen von Informationen bestehen. In DAC ist es einem Subjekt möglich, Rechte an ein anderes Subjekt weiterzureichen, ohne dass dies durch systembezogene Kontrolle eingeschränkt wird. Genau diese Verhaltensweise spricht MAC zuwider, wenn Information zwischen zwei unterschiedlichen Ebenen der Vertraulichkeit fließt. Es muss also sichergestellt sein, dass der Informationsfluss zwischen Level a und Level b kontrolliert durch das System abläuft, also nicht durch den Benutzer alleine durch Vergabe von Zugriffsrechten bestimmt werden kann.

Prominente Vertreter dieser Systeme sind das Bell La Padula Modell und das Verbandsmodell nach Dorothy Denning, welche nachfolgend erläutert werden.

1.3.2.1 Verbandsmodell nach Dorothy Denning

Das Verbandsmodell nach Dorothy Denning bildet das Grundprinzip für alle MAC Systeme, weil es den mathematischen Hintergrund definiert, auf dem komplexere Systeme aufbauen.

Verbandsgesetze

Ein Verband $(M, \sqcap, \sqcup)$ ist ein Trippel aus einer nicht leeren Menge M und zwei inneren Verknüpfungen $\sqcap$ und $\sqcup$ die folgende Bedingungen für alle $u, v, w \in M$ erfüllen:

Kommutiativität
$u \sqcap v = v \sqcap u$
$u \sqcup v = v \sqcup u$

Assoziativität
$u \sqcap (v \sqcap w) = (u \sqcap v) \sqcap w$
$u \sqcup (v \sqcup w) = (u \sqcup v) \sqcup w$

Idempotenz
$u \sqcap u = u$
$u \sqcup u = u$

Absorbtion
$u \sqcap (u \sqcup v) = u$
$u \sqcup (u \sqcap v) = u$

Es kann dann eine Verbandsordnung auf M definiert werden. Eine geordnete Menge $(M, \leq)$ heißt verbandsgeordnet, wenn alle Paare $(a, b) \in M$ ein Infimum und ein Supremum besitzen. Das bedeutet $inf(a, b)$ ist die größte untere Schranke. Unter allen Elementen c aus M für die gilt $c \leq a$ und $c \leq b$ ist es das größte. Umgekehrt stellt unter allen Elementen c aus M für die gilt $a \leq c$ und $b \leq c$ das kleinste Element $sup(a, b)$, also die kleinste obere Schranke dar.

Ist $M \leq$ verbandsgeordnet, so ist $(M, \leq, inf, sup)$ ein Verband.

Für das Verbandsmodell von Dorothy Denning müssen folgende Prämissen gelten:

Das Subjekt muss alle Rechte besitzen, die es für die Erledigung seiner Aufgaben benötigt, jedoch nicht mehr. Dies gilt als das „Need to Know“-Prinzip.

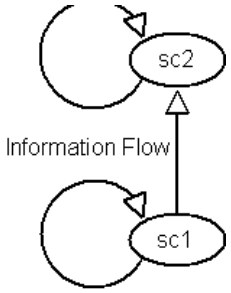


Abbildung 1.1: Informationsfluss Hasse Diagramm

Jedem Objekt wird eine Sicherheitsklasse $sc(s) \in SC$ zugewiesen. Die Menge SC ist Anwendungs- und Organisationsabhängig. Mögliche Ausprägungen für SC sind beispielsweise *Public, Sensitive, Private, Confidential* oder *Anonymous Data, PII (Privacy Identifiable Information), Sensitive PII.*

Verbandsordung
Auf SC wird eine Ordnung, ein Verband, definiert. Diese Ordnung verhindert, dass Information zwischen zwei Sicherheitsklassen unkontrolliert fließen kann. So ist ein Informationsfluss nur zu gleicher oder höherer Sicherheitsklasse möglich. Seien $sc1$ und $sc2$ zwei Elemente aus SC mit $sc1 \leq sc2$, so kann von Objekten mit $sc(o) = sc1$ Information zu Objekten fließen die ebenfalls $sc(o) = sc1$ oder $sc(o) = sc2$ besitzen. Ein Informationsfluss von Objekten mit $sc2$ ist nur zu Objekten die ebenfalls $sc2$ besitzen möglich, nicht jedoch zu Objekten mit $sc1$. Der erlaubte Informationsfluss kann durch ein Hasse Diagramm sehr anschaulich dargestellt werden.

Das tuple $(M \leq, \oplus, \otimes)$ bildet den Verband. $\oplus$ und $\otimes$ sind die Operatoren, die bei definierter partieller Ordnung das Supremum bzw. Infimum liefern. Auf Objekte, die mit Infimum von M markiert sind, kann von allen Subjekten lesend oder schreibend zugegriffen werden. Objekte die mit Supremum von M markiert sind, entsprechen der höchsten Sicherheitsstufe.

Kombinationsregel
Werden 2 Objekte zu einem neuen kombiniert, so lautet die Regel $sc(o3) = sc(o1) \oplus sc(o2)$. Da der Operator $\oplus$ kommutativ und assoziativ ist kann eine Berechung für eine beliebige Menge an Objekten einfach erfolgen $sc(oneu) = sc(o1) \oplus sc(o2) \oplus sc(o3) \oplus \ldots$

Auch wenn das Verbandmodell nur von einer Objektklassifizierung spricht, so gilt diese auch für die Subjekte, weil sie streng genommen auch System-Objekte sind. Informationen können im Wesentlichen in drei verschiedenen Ausprägungen fließen. Objekt-Subjekt, Subjekt-Objekt und Objekt-Objekt. Der Objekt-Subjekt Informationsfluss ist immer ein lesender. Subjekt-Objekt bzw. Objekt-Objekt sind immer schreibende Informationsflüsse. Sie alle werden im Wesentlichen durch die Verbandsordnung kontrolliert.

Ein Subjekt mit einer gewissen Klassifizierung kann also kein Objekt einer höheren Klassifizierung lesen, jedoch auf dieses schreiben. Umgekehrt kann es auf Objekte niedriger Klassifizierung lesen, jedoch nicht schreiben. Zu Beginn müssen alle Objekte als auch die Subjekte mit einer Klassifizierung versehen werden. Ist dies der Fall, so erfolgt die Klassifizierung neuer Objekte über die Kombinationsregel.

1.3.2.2 Bell La Padula Modell

Das Verbandsmodell beschreibt, dass der Informationsfluss in gewisser Richtung erlaubt und in anderer verweigert werden soll, liefert aber keine formale Anleitung, wie dieser gemäß der Verbandsordnung zu kontrollieren ist. Das Verbandsmodell bildet also die Grundlage für andere Modelle, welche die Informationsflusskontrolle formal spezifizieren, wie das Bell La Padula Modell. Ausgangsbasis für das Bell La Padula Modell ist ein DAC mit einer Access Control Matrix, in welcher Subjekte S, Objekte O und die darauf gültigen Rechte aus der Menge R organisiert werden, für die gilt:

$M[s, o] = \{r \mid r \in R,\ r\ auf\ o\ durch\ s\ akzeptabel\}$

Zusätzlich definiert man Sicherheitsmarken, auch Labels genannt und Sicherheitskategorien, auch Compartments genannt. Die Labels erhalten eine partielle Ordnung. Im einfachsten eine asymmetrische Relation, welche die Labels ordnet. Beispielsweise $\{public < confidential < secret < topsecret\}$. Die Compartments stellen die möglichen Rollen dar, die ein User annehmen kann. Sie sind daher von der Umgebung abhängig, in der das Zugriffskontrollsystem eingesetzt wird. So können sie im Gesundheitswesen beispielsweise $\langle Arzt, Pflegepersonal, Oberschwester, Verwaltung, Patient\rangle$ im militärischen Bereich $\langle Offizier, Unteroffizier, Rekrut, ZivileMitarbeiter\rangle$ usw. sein. Die Compartments stellen lediglich eine Menge ohne Ordnung dar, haben also keine spezielle Reihenfolge. Aus diesen Elementen konstruiert man nun Sicherheitsklassen $SC = \{k \mid k = (a, B),\ a \in SM,\ B \subseteq C\}$ Man wählt also für eine Sicherheitsklasse k ein Element aus den Labels und eine Teilmenge aus den Compartments, bzw. ein Element der Potenzmenge der Compartments. Auf der so entstandenen Menge an Sicherheitsklassen definiert man nun wieder eine partielle Ordnung, die in diesem Fall antisymmetrisch ist. $\forall u, v \in SC,\ u = (a, B),\ v = (\acute{a}, \acute{B}) :\ u \leq v \iff a \leq \acute{a} \wedge B \subseteq \acute{B}$ *oder auch v dominiert u*

Jedes Subjekt bekommt dann ein Element aus den Sicherheitsklassen zugewiesen: $sc(s) \in SC$. $sc(s)$ wird als „Clearance“ des Subjekts bezeichnet. Jedes Objekt bekommt ebenfalls ein Element aus den Sicherheitsklassen zugewiesen: $sc(o) \in SC$. $sc(o)$ wird als „Classification“ des Objekts bezeichnet. Wenn alle Subjekte und Objekte eine

entsprechende Clearance bzw. Classification besitzen, so wird der Informationsfluss mit folgenden Regeln kontrolliert.

Simple Security Rule (no read up)
Diese Regel ist ziemlich intuitiv. Informationen die nicht für die Augen von Subjekten bestimmt sind, deren Vertrauenswürdigkeit nicht ausreicht, um sie zu lesen, müssen vor diesen geheim gehalten werden. Das Hauptinteresse liegt bei MAC in der Vertraulichkeit der Daten. So kann also ein Subjekt auf ein Objekt nicht lesend zugreifen, wenn die Classification des Objektes höher als die Clearance des Subjektes ist.

$$s \in S \; kann \; auf \; o \in O \; nur \; dann \; lesen \; wenn$$
$$r \in M[s,o], \; wobei \; r = read \; \wedge \; sc(s) \geq sc(o)$$

Damit kann ein Subjekt immer noch anderen Subjekten Rechte auf den eigenen Objekten einräumen, wie dies auch im Discretionary Access Control der Fall ist. Die rechte Seite der Konjunktion erzwingt aber, dass das Subjekt welchem das Lesen erlaubt werden soll auch vom System dazu autorisiert wird. Dies ist nur dann der Fall wenn das Subjekt das Objekt dominiert, also die Clearance gleich oder höher als die Classification ist.

**-Rule (no write down)*
Diese Regel ist auf den ersten Blick nicht so intuitiv wie die Simple Security Rule, hat jedoch einen tieferen Sinn. Sie besteht aus zwei Teilen und es geht bei dieser Regel im Wesentlichen darum, dass die Simple Security Rule nicht unterlaufen werden kann. Dies kann absichtlich oder unabsichtlich geschehen. Ein Subjekt a mit bestimmter Clearance könnte ohne diese Regel einem anderen Subjekt b mit niedriger Clearance Informationen zugängig machen, die eigentlich nicht für die Augen von b besitmmt sind, indem a Informationen aus einem Objekt, das sich dem Zugriff von b entzieht in ein neues Objekt kopiert, auf dem b Zugriff besitzt. Dieses Problem wird als das „Confinement Problem“ bezeichnet. Um dies zu verhindern verbietet die *-Rule, dass ein Subjekt auf ein Objekt schreiben kann, wenn die Clearance des Subjektes höher als die Classification des Objektes ist.

$$s \in S \; kann \; auf \; o \in O \; nur \; dann \; schreiben \; wenn$$
$$r \in M[s,o], \; wobei \; r = write \; \wedge \; sc(s) \leq sc(o)$$

$$s \in S \; kann \; auf \; o \in O \; nur \; dann \; lesen \; und \; schreiben \; wenn$$
$$r \in M[s,o], \; wobei \; r = read/write \; \wedge \; sc(s) = sc(o)$$

Der erste Teil der *-Regel stellt sicher, dass Subjekte keine Informationen an bereits vorhandene Objekte anhängen können wenn ihre Clearance größer als die Classification

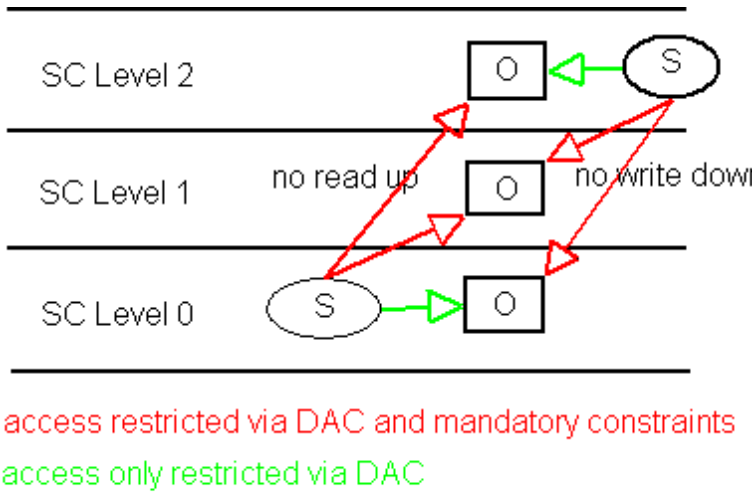


Abbildung 1.2: Informationsfluss zwischen den Sicherheitsklassen

ist. Mit „anhängen“ ist ein blindes Schreiben auf dem Objekt gemeint und schließt damit das gezielte Löschen von Informationen durch Überschreiben aus. Der zweite Teil stellt sicher, dass lesen und schreiben und damit auch das Löschen nur dann möglich sind, wenn die Clearance gleich der Classification ist.

Mit den beiden Regeln ist garantiert, dass der Informationsfluss nur von unten nach oben oder innerhalb einer Sicherheitsklasse statt findet.

1.3.3 Role Based Access Control Systems

In DAC wie auch in MAC ist der Benutzer selbst Besitzer der Objekte ist und kann darauf Rechte vergeben. Während dies bei MAC durch systembezogene Strategien limitiert wird, können bei DAC die Rechte ohne Einschränkung durch den Benutzer vergeben werden. Tatsache ist, dass in konventionellen Unternehmen meist nicht der Benutzer sondern die Organisation selbst Besitzer der Objekte ist. Der Benutzer selbst besitzt lediglich Rechte auf den Objekten, die er benötigt, um seiner Aufgabe im Unternehmen gerecht zu werden. Er kann also keine Rechte benutzerbestimmt für Objekte vergeben, selbst nicht auf diesen, die er im Zuge seiner Arbeit selbst erzeugt hat. Ein zentraler Unterschied von RBAC zu DAC und MAC ist also, dass die Systemobjekte nicht im Besitz der Anwender sondern der Organisation sind.

Hierfür werden Rollen definiert, die eine Funktion im Unternehmen beschreiben. Einer Rolle werden gemäß dem „Principle of Least Privilege“ genau so viele Rechte zugeord-

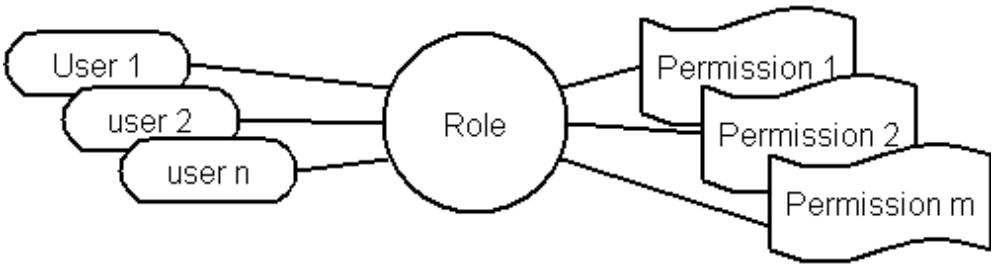


Abbildung 1.3: Indirektionskonzept „Rolle“

net, wie benötigt werden, um die beschriebene Funktion im Unternehmen zu erfüllen. Einer definierten Rolle können mehrere Mitglieder, sprich Benutzer zugewiesen werden. Für die Benutzer hat dies zur Folge, dass sie alle in der Rolle enthaltenen Berechtigungen nutzen können. Die Rechte werden also nicht direkt dem Benutzer – vergleichsweise den Capabilities – oder dem Objekt (Access Control List) sondern über das Indirektionskonzept „Rolle“ zugeordnet.

Diese Indirektion hat ein Entkoppeln der Benutzer und Rechte zur Folge. Es können einer Rolle ohne Rücksicht auf die zugeordneten Benutzer Rechte zugewiesen oder entzogen werden. Analog gilt dies für die Benutzer, die einer Rolle zugeordnet oder entfernt werden können, ohne dass dabei die Berechtigungsstruktur angetastet werden muss. Dies hat einen enormen administrativen Vorteil und verringert die Fehlerwahrscheinlichkeit bei der Umsetzung der Sicherheitspolitik. Vor allem in Anwendungsbereichen, wo eine komplexe Organisationsstruktur herrscht, die von einer unternehmensweiten Sicherheitspolitik begleitet wird, ist RBAC enorm im Vorteil.

[6] definiert RBAC als eine Form von MAC ohne Multilevel-Sicherheitsanforderungen. Das bedeutet es existieren keine Vertraulichkeitsstufen. Daher ist RBAC für Unternehmen interessant, die nicht nach einem streng hierarchischen Vertraulichkeitssystem aufgebaut sind, also nicht die Geheimhaltung der Daten sondern die Integrität an oberster Stelle steht. Wie bereits beschrieben erfolgt in MAC der Prozess der Clearance von Subjekten – $sc(s) \in SC$ und der Classification von Objekten $sc(o) \in SC$ Diese beiden Schritte sind der Mitgliedschaftszuweisung der User zu den Rollen und der Rechte-Ausstattung der Rollen sehr ähnlich, Abbildung 1.4

Ein weiterer Unterschied zwischen MAC und RBAC ist, dass MAC definiert, welche Art Informationsfluss nicht statt finden darf. RBAC hingegen definiert, welcher Informationsfluss statt finden kann und erlaubt mit den Rollen einen multidirektionalen Informationsfluss. [10]

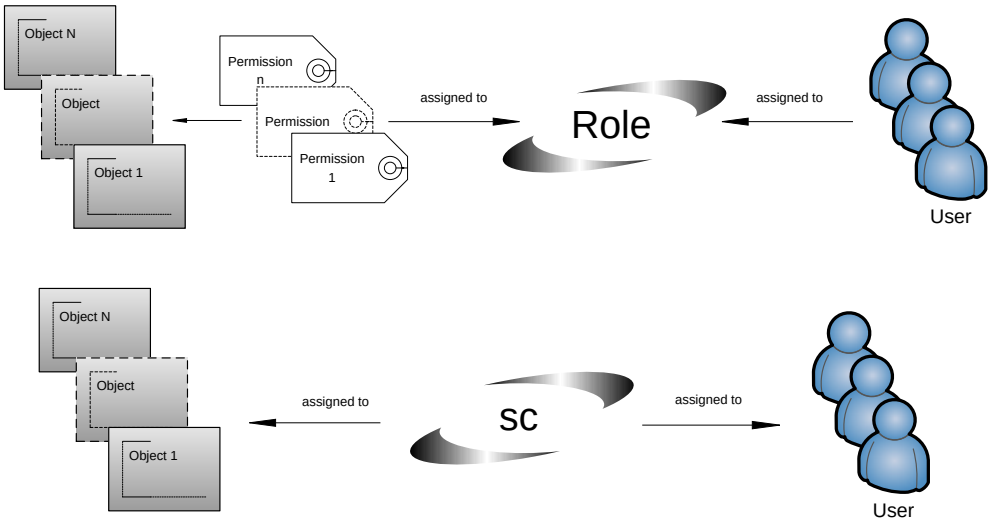


Abbildung 1.4: Rollen-SC Vergleich

Kapitel 2

Geschichte von RBAC

2.1 Von den Anfängen zum ANSI INCITS Standard

1992 formulierten David Ferraiolo und Richard Kuhn am National Institut of Standards and Technology (NIST) das Paper „Role Based Access Control“ [6], das durch seine Effizienz als das prädominierende Modell für fortgeschrittene und unternehmensbezogene Zugriffskontrollsysteme angesehen wird. Eine große Herstelleranzahl von IT Produkten, darunter IBM, Sybase, Secure Computing und Siemens, haben sich an diesem Modell, welches auch unter dem Namen „Ferraiolo-Kuhn-Model“ bekannt wurde, orientiert. 2000 wurde das Ferraiolo-Kuhn-Model mit dem Framework von Shandhu et al. [13], das 1996 veröffentlicht wurde, verbunden, um ein einheitliches RBAC-Modell zu erstellen. Dieses Modell wurde unter dem Titel „The NIST Model for Role-Based Access Control“ [12] veröffentlicht. 2004 wurde es in den ANSI/INCITS Standard [8] überführt, der bis heute die Basis für eine einheitliche Entwicklung und der funktionalen Spezifikation von RBAC Systemen darstellt. [11]

2.2 Ferraiolo-Kuhn-Model (NIST 1992)

Das Paper beinhaltet nach einer Einleitungsphase, in der die Motivation und die Einsatzgebiete von RBAC Systemen erläutert werden, eine formale Definition der RBAC-Elemente.

- Für jedes Subjekt ist die aktive Rolle diejenige, die das Subjekt gegenwärtig nutzt. $AR(s : subject) = \{active\ role\ for\ subject\ s\}$.

- Jedes Subjekt kann autorisiert werden, eine oder mehrere Rollen auszuüben. $RA(s : subject) = \{authorized\ roles\ for\ subject\ s\}$.

- Jeder Rolle kann erlaubt werden, mehrere Transaktionen auszuführen. $TA(r : role) = \{transactions\ authorized\ for\ role\ r\}$.

Eine Transaktion repräsentiert im Ferraiolo-Kuhn-Model eine Menge an Operationen, die an Daten-Objekte gebunden sind – also eine Transformationsprozedur mit einer zugeordneten Menge an Daten, auf welche bei der Ausführung zugegriffen wird.

Subjekte können Transaktionen ausführen; Die Funktion *exec(s,t)* ergibt *true*, wenn ein Subject s eine Transaktion t ausführen kann, andernfalls *false*.

$exec(s : subject,\ t : tran) = true\ if\ subject\ s\ can\ execute\ transaction\ t.$

Auf Basis der eben definierten Funktionen werden drei Regeln definiert:

- *Regel 1 - Role Assignment* Ein Subjekt kann eine Transaktion nur dann ausführen, wenn das Subjekt eine Rolle ausgewählt hat oder automatisch aktiviert wurde. $\forall s : subject,\ t : tran(exec(s,t) \Rightarrow AR(s) \neq \oslash)$. Der Identifikations- und Authentifikations-Prozess z.B der Benutzer-Login wird dabei nicht als Transaktion angesehen. Alle anderen Aktivitäten am System sind hingegen transaktionsgesteuert. Alle aktiven User müssen eine Rolle aktiviert haben, um überhaupt handlungsfähig zu sein.

- *Regel 2 - Role Authorization* Eine aktive Rolle eines Subjekts muss für dieses autorisiert werden. $\forall s : subject,\ (AR(s) \subseteq) RA(s))$. Zusammen mit Regel 1 stellt Regel 2 sicher, dass ein Subjekt nur Rollen annehmen und die damit verbundenen Privilegien nutzen kann, für die es auch autorisiert ist.

- *Regel 3 - Transaction Authorization* Ein Subjekt kann eine Transaktion nur dann ausführen, wenn die Transaktion für die aktive Rolle autorisiert ist. $\forall s : subject,\ t : tran,\ (exec(s,t) \Rightarrow t \in TA(AR(s)))$. Zusammen mit Regel 1 und Regel 2 stellt Regel 3 Sicher, dass das Subjekt lediglich Transaktionen ausführen kann, für die es auch tatsächlich autorisiert ist. Es ist wesentlich hervorzuheben, dass Regel 3 eine notwendige Bedingung darstellt, also die Rollenautorisierung Grundvoraussetzung ist, dass überhaupt eine Transaktion ausgeführt werden kann, dies aber durch weitere Einschränkungen aber wieder unterbunden werden kann. Die Regel garantiert also keine Ausführbarkeit einer Transaktion nur weil sie in der Menge TA(AR(s)) der potentiell ausführbaren Transaktionen aufscheint.

Die Zugriffskontrolle unter Verwendung der obigen Regeln und Funktionen beschreibt bzw. erfordert nicht, dass eine Prüfung der Zugriffsrechte auf den dazugehörigen Datenobjekten für den User statt finden muss oder die Zugriffe der Transformationsprozedur auf den Datenobjekten überwacht werden müssen. Viel mehr erfolgt mit den Regeln die Prüfung auf höherer Abstraktionsebene, ob ein User über eine Rolle autorisiert ist, eine bestimmte Transaktion auszuführen oder nicht.

Sicherheitsfragen, die den Zugriff auf die Datenobjekte betreffen müssen also schon zu Designzeit beim Transaktionsentwurf getroffen werden, wo Datenobjekte und darauf ausführbare Operationen zu einer untrennbaren Einheit zusammen gefasst werden.

Ferraiolo-Kuhn stellt auch frei, über das Konzept „Transaktion“ lediglich das Transformations-Prozedere zu definieren, die zugehörigen Daten-Objekte jedoch außen vor zu lassen. Diese Variante erfordert allerdings eine vierte Regel, in welcher die Kontrolle über die Zugriffsmodalitäten statt findet, mit denen ein bestimmter User Daten-Objekte über Transaktions-Programme erreichen kann. Eine solche Regel könnte folgendermaßen aussehen. $\forall s : subject,\ t : tran,\ o : object,\ (exec(s,t) \Rightarrow access(AR(s),t,o,x))$. Eine Transaktion-zu-Objekt-Zugriffsfunktion *access(r,t,o,x)* die angibt, ob es für ein Subjekt in Rolle r erlaubt ist, dass Objekt o im Modus x über Transaktion t zu erreichen. x kann eine Reihe von Modi annehmen wie *read*, *write*, *execute*. Diese Regel kann auch dazu verwendet werden, die Zugriffs-Invarianz der Transaktionen auf den zugeordneten Objekten zu unterbinden. Das bedeutet, wenn ein Transaktions-Programm einmal mit den zugehörigen Objekten verknüpft und einer Rolle zugewiesen wurde, so ist diese Transaktion für alle User ausführbar, die Mitglied der Rolle sind. Überwacht wird lediglich die Ausführung der Transaktionen, nicht der Zugriff auf die Objekte. In manchen Fällen kann es aber angebracht sein, dass die Transaktion für bestimmte User auf gewissen Objekten nicht erlaubt sein soll, indem die zusätzliche Funktion *access* genau dies unterbindet, und damit den Zugriff auf die Objekte unterschiedlich behandelt. *exec* stellt dabei eine hinreichende Bedingung für *access* dar, *access* hingegen eine notwendige Bedingung für *exec*. Das bedeutet, wenn *exec* wahr liefert, also die Ausführung erlaubt ist, so muss auch *access* wahr liefern. Ist die Ausführung nicht erlaubt, sprich *exec* liefert falsch so kann *access* erlaubt sein oder auch nicht.

Im Allgemeinen kann es sehr schwierig sein, herauszufinden, ob und in welcher Art und Weise Daten modifiziert wurden und ob dies in einer autorisierten Art und Weise geschehen ist. Das Modell schlägt daher eine Zertifizierung der Transaktionen vor, um die Ausführung von unautorisierten Operationen zu verhindern und die Integrität der Daten sicherzustellen. Nicht nur die Ausführung der Operationen, die in einem Transaktionsprogramm enthalten sind, können mit Regel 4 überwacht werden. So besteht beispielsweise die Möglichkeit die System-Zugriffe auf die Transaktions-Programme mit

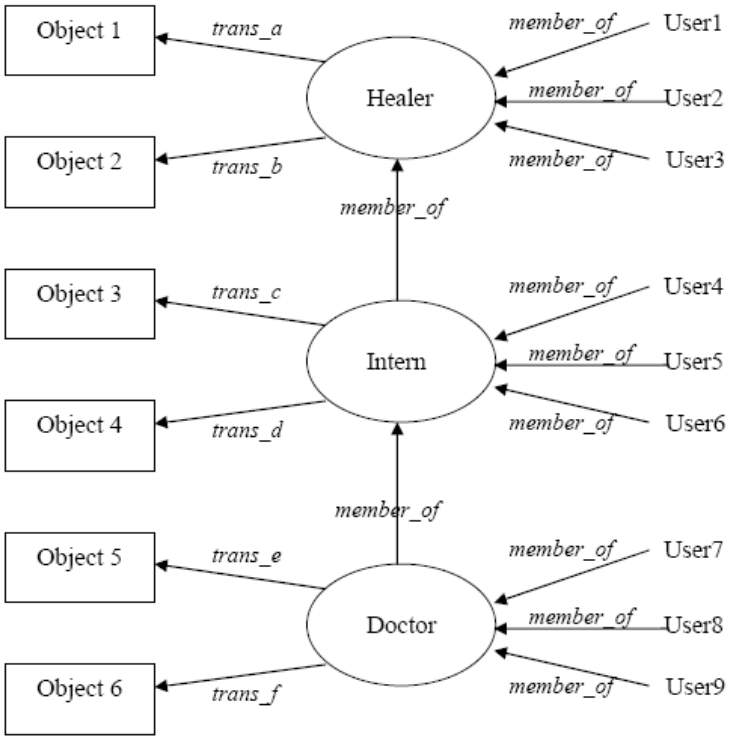


Abbildung 2.1: Vererbung Ferraiolo-Kuhn-Model, Quelle: [6]

dieser Regel zu überwachen und die Zertifizierung zu überprüfen, indem das System selbst als Subjekt auftritt.

2.2.1 Vererbungsstrategien

Ferraiolo-Kuhn führt auch bereits ein einfaches Konzept der Vererbung über eine „member_of"- Relation ein, in der die Senior-Rolle sämtliche Transaktionen von der Junior-Rolle erbt, bzw. diese durch die zugeordneten User ausführbar sind.

2.2.2 Constraints

Auch das „Principle of Least Privilege" sowie das Konzept der „Separation of Duty" werden im Ferraiolo-Kuhn Modell angesprochen. Letzteres beschreibt, dass für eine bestimmte Menge an Transaktionen, meist ein Geschäftsprozess, ein Individuum nicht in der Lage sein sollte, diese im Alleingang auszuführen. Dabei hängen die Mächtigkeit der Mengen, der Inhalt und die Art des wechselseitigen Ausschlusses innerhalb maßgeblich von der Natur der Applikation ab, in welcher das System zum Einsatz kommt. Im einfachsten Fall beinhaltet die Menge zwei Transaktionen, die im wechselseitigen Aus-

schluss stehen. Auch schlägt das Modell bereits vor, dass dies statisch oder dynamisch erfolgen kann und weist darauf hin, dass die dynamische Umsetzung der Separation of Duty die schwierigere der Beiden Varianten ist, jedoch manchmal aus Wirtschaftlichkeitsgründen vor zu ziehen ist. Vor allem dann, wenn durch statische Einschränkungen die Kosten größer sind als der Nutzen, da ansonsten für jede Transaktion, die separiert werden soll, im statischen Fall auch eine reale Person vorhanden sein müsste, die diese Rolle allzeit verkörpert, da ansonsten eine Zuweisungs-Änderung im System durch den Administrator notwendig wäre. Die Static Separation of Duty kann durch Überwachung der User-Rollen-Zuweisung oder der Transaktions-Rollen-Zuweisung erfolgen, setzt also bei der Role-Authorization *RA(subject:s)* an.

Dynamic Separation of Duty hingegen schränkt bei der Role-Authorization nicht ein, beschränkt jedoch in weiterer Folge die Role-Activation *AR(subject:s)*. Für Static Separation of Duty müssen also bei der Transaktions-Ausführung lediglich die autorisierten Rollen, sprich die Rollenmitgliedschaft des Users überprüft werden – die Funktion *RA(s:subject)* liefert die notwendigen Informationen. Für die dynamische Einschränkung muss das System die Rollenmitgliedschaft als auch die User-ID für die Ausführung einer Transaktion prüfen.

Als Beispiel führt Ferraiolo-Kuhn ein Szenario auf, in dem ein Zahlungsvorgang durchgeführt wird. In diesen Vorgang sind zwei Rollen involviert, der Zahlungs-Initiator und der jenige der den Zahlungslauf autorisiert, also freigibt. Um vor Missbrauch zu schützen, soll kein Individuum in der Lage sein, diesen Vorgang alleine ausführen zu können. Im statischen Fall kann man diese Anforderung leicht lösen, indem bei der Rollenzuweisung darauf geachtet wird, dass die beiden Rollen nur unter wechselseitigem Ausschluss zugewiesen werden. Dies ist aber oft nicht praktikabel, da ansonsten für jede Rolle mindestens ein Mitarbeiter vorhanden sein und damit keine Engpässe entstehen, auch eine entsprechende Mitarbeiter-Redundanz eingeplant werden müsste, falls einer der Mitarbeiter ausfällt. Die dynamische Variante hingegen erlaubt Autorisierung beider Rollen für einen Mitarbeiter, unterbindet aber, dass der komplette Vorgang durch den selbigen durchgeführt werden kann, indem die User-ID für den Vorgang überwacht wird. Wichtig hierbei ist, dass Ferraiolo-Kuhn kein Session Konzept erwähnt. Die Funktion *AR(s:subject)* liefert immer nur ein Element aus der Menge *RA(s:subject)*. Die Dynamic Separation of Duty, wie sie im ANSI [8] Standard beschrieben ist, baut auf dem Session-Begriff auf. In hiesigem Fall ist es lediglich ein erster Ansatz, der über die User-ID verhindert, dass zwei nicht verträgliche Rollen sequentiell ausgeführt werden können, da immer lediglich eine Rolle aus der Menge der autorisierten Rollen als aktiv gilt und somit die Session lediglich eine Rolle enthält.

In der Conclusion weist das Paper noch darauf hin, dass eine breite Palette an RBAC-Systemen über die Jahre hinweg ohne jeglichen Standard entwickelt wurden und ge-

genwärtig ihren Einsatz in kommerziellen Systemen finden, diese jedoch nur schwer evaluiert, miteinander verglichen und getestet werden können, da kein formaler Standard über RBAC existiert. TCSEC behandelt lediglich DAC- und MAC- Systeme und so empfiehlt sich Ferraiolo-Kuhn quasi selbst als Basis für die Entwicklung eines einheitlichen Standards.

2.3 Sandhu et al. Model (NIST 1996)

Vier Jahre später wurde das Paper Role Based Access Control Models von Ravi S. Sandhu et al. veröffentlicht. Es stellt eine Erweiterung als auch eine Wiederholung und teilweise Neugestaltung der Konzepte aus dem Jahre 1992 vor. Zu Beginn erfolgt die generelle Definition und Motivation von RBAC. Es wird eine Studie erwähnt, an der 28 Organisationen teilgenommen haben und bei welcher die unterschiedlichen Sicherheitsbedürfnisse der Organisationen erhoben wurden. Diese rangierten von Kundendatenvertraulichkeit, über Privatsphärenschutz, unerlaubter Verteilung von finanziellen Anlagen, bis hin zur Einschränkung von Ferngesprächen. Interessant zu beobachten war, dass die Unternehmen ihre Zugriffsentscheidungen auf Basis von Rollen stellten, die ein Mitarbeiter als Teil des Unternehmens verkörperte. Die Unternehmen bevorzugten dabei eine zentrale Kontrolle um die Zugriffsrechte zu verwalten. Dabei erfolgte dies meist eher auf Basis einer unternehmensweiten Sicherheitspolitik, als auf den persönlichen Entscheidungen und Einschätzungen des Systemadministrators. Die Studie brachte weiters zum Vorschein, dass die Organisationen annähernd eine korrelierende Sichtweise der Sicherheitsbedürfnisse hatten und der Meinung waren, dass die gegenwärtigen Sicherheitsprodukte in Hinblick auf die Flexibilität nicht ausreichend seien.

Erwähnenswert scheint auch die strikte Trennung zwischen Kompetenz und Autorität. Wenn ein Subjekt in einem Unternehmen eine Aufgabe inne hat, also dafür autorisiert ist, so besitzt es in der Regel die Kompetenz diese zufriedenstellend durchzuführen, da ansonsten die Entscheidungen der Personalabteilungen angezweifelt werden müssten. Das bedeutet, Autorität impliziert Kompetenz. Dass jemand kompetent genug ist, eine Aufgabe zu erfüllen bedeutet aber nicht gleichzeitig, dass er auch dafür autorisiert wurde diese Aufgabe auszuführen bzw. die Verantwortlichkeit dafür trägt. So kann ein Subjekt möglicherweise kompetent genug sein, mehrere Abteilungen zu führen, tatsächlich aber nur für genau eine als Abteilungsleiter autorisiert sein. Somit herrscht keine Äquivalenz zwischen Autorität und Kompetenz. Die beiden Begriffe Kompetenz und Verantwortlichkeit bzw. Autorität werden im Sandhu et al. Model streng unter Rücksichtnahme auf ihre Relation, bzw. getrennt voneinander behandelt.

Weiters weist Sandhu et al. darauf hin, dass das Konzept RBAC noch immer keinem einheitlichen Standard unterliegt und daher die Landschaft der implementierten Systeme

hinsichtlich ihrer Funktionalität sehr durchwachsen ist. Das Paper beschreibt hierfür ein Referenzmodell, das RBAC Systeme hinsichtlich ihrer Komplexität kategorisiert und damit erstmals eine Einteilung der tatsächlich am Markt vorhandenen Systeme möglich macht.

Die Motivation für ein RBAC-System besteht neben der besseren Abbildbarkeit der Organisationsstruktur und Sicherheitspolitik auf ein Zugriffskontrollsystem vornehmlich in der Flexibilität, die ein solches System in einem sich rasch ändernden Umfeld bieten kann. Mitarbeiter kommen und gehen, neue Geschäftszweige werden erschlossen, alte versiegen, der Ablauf im Unternehmen wird als Rückkopplungsfolge der Qualitätssicherung ständig verbessert. All dies hat eine sich ständig ändernde Belegschaft, Organisationsstruktur, Ablaufverhalten und auch Sicherheitspolitik zur Folge. Die Rollen bleiben allerdings über einen gewissen Zeitraum in einem Unternehmen stabil, bzw. kommen lediglich dann neue hinzu oder werden alte entfernt, wenn sich die Organisationsstruktur bzw. das Ablaufverhalten im Unternehmen ändert. Ein Beispiel ist eine Rolle die aus Gründen des „Vier-Augen-Prinzips“ in zwei Rollen aufgeteilt werden muss, da die Unternehmens-Erfahrung bzw. die kontinuierliche Unternehmensverbesserung dies als notwendig erachtet.

Ein Administrator hat in einem DAC-System in der Regel die Möglichkeit, sich selbst Rechte zu verschaffen, da er als Benutzer auftritt und damit benutzerbestimmt sich Rechte an den Objekten selbst vergeben kann. In RBAC kann der Administrator auch Rechte vergeben bzw. entziehen, indem er die Benutzer den Rollen zuweist bzw. entfernt. Es ist aber in RBAC ein leichtes dem Administrator zu untersagen, dass er sich selbst einer Rolle zuordnen kann – entweder weil dies über Constraints gelöst wird oder er selbst nicht als System-Subjekt auftritt, also quasi als Überwachung- und Verwaltungsinstanz über dem System steht. Damit erhält er die Möglichkeit zu administrieren, ohne dass ihm die Autorität der zu vergebenden Rollen als Nebeneffekt überlassen wird. Die RBAC Komponenten können also vom System-Besitzer direkt verwaltet oder entsprechende Rollen eingeführt werden, die diese Aufgabe übernehmen. Ein oft vernachlässigter aber nicht zu unterschätzender Vorteil von RBAC liegt darin, dass es den Überblick bewahren lässt, welche Rechte an welche User vergeben wurden.

Sandhu et al. unterstreicht, dass RBAC hervorragend dafür geeignet ist, bewährte Sicherheitsprinzipien wie „Least Privilege“ und „Separation of Duty“ umzusetzen, warnt aber gleichzeitig davor, dass der Einsatz von RBAC keine Garantie für eine erzwungene Umsetzung dieser Prinzipien darstellt, da RBAC Policy neutral ist. Das bedeutet, wenn das RBAC System vom Sicherheitsbeauftragten absichtlich falsch konfiguriert wird, oder die Sicherheitspolitik mangelhaft ist, so ist es ein leichtes, diese Prinzipien zu verletzen.

Auch ist RBAC kein Allheilmittel für Zugriffskontroll-Angelegenheiten. Sandhu et al. weist darauf hin, dass RBAC eine Grundlage für komplexere Zugriffskontrollsysteme bieten kann. Beispielsweise haben die Transaktionen, die einer Rolle zugeschrieben wurden, in RBAC keine Ordnung, bilden also eine Menge. Das bedeutet, dass ein sequentieller Ablauf der Transaktionen in RBAC ohne Zusatzsysteme nicht erzwungen werden kann. Derartige Anforderungen können dann durch Erweiterungen implementiert werden, die nicht mehr zum RBAC Standard gehören.

2.3.1 Gruppen vs. Rollen

Gruppen sind ein generelles Konzept in Zugriffskontrollsystemen wie DAC und MAC. Sie stellen typischerweise eine Kollektion von Usern dar, die in gewissen Gesichtspunkten gleich behandelt werden können. Eine Rolle vereint ebenfalls eine Menge an Benutzern, akkumuliert jedoch auch noch zusätzlich Rechte. Sie stellt daher eine n zu m Beziehung zwischen Benutzern und Rechten dar.

Damit können alle Rechte, die ein bestimmter Benutzer besitzt, als auch alle Benutzer, die ein bestimmtes Recht besitzen, ohne großen Aufwand erhoben werden. In diesem Punkt hat RBAC einen erheblichen Vorteil gegenüber DAC und MAC, weil in einer Access Control List bzw. einem Capability Vector jeweils eine der beiden Abfragen einen erhöhten Aufwand darstellt, da entweder alle Objekte oder alle User traversiert werden müssen, Abbildung 2.2.

In RBAC erfolgt die Berechung folgendermaßen:

- Fall A: Es sollen alle Rechte, die ein User besitzt, erhoben werden. Suche nach Rollen, in denen der User Mitglied ist. Traversierung und distinkte Akkumulation der Operationen aller gefundenen Rollen.

- Fall B: Es sollen Alle User, die ein bestimmtes Recht besitzen, erhoben werden. Suche nach Rollen, die bestimmtes Recht bzw. Operation enthalten. Traversierung und distinkte Akkumulation der den Rollen zugeordneten Benutzer.

Obwohl Eingangs erwähnt wurde, dass der Unterschied zwischen Gruppen und Rollen darin besteht, dass Gruppen lediglich User vereinen, Rollen jedoch User und Permissions, so stimmt dies nur bedingt. Tatsächlich sind Gruppen beispielsweise in den ACLs der Objekte mit ihren jeweiligen Rechten verzeichnet. Es besteht also nur ein Unterschied im Speicherort der Permissions, die entweder beim User (Capabilities) oder beim Objekt (ACL) liegen, nicht jedoch bei der Gruppe direkt. Wenn man dynamische Komponenten außer Acht lässt und von den komplexen Vererbungs-Strategien absieht, die

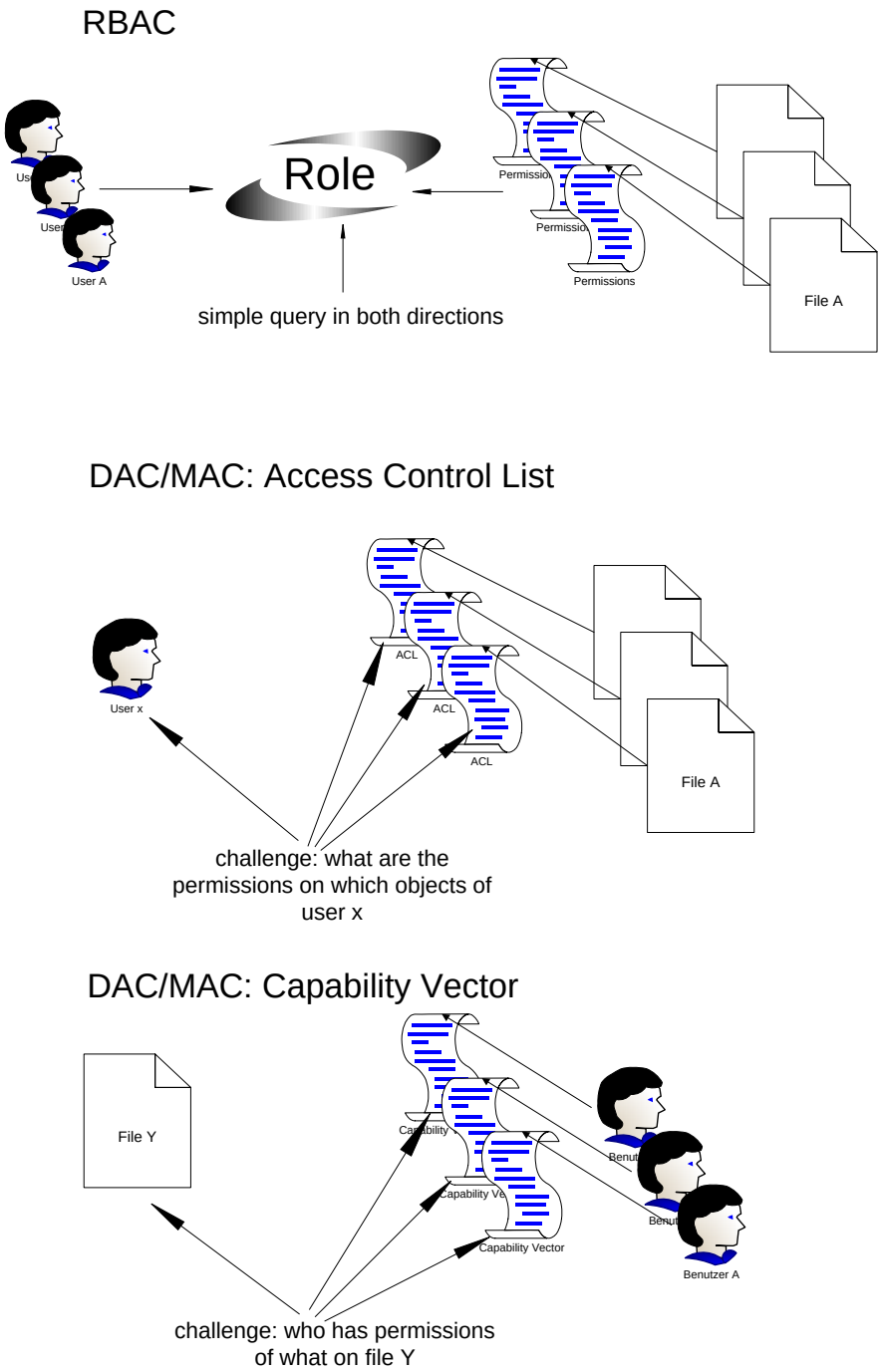


Abbildung 2.2: Berechnung der effektiven Berechtigungen

später für das Konzept „Rolle“ noch genauer erläutert werden, so unterscheiden sich Gruppen von Rollen also lediglich im Speicherort der Berechtigungen.

2.3.2 Einordnung von RBAC

Sandhu et al. geht ebenfalls auf die Analogie zu MAC ein und behandelt die oft gestellte Frage der Ähnlichkeit zwischen Rollen und Compartments. Er übernimmt die Aussage aus dem Ferraiolo-Kuhn Modell, bei welchem der essentielle Unterschied im unidirektionalen Informationsfluss-Gesetz vs. einer multidirektionalen Informationsfluss-Politik liegt. In TCSEC wird lediglich zwischen DAC und MAC Systemen unterschieden. RBAC wurde erst später entwickelt. Es drängt sich daher die Frage auf, in welche der beiden Kategorien RBAC einzuordnen ist, bzw. ob es nicht eigenständig eine völlig neue Klasse von Zugriffsystemen definiert. RBAC kann als unabhängiges Zugriffskontrollsystem gesehen werden, dass neben DAC und MAC – falls angemessen – zur Verwendung kommt. RBAC stellt also nicht die Universallösung für alle Probleme dar, ist jedoch in machen Fällen besser geeignet als DAC oder MAC. Sandhu gibt auf die Klassifizierungsfrage folgende Antwort: Ein System bei dem die Benutzer nicht exclusiv den Einfluss auf die Rechtevergabe haben, ist MAC. Daher folgt in diesem Punkt das RBAC Modell dem MAC Konzept. Andererseits spezifiziert MAC, welcher Informationsfluss nicht statt finden darf und RBAC welcher statt finden kann. Weiters existieren in RBAC Konzepte wie Delegation bei dem die Benutzer selbstbestimmt Rechte weitervergeben können, was wiederum einen DAC-Charakter aufweist.

2.3.3 Eine Familie von Referenzmodellen

Um dem Misstand, dass RBAC unter keinem einheitlichen Standard zusammengefasst war, entgegenzutreten, definierte das Sandhu et al. Model eine Familie an Referenzmodellen, die das funktionale Spektrum eines RBAC-Systems beschreiben und somit in der Lage sein sollten, es zu klassifizieren. Dabei wurden vier konzeptuelle Modelle definiert, wobei das erste Modell – RBAC0 – die Basis für alle RBAC Systeme darstellt. Auf diesem aufbauend folgen die Modelle RBAC1 und RBAC2. Jedoch stehen RBAC 1 und 2 auf einer Entwicklungsstufe und können unabhängig voneinander umgesetzt werden. Ein System, das beide Modelle implementiert, wird dann als RBAC3 bezeichnet. Es vereint also RBAC0, RBAC1 und RBAC2 und stellt somit die höchste Entwicklungsstufe der rollenbasierten Zugriffkontrollsysteme nach Sandhu dar.

Wie bereits erwähnt, ist die Basis aller RBAC Systeme das RBAC0 Modell. Jedes Zugriffkontrollsystem, das die Klassifikation RBAC nach Sandhu tragen möchte, muss daher mindestens die Spezifikation des RBAC0 Modells erfüllen. RBAC1 fügt dieser

Grundausstattung dann Rollenhierachien hinzu. RBAC2 erweitert es um Constraints. Wenn auch ein indirekter Zusammenhang zwischen Hierarchien und Constraints besteht, auf welchem später noch genauer eingegangen wird, so können diese beiden Features bzw. Stufen unabhängig voneinander umgesetzt bzw. implementiert werden. Das RBAC3 Modell ist anschließend nur mehr ein Dachverband, um nicht das Grundmodell und die beiden aufbauenden Modelle explizit erwähnen zu müssen, wenn ein System „Full-RBAC“ nach Sandhu et al. ist. Abbildung 2.3 zeigt die Zusammenhänge der vier Modelle.

2.3.3.1 Base Model – RBAC0

Das RBAC-Basis-Modell besteht aus drei verschiedenen Arten von Entitäten, den Usern U, den Rollen R und den Permissions P. Außerdem existiert eine Menge an Sessions S. Ein User ist dabei ein Mensch, wenngleich er auch zu einem intelligenten autonomen Agenten – beispielsweise Robot, Computer oder Netzwerk – generalisiert werden kann. [13]

Die Rolle entspricht dabei wieder einer Job-Funktion innerhalb einer Organisation und verleiht den Mitgliedern damit mit eine gewisse Autorität und Verantwortlichkeit. Realisiert wird diese durch Permissions, die eine gewisse Art des Zugriffs auf ein oder mehrere Systemobjekte gewähren und einer oder mehreren Rollen angehaftet sind. Permissions sind dabei immer positiv, drücken also aus, dass der Zugriff auf ein Objekt in einer bestimmten Art erlaubt ist. Objekte sind dabei Daten-Objekte, als auch Ressource-Objekte, welche wiederum durch Daten-Objekte innerhalb des Systems repräsentiert werden.

Zitat aus [13]: „Jedes System schützt Objekte der Abstraktion welche es implementiert“ .

Das bedeutet, das zu schützende Objekt ist lediglich die Abstraktion der zugehörigen Implementierung. Eine Datei besteht beispielsweise auf physikalischer Ebene aus Bits, welche elektronisch, magnetisch oder optisch repräsentiert werden können. Auf logischer Ebene kommen Konzepte wie Header, Payload, Footer, Attributwerte, Sicherheitsdeskriptoren usw. dazu. Letzten Endes ist eine Datei aber lediglich ein Eintrag in der Masterfile-Table des Dateisystems und repräsentiert somit eine Abstraktion der Implementierung. Die Kapselung der Implementierung stellt dabei ein enorm wichtiges Konzept dar. Nur wenn sichergestellt ist, dass der Zugriff auf das tatsächliche Objekt ausnahmslos über das Abstraktions-Objekt möglich ist, kann das Zugriffskontrollsystem effektiv arbeiten.

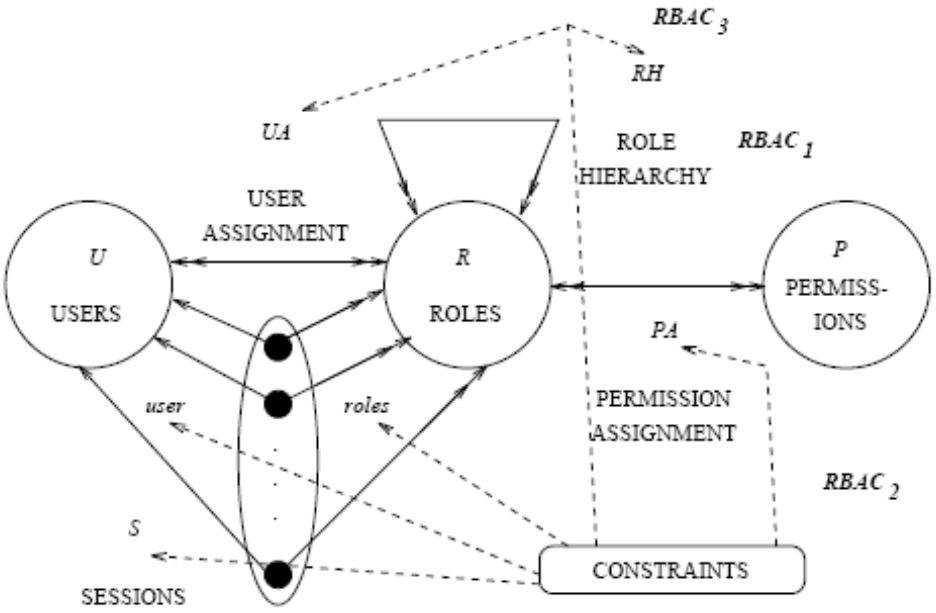


Abbildung 2.3: Users/Roles/Permissions, Quelle: [13]

Wie die Permissions konkret aussehen oder interpretiert werden, wird nicht durch das Modell beschrieben sondern ist stark von der Natur des Systems abhängig. Die Arten der Zugriffe können dabei genau so individuell sein wie die Granularität des Zielbereiches. Eine Permission kann beispielsweise den pauschalen Zugriff auf eine ganze Gruppe von Objekten oder aber nur auf ein Attributfeld eines einzigen Objektes erlauben. Für das RBAC Modell nach Sandhu sind diese Ausprägungen egal. Permissions werden als neutrale uninterpretierte Symbole sprich Variablen behandelt. Negative Permissions, so genannte Denials, werden im Sandhu Modell durch Constraints, welche Bestandteil des RBAC2 Models sind, beschrieben. Sie sind also demnach nicht in der Rolle direkt verankert.

2.3.3.1.1 Zentraler Kern

Abbildung 2.3 zeigt *UA* (User-Assignment) und *PA* (Permission-Assignment) Relationen. Beide stellen *m:n* Relationen dar. Der Schlüssel in RBAC liegt in diesen beiden Relationen. Eine Rolle ist ein Indirektionskonzept, dass einem User ermöglicht, Permissions auszuführen. Dies ermöglicht eine bessere Kontrolle über die Konfiguration und eine bessere administrative Übersicht anstatt die User direkt mit Permissions zu verbinden.

2.3.3.1.2 Session-Konzept

Eine Session ist eine Abbildung von einem User auf n Rollen. Ein User etabliert eine Session, indem er eine Teilmenge der Rollen aktiviert, in welchen er Mitglied ist. Der Doppelkopf-Pfeil von Session S zu Rolle R in Abbildung 2.3 indiziert, dass in der Session mehrere Rollen gleichzeitig aktiviert werden können. Jede Session ist mit einem Einzelkopf-Pfeil genau einem User zugeordnet, wobei ein User mehrere Sessions abhalten kann, welche wiederum unterschiedliche Teilmengen an aktivierten Rollen aufweisen können. Die Verbindungen zwischen U und S bleiben für die Lebensdauer der Session konstant. Die Verbindungen zwischen S und R können sich dynamisch innerhalb der Session-Lebensdauer ändern.

In diesen Punkten unterscheidet sich das Modell von Sandhu gegenüber dem von Ferraiolo-Kuhn, indem es den Session-Begriff einführt, der dem User erlaubt, eine oder mehrere Rollen zu aktivieren und derartige Sessions in multiplen Instanzen zu halten. Ferraiolo-Kuhn spricht in diesem Zusammenhang lediglich von der „aktiven Rolle“ $AR(subject : s)$, definiert die Funktion nicht als Teilmenge der autorisierten Rollen $RA(subject : s)$ und spricht schon gar nicht von einer multiplen Instanzierung. Das Konzept der Session ist in der Literatur der Zugriffskontrollsysteme das Äquivalent zum dem des Subjekts in einem konventionellen System, wie beispielsweise DAC. Dies resultiert daraus, dass die Session zum Zeitpunkt t das Potential eines Benutzers charakterisiert, genau wie das statische Subject in einem konventionellen System, dessen Rechte-Profil entweder über ACL oder CV charakterisiert wird. Vereinfacht gesagt reicht es im Gegensatz zu einem statischen System bei RBAC nicht, lediglich das Subject zu betrachten, um Aussage darüber treffen zu können, welche effektiven Permissions es zum Zeitpunkt t besitzt. Hierfür ist zusätzlich die Session-Information zum Zeitpunkt t notwendig.

Das Session-Konzept unterstützt erneut das „Principle of Least Privilege“. Es wird von RBAC in vielerlei Hinsicht unterstützt. Wie bereits erwähnt liegt ein wesentliches Merkmal darin begründet, dass jeder Rolle nur genau so viele Rechte zugewiesen werden, wie es die damit verbundene Unternehmens-Job-Funktion verlangt. In Zusammenhang mit dem Session-Konzept, kann der Benutzer genau die Rollenmenge aktivieren, die notwendig ist, um die ihm zugewiesenen Aufgaben erfolgreich erledigen zu können. Hier kann wieder auf die Unterscheidung zwischen Kompetenz und Autorität verwiesen bzw. das Konzept erweitert werden. Ein Benutzer kann kompetent genug sein, n Rollen in einem Unternehmen zu verkörpern. Tatsächlich wird er aber nur für m Rollen autorisiert, wobei m eine Teilmenge von n ist. Dies kann beispielsweise der Fall sein, wenn eine Trennung der Autoritäten erforderlich ist, z.B. das Vier-Augen-Prinzip. Auch wenn der Benutzer autorisiert ist, m Rollen einzunehmen bzw. zu aktivieren, so muss er dies für die Bewältigung seiner täglichen Aufgaben nicht zwangsläufig tun. Er sollte tatsäch-

lich nur eine Teilmenge o von m verwenden, um dem PoLP gerecht zu werden, wobei o genau so groß gewählt werden sollte, dass die erforderlichen Aufgaben durchgeführt werden können. Damit können unnötige und sehr mächtige Rollen von der täglichen Arbeit ausgeschlossen und nur dann aktiviert werden, wenn dies die Situation unbedingt erfordert. Wie Eingangs bereits erwähnt, bietet RBAC Konzepte, mit denen das PoLP gut umsetzbar ist. Jedoch unterliegt man einem Trugschluss, zu glauben, dass mit dem Einsatz von RBAC das PoLP ohne weiteres Zutun erkauft wird. In RBAC0 obliegt es ausschließlich dem User selbst, einen vernünftigen Umgang mit der Aktivierung seiner Rollen zu pflegen und damit dem PoLP gerecht zu werden, während dies im RBAC2 Modell durch Constraints gesteuert werden kann.

2.3.3.1.3 Management

Sandhu unterstreicht, dass in RBAC0 die Permissions, welche im Modell als neutrale uninterpretierte Symbole behandelt werden und stark von der Art der Anwendung abhängen, sich ausschließlich auf Daten- und Ressource-Objekte beziehen, nicht jedoch auf RBAC-Objekte selbst. Permissions, die es erlauben, U R, P, PA und UA zu modifizieren, werden als Administrative-Permissions bezeichnet. Diese Permissions werden im RBAC-Management-Model behandelt. RBAC0 geht davon aus, dass ein einziger Security-Officer, sprich Admin existiert, der die Komponenten des RBAC-Systems verwaltet.

2.3.3.1.4 Formalisierung

$U,\ R,\ P\ und\ S\ (Users,\ Roles,\ Permissions\ und\ Sessions)$,

$PA \subseteq P \times R,\ ein\ many-to-many\ Permission\ to\ Role-\ Assignment$,

$UA \subseteq U \times R,\ ein\ many-to-many\ User\ to\ Role-\ Assignment$,

$user : S \rightarrow U,\ eine\ Funktion,\ die\ jede\ Session\ s_i\ auf\ einen\ User\ user(s_i)\ abbildet$
$(konstant\ fuer\ die\ Lebensdauer\ der\ Session)$,

$roles : S \rightarrow 2^R,\ eine\ Funktion,\ die\ jede\ Session\ s_i\ auf\ eine\ Menge\ an\ Rollen\ roles(s_i) \subseteq$
$\{r|(user(s_i), r) \in UA\}\ abbildet,\ (was\ ber\ die\ Zeit\ hinweg\ variieren\ kann)$
$und\ Session\ s_i\ hat\ die\ Permissions\ \bigcup_{r \in roles(s_i)} \{p|(p, r) \in PA\}$.

2.3.3.1.5 Zusammenfassung

Die Permission-Rollen- und die User-Rollen-Zuordnung erfolgt statisch über einen designierten Systemadministrator. Das RBAC0-Modell unterliegt keinerlei Constraints. Dies betrifft sogar Konzepte wie ein Session-Timeout, das die Sitzung automatisch beendet, wenn für einen bestimmten Zeitraum keine Benutzerinteraktion erfolgt. Streng genommen müsste es als Constraint angesehen werden und hat somit in RBAC0 keinen Platz. Die Gestaltung der Sessions erfolgt benutzerbestimmt, jeder User kann ohne weitere Einschränkung die Rollen aktivieren, denen er zugewiesen wurde. Permissions sind Variablen, die in einem konkreten System mit Inhalt gefüllt werden. User sind im allgemeinen Menschen, können aber auch digitale autonome Agenten sein. PoLP ist in RBAC0 umsetzbar, jedoch nicht erzwingbar.

2.3.3.2 Role Hierachies - RBAC1

Da in jedem größeren Unternehmen eine Hierarchie an Autoritäten und Verantwortlichkeiten existiert, sind Rollen-Hierarchien sind ein intuitives Konzept, um diese authentisch abzubilden. Per Konvention werden mächtigere Rollen – so genannte Senior-Roles – weiter oben und weniger mächtige Rollen – die Junior-Roles – weiter unten angesiedelt. Abbildung 2.4 zeigt eine simple Hierarchie, in der der *Health-Care-Provider* die „Most-Junior-Role“ darstellt, also die Rolle, die mit der geringsten Autorität ausgestattet ist. Die Rolle *Physician* ist senior zu *Health-Care-Provider* und erbt damit alle Permissions von *Health-Care-Provider*. *Physician* kann zusätzlich zu den geerbten weitere Permissions haben. Die Vererbung ist transitiv, *Primary-Care-Physician* erbt die Permissions von *Physician* und der *Health-Care-Provider*. *Primary-Care-Physician* und *Specialist-Physician* erben beide von *Health-Care-Provider* und *Physician*, besitzen aber zusätzlich direkt zugewiesene individuelle Permissions.

Abbildung 2.5 zeigt multiple Vererbung, in der die Project-Supervisor-Role von der Test-Engineer-Role und der Programmer-Role erbt. Mathematisch sind diese Hierarchien partielle Ordnungen. Eine partielle Ordnung ist eine reflexive, transitive, antisymmetrische Relation. Die Vererbung ist reflexiv, weil eine Rolle die eigenen Permissions erbt, transitiv weil es eine natürliche Eigenschaft der Vererbung ist und antisymmetrisch, weil wenn Rolle a von b erbt und vice versa, Rolle a gleich Rolle b sein muss.

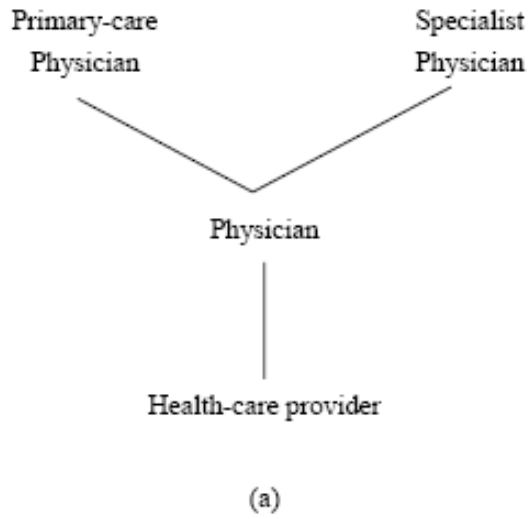


Abbildung 2.4: Rollenhierarchie, Quelle: Sandhu et al. Model, figure 2(a) [13]

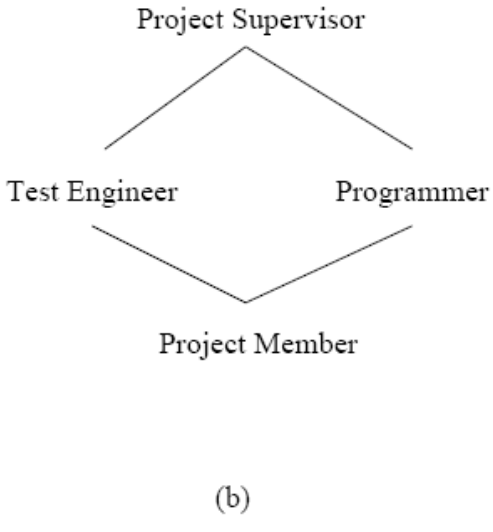
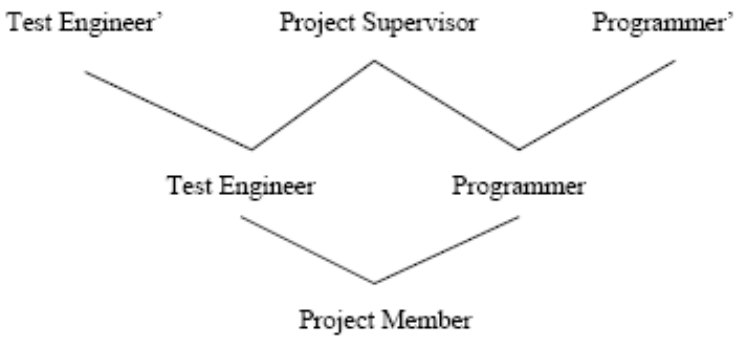


Abbildung 2.5: Rollenhierarchie, Quelle: Sandhu et al. Model, figure 2(b) [13]



(c)

Abbildung 2.6: Rollenhierarchie, Quelle: Sandhu et al. Model, figure 2(c) [13]

2.3.3.2.1 Private Roles

In manchen Fällen kann es hilfreich sein, die Reichweite der Vererbung einzuschränken. In Abbildung 2.5 ist *Project Supervisor* senior zu *Test Engineer* und *Programmer*. Für den Fall, dass *Test Engineer* einige Rechte privat halten und die Vererbung der Rechte verbieten möchte, empfiehlt [13] die Einführung von privaten Rollen. Dies geschieht wie in Abbildung 2.6 dargestellt.

Die Rollen *Test Engineer* und *Programmer* haben jeweils eine private Rolle *Test Engineer'* und *Programmer'*. Diese beiden erben von ihren Junior Rollen *Test Engineer* und *Programmer*. Test-Engineers werden der Rolle *Test Engineer'* zugewiesen, Programmers der Rolle *Programmer'* Die privaten Permissions von *Test Engineer'* sowie *Programmer'* werden somit nicht an *Project Supervisor* vererbt, bleiben also privat. Die Vererbung könnte auch mittels Constraints unterbunden werden, wäre aber dann Teil von RBAC2. Es ist jedoch für die Übersicht und damit verbundenen Wartbarkeit empfehlenswert, den hierarchischen Vererbungsmechanismus unangetastet zu lassen und eine Einschränkung der Vererbungs-Reichweite mit privaten Rollen wie in RBAC1 vorgestellt zu realisieren.

2.3.3.2.2 Formalisierung

U, R, P, S, PA, UA und die User werden unveraendert aus $RBAC0$ uebernommen,

$RH \subseteq R \times R$ ist eine partielle Ordung auf R, auch Rollenhierarchie genannt und notiert als „ $\geq$ “ und

$roles : S \rightarrow 2^R$ *aus RBAC0 modifiziert zu* $roles(s_i) \subseteq \{r|(\exists \acute{r} \geq r)\,[(user(s_i), \acute{r}) \in UA]\}$ (*was ber die Zeit hinweg variieren kann*) *und Session* s_i *hat die Permissions* $\bigcup_{r \in roles(s_i)} \{p|(\exists \tilde{r} \leq r)\,[(p, \tilde{r})) \in PA]\}$.

RBAC1 baut auf den Basiskonzepten von RBAC0 auf. Es existiert zusätzlich eine Rollenhierarchie RH, Teilmenge des Kreuzproduktes von $R \times R$, welche eine partielle Ordnung auf R darstellt. Die Definition der Session muss für RBAC1 modifiziert werden. Die Session s_i ist eine Teilmenge aller Rollen r, für die eine Rolle $\acute{r}$ existiert die größer oder gleich der Rolle r ist und dem User zugewiesen wurde. Der User kann somit eine Session mit beliebiger Kombination an Junior Rollen zur der Rolle welcher er Mitglied ist etablieren. Das bedeutet, er muss nicht zwangsläufig die Senior Rolle für eine Session auswählen, sondern kann eine weniger mächtigere Junior Rolle aktivieren. Die Permissions innerhalb einer Session sind eine Akkumulation der den aktiven Rollen direkt zugewiesenen Permissions, als auch aller Permissions der den aktivierten Rollen assoziierten Junior Rollen.

2.3.3.2.3 Zusammenfassung

Das RBAC1 Modell baut auf RBAC0 auf, verwendet die darin enthaltenen Konzepte, und führt eine Rollenhierarchie ein, welche durch eine partielle Ordnung auf den Rollen repräsentiert wird. Dadurch ergibt sich eine Modifikation des Session-Begriff da durch die Vererbung neben den explizit aktivierten auch implizite Rollen am geschehen beteiligt sind. Auch macht das Modell einen Vorschlag zur Einschränkung der Vererbungsreichweite ohne das mathematische Konzept der partiellen Ordnung und damit den Vererbungsfluss künstlich mit Constraints aufzubrechen. Der wesentliche Unterschied zum Ferraiolo-Kuhn-Model besteht in der Notation des Vererbungsbaumes und in der Einführung von privaten Rollen.

2.3.3.3 Constraints - RBAC2

Constraints sind ein wesentliches Werkzeug, um die Mächtigkeit eines RBAC-Systems zu steigern und es einer individuellen Sicherheitspolitik anzupassen. RBAC2 stellt wie bereits erwähnt keinen Fortschritt zu RBAC1 dar und vice versa. Die beiden Modelle können also jeweils mit oder ohne dem jeweilig anderen realisiert werden. Constraints finden überall dort Verwendung, wo die allgemeinen Regeln von RBAC zu Gunsten der Sicherheit aufgebrochen werden müssen. Dies kann wechselseitigen Ausschluss von Rollenzuweisungen zum Zwecke der Separation of Duty, die Durchsetzung des PoLP durch die Einschränkung von Rollenaktivierung innerhalb von Sessions und vieles mehr betreffen. Damit stellen sie einen mächtigen Mechanismus zur Umsetzung einer High Level

Organisationspolitik dar. In Folge erleichtert dies die Vergabe von administrativen Aufgaben an Mitarbeiter, da Betrugsversuche und Fehlkonfigurationen des Systems durch einmal definierte Constraints unterbunden werden und damit geringere Gefährdung der Organisations-Interessen besteht. Laut [13] sind Constraints für den Management-Bereich eine hilfreiche und bequeme Einrichtung, solange das RBAC System von einem zentralisierten Punkt aus und einem einzelnen Sicherheitsbeauftragten gewartet wird. Damit ist gemeint, dass es sehr hilfreich sein kann, wenn das System gewisse Konfigurationen nicht akzeptiert und damit den Sicherheitsbeauftragten unterstützt, die Sicherheitspolitik nicht zu verletzen. Derselbe Effekt könnte jedoch auch durch akribische Genauigkeit des Sicherheitsbeauftragten erzielt werden. Wenn das Management jedoch dezentral arbeitet und oder mehrere Mitarbeiter mit dieser Aufgabe betraut sind, so sind Constraints ein wichtiger Bestandteil zur Wahrung der Organisationsinteressen.

Auch RBAC2 baut wie RBAC1 auf RBAC0 auf und verwendet die darin enthaltenen Konzepte. Constraints können auf User Assignments *UA* und Permission Assignments *PA*, User, Rollen und Funktionen für eine Vielzahl an Sessions angewandt werden und treffen eine binäre Aussage darüber, ob die entsprechende Konstellation akzeptabel ist oder nicht.

2.3.3.3.1 Mutual Excusive Roles

Das in der Literatur am häufigsten erwähnte Constraint ist der wechselseitige Ausschluss von Operationen bzw. Transaktionen. Dabei wird gefordert, dass eine gewisse Menge an Transaktionen nicht vollständig durch ein einziges Individuum ausgeführt werden kann, weil dies zu Sicherheitsproblemen führen könnte. Die Operationen bzw. Transaktionen stehen somit im wechselseitigen Ausschluss. Genau dieses Konzept wurde bereits im Ferraiolo-Kuhn-Model vorgestellt. Permissions können auf zwei verschiedenen Wegen erlangt werden. Entweder wird ein User eine Rolle zugewiesen, so erhält er alle damit verbundenen Permissions. Oder der Rolle wird eine Permission zugewiesen, somit erhalten alle damit verbundenen User genau diese Permission. Die Constraints für den wechselseitigen Ausschluss von Operationen werden im Sandhu et al. Model über das Konzept der „Mutual Exclusive Roles“ (MER) beschrieben, können also auf das User Assignment UA oder das Permissions Assignment PA wirken. Im UA Fall kann ein User nur einer limitierten Mengen an Rollen aus einem ME-Rollen-Set zugewiesen werden, im einfachsten Fall einer. Im PA Fall betrifft dies anstatt der User die Permissions.

Als Beispiel diene eine Menge M an wechselseitig exclusiven Rollen $M = \{accounts - manager,\ puchasing - manager\}$ Die Menge M im Bezug auf UA bedeutet, dass ein beliebiger User U nicht *accounts-manager* und *puchasing-manager*

gleichzeitig sein kann, da er nur eine der beiden Rollen annehmen darf. Die Menge M in Bezug auf PA bedeutet, dass die Rollen *accounts-manager* und *puchasing manager* keine gemeinsamen Rechte aufweisen können, da ein beliebiges Recht P nur einem der beiden zugewiesen werden kann. Auf diese Weise ist doppelt sichergestellt, dass das Seperation of Duty Konzept durchgesetzt wird. Ein Verzicht auf PA-Constraints würde ein unterlaufen der UA-Constraints möglich machen und vice versa. Im einen Fall dadurch, dass die Rollen mit identischen Permissions ausgestattet werden können, im anderen, dass dem User alle notwendigen Rollen zugewiesen werden können. Eine gleichzeitige Absicherung nach UA und PA nennt sich „Dual-Constraint“ .

[13] generalisiert anschließend den Begriff der Mutual Exclusive Roles und spricht davon, dass eine Vielzahl von User-Rollen- und Permission-Rollen- Kombinationen als akzeptabel angesehen werden können oder nicht. Ein komplexeres Beispiel inkludiert den Umstand, dass ein User Programmierer und Tester niemals gleichzeitig im selben Projekt sein kann, Projektübergreifend dies aber sehr wohl möglich sein sollte.

2.3.3.3.2 Cardinality Constraints

Eine völlig andere Einschränkung betrifft die Kardialität der Rollenmitgliedschaften. Das RBAC0 Modell macht keine Einschränkungen dahingehend, wie viele User einer bestimmten Rolle zugeordnet werden können. Dies kann aber sehr wohl Sinn machen. Wenngleich es theoretisch möglich, praktisch jedoch nicht realistisch ist, so wird eine Organisation nur eine begrenzte Anzahl an Geschäftsführern, Abteilungsleiter usw. haben. Eine bestimmte Rolle wird dahingehend einschränkt, dass ihr nur eine bestimmte Anzahl an Usern zugewiesen werden können. Dabei kann eine untere wie auch eine obere Schranke existieren. [13] weist darauf hin, dass es unter Umständen schwierig sein kann, Minimal-Cardinality-Constraints umzusetzen, da immer eine gewisse Anzahl User im Unternehmen gefunden werden müssen, die kompetent genug sind, in dieser Rolle Mitglied zu sein.

2.3.3.3.3 Prequisite Roles

Bei diesem Constraint handelt es sich um eine notwendige Vorraussetzung an Kompetenz für die Zuweisung eines Users zu einer Rolle. Wie bereits erwähnt, impliziert Autorität Kompetenz. Ein User kann beispielsweise erst dann einer Rolle B zugewiesen werden, wenn er bereits Mitglied der Rolle A ist und damit die notwendige Kompetenz nachgewiesen wurde. Damit ist die Rollenmitgliedschaft von A notwendige Grundvorrausetzung für die Mitgliedschaft in Rolle B. Prequisite-Role-Constraints können zwischen Rollen bestehen, die miteinander in Vererbungsrelation stehen oder aber auch

inkomparabel sind, wobei zweite Variante in der Praxis weniger häufig vorkommt. Das bedeutet, im ersten Fall wäre A als Junior-Rolle Grundvoraussetzung für die Senior Rolle B, im zweiten Fall würde zwischen den beiden Rollen A und B mit Ausnahme des Prequisite-Role-Constraints keine Beziehung bestehen. Auch bei den Prequisite-Role-Constraints kann wie bei den MER-Constraints eine duale Anwendung erfolgen. Ein Beispiel hierfür ist, das die Zuweisung von Permission q zu einer Rolle R erst dann möglich ist wenn die Rolle bereits Permission p besitzt.

2.3.3.3.4 Session-Constraints

Constraints können auch Sessions betreffen. Es kann akzeptabel sein, dass ein User Mitglied in mehreren Rollen ist, diese jedoch nicht gleichzeitig aktivieren kann. Hier sei noch mal der Fortschritt zum Ferraiolo-Kuhn-Model erwähnt, bei dem dynamische Constraints nur anhand der User ID in Kombination der Rollenmitgliedschaft vorgeschlagen wurden, da dort kein Sessionbegriff existiert. Constraints können innerhalb einer Session, oder auch sessionübergreifend wirken. Sie können die Anzahl der aktiven Sessions für einen User limitieren oder die Session-Anzahl hinsichtlich des Vorkommens einer bestimmten Permission einschränken. Der Mannigfaltigkeit sind hierbei keine Grenzen gesetzt.

Damit die Constraints effektiv arbeiten und nicht unterlaufen werden können, ist eine Absicherung des Umfeldes notwendig. Dies betrifft einerseits die akribische Genauigkeit bei der Vergabe von User-IDs an Mitarbeiter. Die Mensch-User-Zuweisung sollte eine bijektive Abbildung sein, damit sichergestellt ist, dass jedem Mitarbeiter genau eine Identität zugewiesen wird. Andererseits besteht auch bei der Gewährung von System-Operationen durch die Permissions Potential für Fehlverhalten, wenn beispielsweise dieselbe Operation von zwei unterschiedlichen Permissions gewährt wird.

[13] erwähnt auch, dass eine Rollen-Hierarchie streng genommen als Constraint gesehen werden muss. Der Systemzwang hier lautet, dass bei einer User-Zuweisung zu einer Senior Rolle der User automatisch allen Junior Rollen zugewiesen wird. Damit ist RBAC1 in gewisser Weise ein Teil von RBAC2. Gleichzeitig räumt er aber ein, dass die Rollenhierarchie eine Berechtigung als autonomes Konzept besitzt und empfiehlt daher eine separate Behandlung.

2.3.3.3.5 Zusammenfassung

RBAC2 baut auf den Konzepten von RBAC0 auf und stellt keinen Fortschritt zu RBAC1 dar. Es führt Zwänge, sogenannte Constraints ein, mit denen die allgemei-

nen Regeln, die in einem RBAC-System herrschen, verschärft oder aufgehoben werden können. Sie stellen daher – richtig angewendet – ein mächtiges Konzept zu Wahrung von Organisations-Sicherheitspolitiken dar, deren Durchsetzung mittels Standard-RBAC nicht möglich wäre. Wichtig dabei ist, dass das grundlegende Umfeld, die Basis, des RBAC Systems unter disziplinierter Verwaltung steht und somit keinen Spielraum bietet, die Constraints an anderer Stelle zu unterminieren. Im Gegensatz zum Ferraiolo-Kuhn-Model, welches lediglich die SoD behandelt und somit einen ersten Ansatz für die Notwendigkeit von Constraints liefert, zeigt Sandhu et al. ein Vielzahl von Varianten auf, bei welchen Constraints sicherheitstechnischen Sinn ergeben.

2.3.3.4 Consolidated Model - RBAC3

RBAC3 bildet einen Dachverband aus der Basis RBAC0 und den beiden Modellen RBAC1 und RBAC2. Allerdings stellt es etwas mehr als die blose Zusammenfassung der drei vorgestellten Modelle dar, da sich bei der Kombination von RBAC1 und RBAC2 neue Konzepte und Wechselwirkungen ergeben, die als beachtenswert gelten.

Die Rollenhierarchie aus RBAC1 kann wie bereits erwähnt als Constraint angesehen werden. Die partielle Ordnung auf den Rollen ist hierbei ein intrinsisches Constraint auf das Modell selbst. Weitere Constraints können die Anzahl der Senior oder Junior Rollen betreffen, oder mehrere Rollen dahingehen eingeschränkt werden, dass sie keinen gemeinsamen Junior oder Senior haben dürfen. Alle diese neu hinzugekommenen Constraints an der Rollenhierarchie sind auch wieder dem Managementbereich zuzuordnen. Sie sind daher wieder von besonderem Interesse, falls die Management Aufgaben dezentralisiert wurden und mehrere Mitarbeiter die Aufgabe der Administration übernehmen, der Leiter der Sicherheit aber seinen Untergebenen keine unlimitierte Rollenhierarchieänderung gestatten möchte.

Weiters ergeben sich zwischen Hierarchien und Constraints aufgrund der Fusion beider Modelle Diskrepanzen, die gelöst werden müssen. Als Beispiel dienen zwei Rollen, die eine gemeinsame Senior Rolle besitzen und mutual exclusiv deklariert werden. Dann existiert genau hier ein Widerspruch zwischen dem MER-Constraint und der Vererbungsrelation, weil ein User, der der Senior Rolle zugewiesen wird, eine Verletzung des wechselseitigen Ausschlusses der beiden Junior Rollen impliziert. In ähnlicher Weise tritt eine Fragestellung bei den Kardinalitäts-Constraints auf. Wie behandelt man Zuweisungen zu Senior Rollen hinsichtlich der Kardinalität, denn tatsächlich erwirbt man mit der Rollenmitgliedschaft einer Senior Rolle auch die Mitgliedschaften aller korrespondierenden Junior Rollen. Falls Kardinalitätsbeschränkungen seitens der User oder der Rollen bestehen, so müssen auch hier Regeln spezifiziert werden, wie derartige Nebeneffekte behandelt werden. Neben dem Vorteil der Vererbungsrelations-Konsistenz

weist Sandhu gleichzeitig darauf hin, dass durch die Verwendung von privaten Rollen auch gleichzeitig das Diskrepanzproblem der MER-Constraints und Rollenhierarchie auf effiziente Art und Weise gelöst werden kann. Grund für diese dankbare Eigenschaft der privaten Rollen ist, dass sie im Vererbungsbaum immer Blätter statt Knoten, und somit maximale Elemente darstellen, die nicht weiter vererbt werden. Damit ist ausgeschlossen dass sie einen gemeinsamen Senior besitzen können. In Folge dessen sind MER-Constraints auf privaten Rollen nie konfliktbehaftet.

2.3.3.4.1 Zusammenfassung

RBAC3 ist mehr als die Summe seiner Teile, da zwischen diesen Wechselwirkungen existieren, die bedacht werden müssen. Es wartet als „Full RBAC nach Sandhu et al.“ mit einer Fülle an neuen Überlegungen und Konzepten gegen über dem Ferraiolo-Kuhn-Model aus dem Jahr 1992 auf. Die Vererbungsstrategien sind überdachter und werden mit Constraints verbunden. Das Session-Konzept wurde einführt und mit ihm die Notwendigkeit der Überarbeitung von Vererbungs- und Constraint-Definitionen.

2.3.3.5 Management Models

Neben den vier Modellen von Sandhu führt das Paper ein fünftes Modell ein, welches für die Wartung des RBAC-Systems vorgesehen ist. Die Umsetzung eines derartigen Modells macht lediglich dann Sinn, wenn eine entsprechende Menge an Rollen und Usern vorhanden ist und die Administration von mehreren Mitarbeitern durchgeführt wird. Für kleine und mittlere Betriebe, die nicht mehr als einen Sicherheitsbeauftragten und in der Regel wenig Mitarbeiter besitzen, ist die Umsetzung des Management-Modells eindeutig unwirtschaftlich, da dem Leiter der Sicherheit, der auch gleichzeitig Administrator ist, letzten Endes sowieso vertraut werden muss, bzw. die Schuldfrage im Fall einer Fehlkonfiguration eindeutig ist.

Da RBAC hervorragend dafür geeignet ist, den Zugriff auf Objekte unter der Voraussetzung einer Jobfunktion eines Users zu steuern, drängt sich die Substitution des Terminus „Objekt“ durch das „RBAC-System“ selbst und „Jobfunktion“ durch *Administrator* förmlich auf. Die Frage, die Sandhu stellt, lautet also: „Wie kann RBAC dazu verwendet werden, um sich selbst zu verwalten?“ . Außerdem spricht er die Überzeugung aus, dass der künftige Erfolg von RBAC maßgeblich von der effizienten Umsetzung eines korrespondierenden Management-Models abhängt.

2.4 ANSI/INCITS Standard

Die beiden prädominierende Modelle, Ferraiolo-Kuhn und Sandhu haben in der Vergangenheit bereits hervorragende Arbeit auf dem Weg zur Standardisierung von RBAC Systemen geleistet. 2004 wurde durch die Zusammenführung dieser beiden Modelle der ANSI/INCITS Standard [8] veröffentlicht. Er dient bis heute als Leitfaden für die Implementierung von RBAC Systemen. Einerseits um Termini zu spezifizieren, die eine gemeinsame Kommunikation zwischen Herstellern und Kunden ermöglichen und die formale Spezifikation zu erleichtern, andererseits um ein Mindestmaß an Funktionalität von RBAC-Systemen zu gewährleisten und somit den RBAC-Markt transparenter und für den Kunden verständlicher zu gestalten.

2.4.1 Funktions-Familien

Funktionen, die ein RBAC System aufweisen muss, um den ANSI/INCITS Standard zu erfüllen, können in 3 Kategorien aufgeteilt werden.

- *Administrative Operations* Diese Funktionen dienen dem Zweck, die RBAC-Elemente und ihre Beziehungen zueinander zu verwalten. Ein Beispiel ist eine Funktion, die einen User einer Rolle zuordnet bzw. entfernt. Sie werden daher ausschließlich von Mitarbeitern, die der Sicherheitsabteilung angehören, zu administrativen Zwecken verwendet.

- *Administrative Reviews* Diese Funktionsgruppe stellt ebenfalls ein wichtiges Hilfsmittel für Administratoren dar. Mit ihrer Hilfe können die Beziehungen zwischen den RBAC-Elementen effizient abgefragt werden. Dieses Monitoring ist wichtig, um den Überblick der Elemente zu behalten und redundante Elemente sowie Zuweisungen zu vermeiden. Ein Beispiel hierfür ist eine Abfrage auf die Rollenmitgliedschaften eines Users.

- *System Level Functionality* Wie im Titel bereits ersichtlich, wird diese Funktionsgruppe ausschließlich vom System verwendet, um den konditionalen Zugriff auf ein Objekt zu prüfen. Sie stellt daher das Herzstück von RBAC dar, ist Entscheidungsgrundlage für den Referenzmonitor und bedient auch sonstige dynamische Mechanismen des RBAC Systems, wie beispielsweise die Session-Verwaltung.

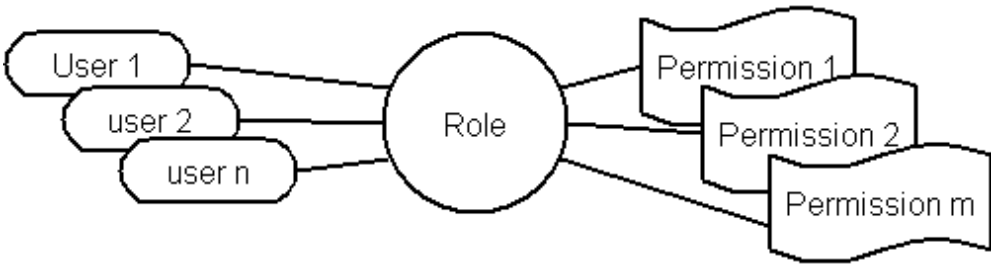


Abbildung 2.7: User und Permissions, Quelle: [14]

2.4.2 Klassifikation

Der ANSI/INCITS Standard hat die Klassifikation der RBAC-Systeme von [13] teilweise übernommen die Namen jedoch modifiziert. RBAC0 wurde zum RBAC Core Set. Dies ist der Kern, den jedes RBAC System implementieren muss, um den Namen RBAC nach ANSI tragen zu dürfen. RBAC1, welches sich mit Rollenhierarchien befasst, wurde zu RBAC Role Hierarchies. RBAC2 wurde aufgrund des Umfangs in 2 Teile aufgespalten, Static Separation of Duty (SSD), Dynamic Separation of Duty (DSD). Für RBAC3 existiert im ANSI Standard keine Entsprechung.

Da die vier Teile des ANSI/INCITS Standards bereits in der Praktikumsarbeit [14] detailliert erläutert wurden, seien hier nur mehr die wesentlichen Unterschiede zu Ferraiolo-Kuhn bzw. Sandhu et al. erwähnt.

2.4.2.1 RBAC Core Set

Der ANSI Standard übernimmt die Definition der Basiselemente aus Ferraiolo-Kuhn und Sandhu et al. unverändert. Das grundlegende Paradigma, dass Rollen das Bindeglied zwischen Usern und Permissions darstellen, wurde auch im ANSI/INCITS Standard übernommen, Abbildung 2.7.

Auch das Konzept der Sessions wurde aus dem [13] Modell unverändert übernommen und stellt die Aktivierung einer oder mehrerer Rollen aus dem Fundus des Users dar. Jede Session ist einem einzigen User zugeordnet, der User kann aber beliebig viele Sessions abhalten. Eine Session stellt also eine Relation zwischen einem User und einem Subset an Rollen, die diesem User zugewiesen wurden, dar. Die in diesem Subset befindlichen Roles gelten dann als aktiv, Abbildung 2.8.

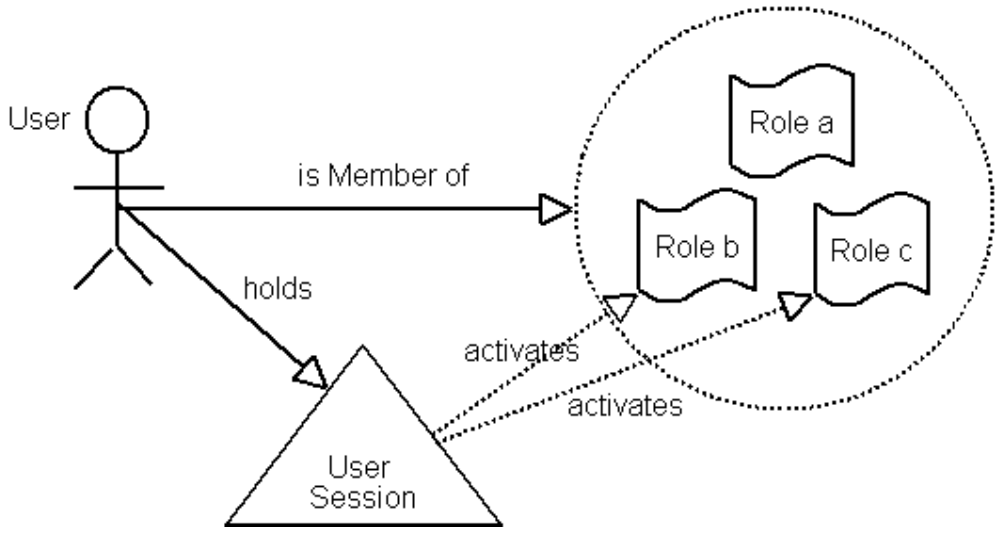


Abbildung 2.8: Session Management, Quelle: [14]

2.4.2.2 RBAC Role Hierarchies

Der ANSI/INCITS Standard unterscheidet im Gegensatz zu Sandhu et al. zwischen zwei unterschiedlichen Varianten der Hierarchiegestaltung, der Inheritance- und der Containment- Hierarchie.

2.4.2.2.1 Containment-Hierarchie

Die Containment-Hierarchie ist vollständig identisch zum Hierarchiekonzept von Sandhu et.al. Da in verteilten RBAC Systemen auch die Rollen meist dezentral definiert und verwaltet werden, sind Containment-Relationen dort das besser Mittel. Diese besagen, dass wenn eine Rolle r2 von einer Rolle r1 erbt, für alle Mitglieder von r2 gilt, dass sie Mitglied in r1 sind. Das bedeutet, dass bei einem User-Role-Assignment der User automatisch Mitglied den korrespondierenden Junior Rollen wird, damit diese Forderung nicht verletzt wird, Abbildung 2.9.

$$r2 inherits r1 \rightarrow (\forall user : u,\ member - of\ r2\,[u\ member - of\ r1])$$

2.4.2.2.2 Inheritance-Hierarchie

Eine andere Möglichkeit besteht in der Inheritance-Hierarchie. Wie die konkrete Implementierung aussieht, überlässt das Modell dem Entwickler. Wesentlich ist, dass bei der Definition einer erbenden Rolle die Permissions aller korrespondierenden Junior Rollen vorhanden sein müssen. Dies kann explizit durch redundante Verankerung der Junior-

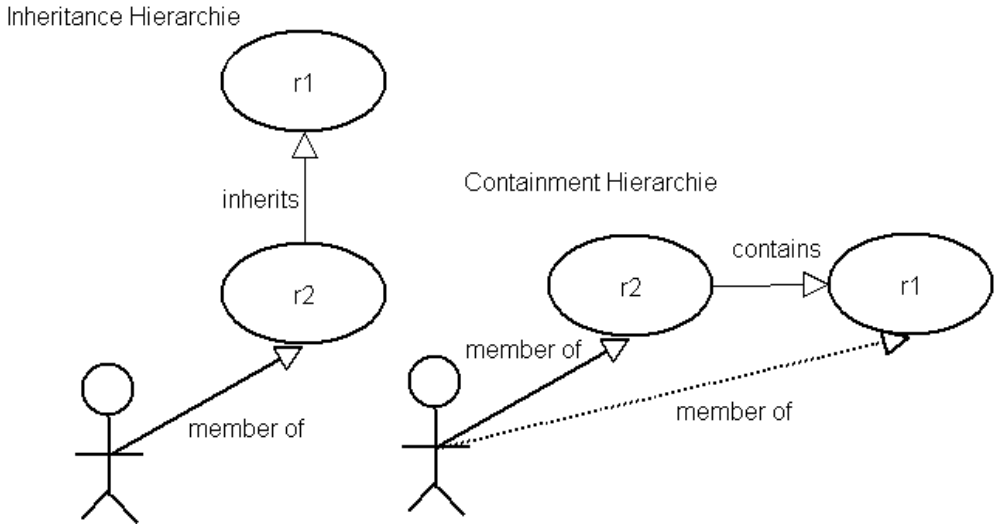


Abbildung 2.9: Hierarchie Varianten, Quelle: [14]

Permissions in der neuen Rolle geschehen, oder implizit durch eine Traversierung des Hierarchie-Baumes bis zur Wurzel bei der Berechnung der Permissions. Die Kernaussage der Inheritance-Hierarchie lautet, dass wenn eine Rolle r2 von einer anderen Rolle r1 erbt, für alle Permissions die in r1 vorhanden sind gilt, dass sie auch in r2 vorhanden sind, Abbildung 2.9.

$$r2 inherits r1 \rightarrow (\forall permissions : p,\ assigned - to\ r1\ [p\ assigned - to\ r2])$$

2.4.2.2.3 General- und Limited-Role-Hierarchies

Der ANSI Standard beschreibt weiters zwei Arten von Rollen-Hierarchien, die General- und Limited-Role-Hierarchie.

General-Role-Hierarchies unterstützen das Konzept der multiplen Vererbung. Das bedeutet, Rollen können eine oder mehrere Rollen erben und einer oder mehrerer Rollen vererbt werden. Der Vorteil daraus ist, dass es eine sehr fein granulare Rollen-Modellierung erlaubt und die Rollenobjekte hinsichtlich der Permissions wenig Redundanz aufweisen. Es kann damit ein Set an Basis-Rollen aufgestellt werden, aus denen später die Rollen, welche einer Job-Function entsrpechen, modelliert werden. Damit weisen die Rollen sehr geringe bis keine Redundanzen in den Permission-Zuweisungen auf. Ein weiterer Vorteil der multiplen Vererbung ist die Hierarchie-Notation der User-Role-Zuweisungsbeziehung und der Role-Role-Vererbungsbeziehung.

Limited Role Hierarchies erlauben einer Rolle zwar mehrere Vorgänger, beschränken sich jedoch auf einen direkten Einzelnachfolger. Dadurch kann ein User-Role nicht in

der Vererbungsstruktur eingebunden werden, da ansonsten jeder Rolle nur ein einziger User zugeordnet werden könnte. Mehrere transitive Nachfolger sind jedoch auch in der Limited- Role-Hierarchie gestattet.

2.4.2.3 Constraints

Der ANSI/INCITS Standard befasst sich bei den Constraints lediglich mit dem Konzept der „Separation of Duty“. Die System-Zwänge betreffen daher vorwiegend das User-Role-Assignment, welche über MER-Constraints realisiert werden. Auch das Konzept der „Dual-Constraints“ wurde nicht explizit in den ANSI Standard übernommen. Es macht jedoch Sinn, die Dual-Constraint-Umsetzung von Sandhu ins Auge zu fassen, auch wenn sie kein integrativer Bestandteil von ANSI/INCITS ist, da sie der Sicherheit zuträglich ist.

Separation of Duty im Allgemeinen soll sicherstellen, dass die Pflichten eines Mitarbeiters im Unternehmen streng von denen der restlichen Mitarbeiter abgegrenzt werden. Wie bereits erwähnt, impliziert Autorität Kompetenz, jedoch nicht umgekehrt. Ein User kann aus Gründen der Unternehmenssicherheit in der Verkörperung unterschiedlicher Unternehmens-Rollen eingeschränkt werden, auch wenn er die Kompetenz hätte, sie auszuführen. Was auf den ersten Blick wie eine Verschwendung von menschlichen Ressourcen aussieht, dient dazu, die Organisations-Sicherheitspolitik durchzusetzen. Man denke dabei an Bereiche, in denen es wichtig ist, dass kein Mitarbeiter in der Lage ist, im Alleingang sensitive Operationen durchzuführen, die dem Unternehmen schaden zuführen könnten. Bei einer unlimitierten Mannigfaltikeit der Rollen-Verkörperung seitens eines Single-Users wäre dies jedoch möglich.

Auch im dynamischen Bereich ist Separation of Duty im ANSI/INCITS Standard oberste Prämisse. Sie baut wie im Vorgängermodell auf den User-Sessions auf. Es wird also nicht mehr auf statischer Ebene entschieden, ob der Administrator das User-Role-Assignment durchführen darf sondern auf dynamischer Ebene innerhalb der Session oder auch Session übergreifend geregelt, ob der User die statisch zugewiesene Rolle aktivieren darf. Grundlage für diese Entscheidung ist wieder die Organisations-Sicherheitspolitik, die entscheidet, welche Rollen gemeinsam eine inakzeptable Aktivitäts-Kombination liefern.

Kapitel 3

Windows Security

3.1 Windows NTFS Security Grundkonzepte

Das Betriebsystem Windows 7 baut wie seine Vorgänger der NT Gruppe auf dem NTFS Dateisystem auf, in welchem auch die Rechte-Verwaltung umgesetzt ist. Daher ist die nachfolgende Erläuterung der Funktionsweise sowohl für ältere auf dem NT-Kernel basierende Systeme wie Windows NT, Windows 2000, Windows XP und Windows Vista, als auch das aktuelle Client-Betriebsystem Windows 7 und das Server-Betriebsystem 2008 zutreffend.

3.1.1 Security Identifier

Jeder Windows Benutzer besitzt einen sogenannten SID, den Security Identifier, da der Benutzername alleine kein invariant eindeutiges Unterscheidungsmerkmal darstellt. Auch wenn der Benutzername geändert wird, so bleibt die SID erhalten, ist also mit dem Benutzerkonto auf Lebenszeit verbunden. Wenn ein Konto gelöscht wird, so erlischt auch die SID und damit auch alle Berechtigungen die das User Konto auf den Systemobjekten hatte. Diese SID setzt sich zusammen aus einem S gefolgt von der Revisionsnummer (für NT und 2000 1), der Identifier Authority (für NT und 2000 immer 5), der Domänen ID und der User ID [9].

3.1.2 Security Descriptor

Jedes Objekt im NTFS besitzt einen dazugehörigen Security Descriptor. In diesem ist verzeichnet, für welche Benutzer bestimmte Operationen auf dem Objekt ausführbar

sind, bzw. verweigert werden. Es handelt sich dabei also um eine reine ACL in einem Discretionary Access Control System. Genau genommen werden nicht die Benutzer in die ACL eingetragen sondern ihre zugehörige SID.

3.1.3 Gruppen-Konzept

Gruppen – welche ebenfalls einen SID besitzen – werden angelegt, wenn für eine bestimmte Menge an User eine Gleichbehandlung Sinn macht. Das bedeutet, wenn eine Gruppe an Benutzern identische Zugriffsberechtigungen auf ein oder mehrere Dateien besitzen soll, so wäre es eine redundante und damit fehlerbehaftete Vorgehensweise, jeden Benutzer separat in die ACL einzutragen. Auch würden Berechtigungsänderungen eines Users enormen Aufwand verursachen, da alle Objekte bedacht werden müssten, an denen der User Permissions besitzt. Stattdessen werden User einer Gruppe zugeordnet und diese Gruppe wird mit den notwendigen Permissions in den ACLs der Objekte eingetragen. Gruppen können selbst Mitglieder in Gruppen sein, also beliebig tief geschachtelt werden, wobei im Sinne der Übersichtlichkeit eine gewisse Tiefe – im Allgemeinen drei Ebenen – nicht überschritten werden sollte. Gruppen haben wie im Paper von [6] und im Kapitel 2 erläutert eine hohe Ähnlichkeit zu Rollen.

3.1.4 Permission-Vergabe

Mit einem Rechtsklick auf ein Objekt – in der Regel eine Datei oder einen Ordner – wählt man den Kontextmenüeintrag „Eigenschaften“ und dort das Tab „Sicherheit“. Das erschienene Fenster, Abbildung 3.1, bietet im oberen Bereich die Möglichkeit, Benutzer hinzuzufügen bzw. zu entfernen. Im unteren Abschnitt kann aus einer Reihe an Basis-Zugriffs-Modalitäten gewählt werden, welche entweder erlaubt oder verweigert werden können.

Diese umfassen wie in Abbildung 3.1 ersichtlich:

- Vollzugriff
- Ändern
- Lesen, Ausführen
- Lesen
- Schreiben

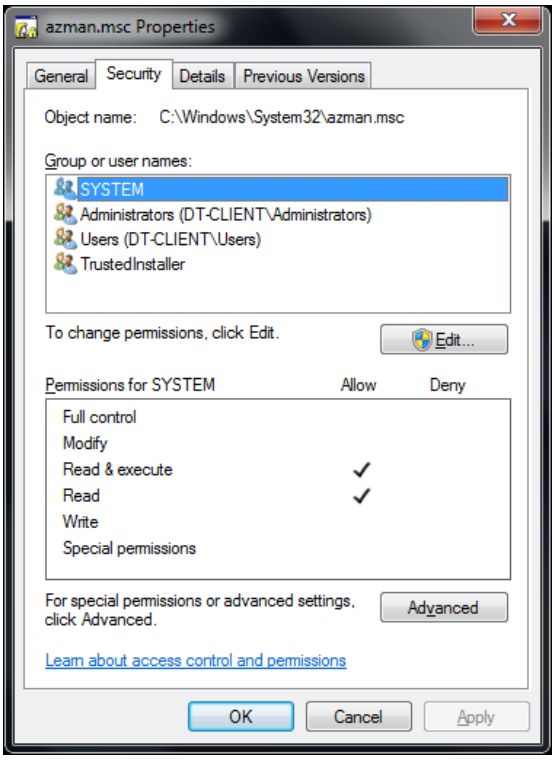


Abbildung 3.1: NTFS Security

- Spezielle Berechtigungen

Die Checkbox der Speziellen Berechtigungen kann über diese Maske nicht ausgewählt werden. Sie wird vom System automatisch gesetzt, wenn über die Grundberechtigungen hinaus speziellere gewährt werden. Ein Klick auf den Button „Erweitert“ bringt die erweiterten Sicherheitseinstellungen zum Vorschein.

In diesen können ebenfalls User der ACL hinzugefügt, entfernt und ihre Berechtigungen bearbeitet werden. Ein Klick auf den Button „Bearbeiten“ listet eine wesentlich feingranularere Auswahl auf. Diese sind die Bausteine der Basis-Permissions und können auch einzeln ausgewählt werden. Die möglichen Permissions beziehen sich auf den Inhalt der Datei, die Attribute und den Security-Descriptor. Ein Vollzugriff erlaubt auf der Datei jegliche Operationen. Einen Ordner zu durchsuchen ist äquivalent zur Ausführung einer Datei. „Ordner auflisten“ und „Datei lesen“ erlauben einen Blick auf die im Objekt enthaltenen Daten [9].

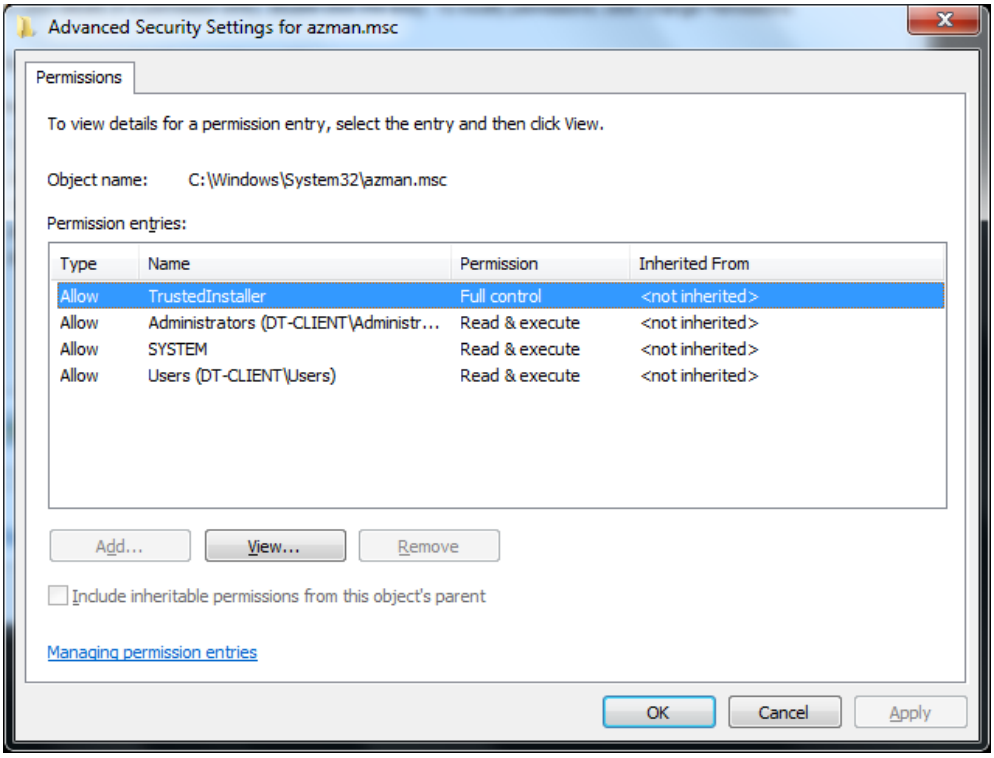


Abbildung 3.2: Advanced NTFS Security

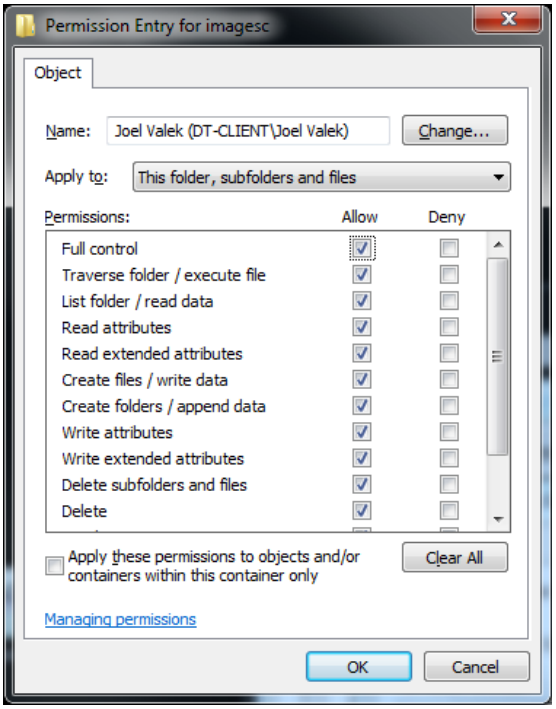


Abbildung 3.3: NTFS Security Entry

3.1.5 Vererbung

3.1.5.0.1 Rechte-Vererbung

Ein Objekt in NTFS erbt in der Standardeinstellung die Zugriffsberechtigungen seines übergeordneten Objektes. Diese Standardeinstellung macht sich mit der angehakten ersten Checkbox im unteren Teil der erweiterten Sicherheitseinstellungen, Abbildung 3.2, bemerkbar. Mit diesem Haken nimmt das Objekt die Erbschaft seines Vaterobjekts an. Wird der Haken entfernt, so gilt dies als Ablehnung der gesamten Erbschaft. Man kann dann entscheiden, ob die bis dato existenten Einträge in der Liste verbleiben oder entfernt werden sollen. Die zweite Checkbox existiert nur für Container-Objekte, sprich Ordner, und erzwingt die Vererbung auf Kinder-Objekte, selbst wenn diese die Erbschaft abgelehnt haben. Sie dient sozusagen als Wechsel-Schalter für die erste Checkbox aller Kinder-Objekte. Sollten diese die Vererbung abgelehnt haben, so kann sie ihnen durch die Aktivierung der zweiten Checkbox im Container-Objekt wieder aufgedrängt werden.

Das Tab „Überwachung“ ist für das Logging zuständig und daher hier nicht weiter von Bedeutung. Im Tab „Besitzer“ kann man den Besitzer der Datei einsehen und auch ändern, wenn man das notwendige Recht dafür besitzt. Windows unterscheidet zwischen „Rights“ und „Permissions“, deutsch „Rechte“ und „Erlaubnis“ bzw. „Berechtigung“. Ein Right ist ein unveränderliches Privileg, das ein bestimmter Benutzer besitzt, während Permissions dem User zugewiesen oder entzogen werden können. Administratoren besitzen das Right „Take Ownership“, deutsch „Besitzrechte übernehmen“, für alle Ordner und Dateien. Da Windows Access Control ein DAC System ist, kann der „Creator/Owner“ deutsch „Ersteller-Besitzer“ der Datei über die Permission-Vergabe selbst entscheiden.

3.1.5.0.2 Gruppen-Vererbung

Auch Gruppen können vererbt werden. Die Vererbungsrelation wird hier über eine „Member-Of“ Relation realisiert. Das bedeutet, wenn Gruppe A Mitglied von Gruppe B ist dass Gruppe A die Permissions von Gruppe B erbt. Die effektiven Berechtigungen, die ein User besitzt, ergeben sich durch die Summer aller Permissions (Allows) die ihm direkt oder über seine Gruppenmitgliedschaft und indirekt über die Gruppenhierarchie zugewiesen wurden, abzüglich aller Verweigerungen (Denies), die ihm direkt, über Guppen-Mitgliedschaft und die Gruppenhierarchie zugewiesen wurden. Eine Ausnahme bilden vererbte Denies, die durch lokale Allows aufgehoben werden. Da diese Struktur sehr komplex werden kann und daher nicht immer trivial zu durchschauen ist, bieten die „Erweiterten Sicherheitseinstellungen“ unter dem Tab „Effektive Berechtigungen“ die

Möglichkeit, die tatsächlichen Permissions eines Users auf dem betreffenden Objekt berechnen zu lassen.

3.2 Active Directory Grundkonzepte

Die bisher vorgestellten Sicherheitseinstellungen und Konzepte funktionieren für lokale Computer, als auch für kleine Netzwerke einwandfrei. In Unternehmen besteht aber meist die Notwendigkeit, Ressourcen auszutauschen und zu teilen. Hier hilft das Konzept der Arbeitsgruppe bzw. in Windows 7 das Konzept der Heimnetzgruppe. Wie der Name schon verrät, sind diese Konzepte für den Heimanwender-Bereich gedacht. In privaten und kleinen Firmennetzwerken kann es noch ausreichend sein, die Computer über eine Arbeitsgruppe miteinander zu verbinden. Werden Dateien auf einem Rechner freigegeben, so muss sich der Benutzer mit einem Konto anmelden, das für ihn auf dem Server-Host-Rechner bereit gestellt wurde. Der Zugriff auf die Ressource erfolgt über einen UNC String, der den Namen des Hostcomputers und das Verzeichnis beinhaltet, auf dem die Daten liegen. Bsp. `\\MyHomeServer\\public\\documents`. Beim Etablieren der Session wird der Benutzer angewiesen, seinen Benuzternamen und das Passwort anzugeben. Wenn man von der „einfachen Dateifreigabe“ in Windows XP absieht, bei der die Verwendung des Gastkontos erzwungen wird, so wird die Vorgehensweise der Kontenbereitstellung am Server-Host mit steigender Anzahl von Clients und Server immer komplizierter. Man denke an ein Beispiel in dem jeder Client zugleich Server ist, weil er im Firmennetz Daten anderen Usern zur Verfügung stellen möchte. Bei n Computer ergeben sich (n*n-1) Konto-Eintragungen an den Server-Hosts, die auch gleichzeitig Clients sind. Neben der unübersichtlichen Struktur gilt es auch Überlegungen anzustellen, welche Konsequenzen der Ausfall eines Servers hätte. Die Lösung für dieses Problem stellt ein zentraler Fileserver mit Userkonten für alle zugreifenden Clienten dar, auf dem auch die Datensicherung effizienter durchgeführt werden kann. Jedoch exisiteren in der Realität auch noch andere Server wie Webserver, Fileserver, Printserver, usw. und dies oft in redundanter Ausfertigung. Egal ob diese physisch oder virtuell sind, auf jedem dieser Server ist ein Authentifizierungsprozess notwendig. Damit müssen wieder alle Benutzer-Konten auf allen Servern eingetragen und auch gewartet werden. Die geforderte Lösung wäre also dass die Benutzerkonten für ein Firmen-Netzwerk lediglich ein einziges Mal definiert werden müssen und dann nur eine Anmeldung notwendig ist – ein „Single-Sign-On“ – bei dem alle Services und Ressourcen ohne erneute Authentisierung genutzt werden können, egal auf welchem Server sie im Netzwerk liegen. Diese Anforderung und noch vieles mehr kann durch die Domänenverwaltung und dem Active Directory von Windows Server bewerkstelligt werden. Im Nachfolgenden werden die Grundkonzepte der Domänenverwaltung und dem Active Directory von Windows Server 2008 erläutert. Die Informationen hierfür entstammen aus dem Buch „Windows Server 2008 R2“ [1].

3.2.1 Sites, Domains & Controller

Zu Beginn steht die Analyse der physikalische Struktur des Unternehmens – die Einteilung in Sites. Eine Site stellt eine Menge an Computern dar, die über eine schnelle Netzwerkverbindung miteinander verbunden sind. Man unterscheidet dabei zwischen Intra-Sites und Inter-Sites. Intra-Sites werden über LAN realisiert, sind daher kostengünstig und weisen hohe Datenübertragungsraten auf. Inter-Sites nutzen WAN Verbindungen, sind daher teurer, langsamer und sicherheitstechnisch bedenklicher. Die Gliederung in Sites stellt eine von mehreren Entscheidungs-Grundlagen für die Erstellung von Domains dar. Sites müssen aber nicht zwangsläufig mit der Domänengliederung einhergehen. So kann es sein, dass sich eine Domäne über mehrere Sites erstreckt. Jede Domäne benötigt mindestens einen Domain-Controller (DC), um arbeitsfähig zu sein, da sich jeder Client bei der Anmeldung zu einem DC seiner Domäne verbinden muss. Es stellt sich bei Domänen, die sich über mehrere Sites erstrecken, die Frage, ob mehrere DCs nötig sind oder die Bandbreite ausreichend ist, lediglich einen DC innerhalb der Domain zu betreiben. Aus Gründen der Ausfallsicherheit wird man in der Regel jedoch immer zwei – sprich einen Domaincontroller und einen Backupdomaincontroller – betreiben. Die Objekte innerhalb einer Domäne sind neben den DCs in der Regel Server-Computer, Client-Computer, Users, Groups, und Organisational Units, welche im Nachfolgenden noch näher erläutert werden. Ein wesentlicher Unterschied zu NT4 Domänen ist, dass die Änderungen auf jedem Domain-Controller getätigt werden können. Domaincontroller und Backupdomain-Controller sind also gewissermaßen gleichberechtigt.

3.2.2 Tree

Wenn die Organisation eine entsprechende Größe erreicht hat, so ist unter Umständen eine Domain nicht mehr ausreichend. Die Gründe für die Aufteilung eines Unternehmens in mehrere Domains reichen von der Site-Struktur, also den geographischen und physikalischen Grundlagen, der Mitarbeiteranzahl, der Unternehmensstruktur und Hierarchie bis hin zu Administrationsgründen. Sind mehrere Domänen erforderlich, so werden diese über einen Baum (Tree) organisiert. Ein Tree beinhaltet also mehrere Domänen, die über ihren einheitlichen Namensraum hierarchisch organisiert sind. Wichtig dabei ist, dass die Domänen trotz hierarchischem Arrangement autonome Verwaltungseinheiten bleiben. Ein Administrator einer Domäne erhält somit nicht automatisch Administratoren-Rechte an den Subdomänen. Auch existiert keine Vererbung von Gruppenrichtlinien.

Domänen, die über einen Tree arrangiert sind, vertrauen untereinander automatisch. Es ist also kein explizites Angeben von Trusts notwendig. Das bedeutet, jeder User,

der Rechte in einer Domain innerhalb des Trees besitzt, kann ohne weitere Authentifikation versuchen, auf die Ressourcen zuzugreifen, wenn er von seiner Domäne bereits authentifiziert wurde. Die Domänen vertrauen also untereinander, dass der Authentifikationsprozess zuverlässig und korrekt durchgeführt wurde. Ob ein User aus Domain A in einer anderen Domain B für den Zugriff auf eine dort enthaltene Ressource autorisiert ist, hängt aber letztlich von der Rechtevergabe innerhalb der Domain B ab.

3.2.3 Forrest

Für große Organisationen, meist Konzerne, ergeben sich durch die Größe weitere Möglichkeiten der Struktur-Verwaltung. So können die aus den Domänen entstandenen Trees zu einem Forrest zusammengefasst werden, indem ein Vertrauensverhältnis, so genannte „Trust“ zwischen den Trees definiert werden. Der Unterschied zum Tree besteht darin, dass ein Forrest keinem einheitlichen Namensraum unterliegt. Die enthaltenen Trees sind also hinsichtlich ihrer Namensräume autonom.

3.2.4 Organisational Units

Innerhalb einer Domäne hat man die Möglichkeit, weiter zu untergliedern. Bewerkstelligt wird dies durch so genannte Organisational Units (OU). In diesen OUs können sich User, Gruppen, Computer und andere OUs befinden. Eine OU stellt also eine Verwaltungseinheit in einem Unternehmen dar. Auf den ersten Blick weisen OUs eine Ähnlichkeit zu Gruppen auf, jedoch existieren zwischen den beiden Konzepten signifikante Unterschiede. Wesentlich ist, dass auf OUs so genannte Group Policy Objects (GPO) angewendet werden. Diese GPOs können dazu verwendet werden, die Konfigurationseinstellungen für die in der OU enthaltenen User, Groups und Computer anzupassen. Weiters können ausführbare Skripte in Abhängigkeit der OU-Zugehörigkeit definiert werden. Es kann ein Benutzer daher nur Mitglied in einer OU sein, weil es ansonsten zu Vorrangs-Konflikten bei der Anwendung von GPOs kommen könnte.

3.2.5 Groups

Gruppen hingegen haben, wie auch in einem System ohne Domänenverwaltung, den Sinn, User hinsichtlich der Rechteverwaltung gleich zu behandeln. Für die Rechte-Vergabe sind OUs nicht geeignet. Man baut in der Regel also die Organisationsstruktur mit den OUs auf und fügt der jeweiligen OU dann die notwendigen Gruppen hinzu.

Dabei ist zu beachten, dass im Active Directory unterschiedliche Gruppenarten existieren.

- Domain-Lokale Gruppen
- Globale Gruppen
- Universelle Gruppen

Domain-Lokale Gruppen sind nur in der Domäne sichtbar, in der sie auch resident sind. Sie stehen in keinerlei Verbindung mit den lokalen Gruppen, die auf einem Rechner selbst definiert sind. Diese Gruppen werden mit Ausnahme der lokalen Administratoren-Gruppe bei einem Beitritt des Computers zu einer Domäne sowieso obsolet. Globale als auch Universelle Gruppen sind in der gesamten Struktur, als auch in den Nachbar-Strukturen die über Trusts zu einem Forest zusammengefasst wurden, sichtbar. Die Gruppenmitgliedschaft von Globalen Gruppen wird aber nicht über die Domaingrenzen hinweg repliziert. Ist dies gefordert, so sind Universelle Gruppen das Mittel der Wahl, wobei bei häufigen Änderungen an der Gruppenstruktur ein erhöhter Replikations-Traffic bedacht werden sollte, der bei Globalen Gruppen durch die fehlende Replikation nicht auftritt.

Der Aufbau der Gruppenstruktur wird, wie in Abbildung 3.4 veranschaulicht, folgendermaßen empfohlen:

- Benutzer in Globale Gruppen zusammenfassen.
- Rechtevergabe nur an Domain-Lokale Gruppen.
- Globale Gruppen als Mitglieder der Domain-Lokalen Gruppen setzen.

Sollen Universelle Gruppen eingeführt werden, so werden diese Mitglieder in den Lokalen Gruppen und die Globalen Gruppen Mitglieder in den Universellen Gruppen.

Die Sichtbarkeit bzw. Schachtelungsmöglichkeit von verschiedenen Gruppen innerhalb der Domain, des Trees und des Forrests, wird in Abbildung 3.5 veranschaulicht. Lokale Gruppen können Benutzer, Lokale Gruppen, Globale Gruppen und Universelle Gruppen der selben Domain, Benutzer, Globale und Universelle Gruppen aus der Struktur und anderer Strukturen des Forrests als Mitglieder aufnehmen. Analog sind die beiden Diagramme für Globale und Universelle Gruppen zu interpretieren [3].

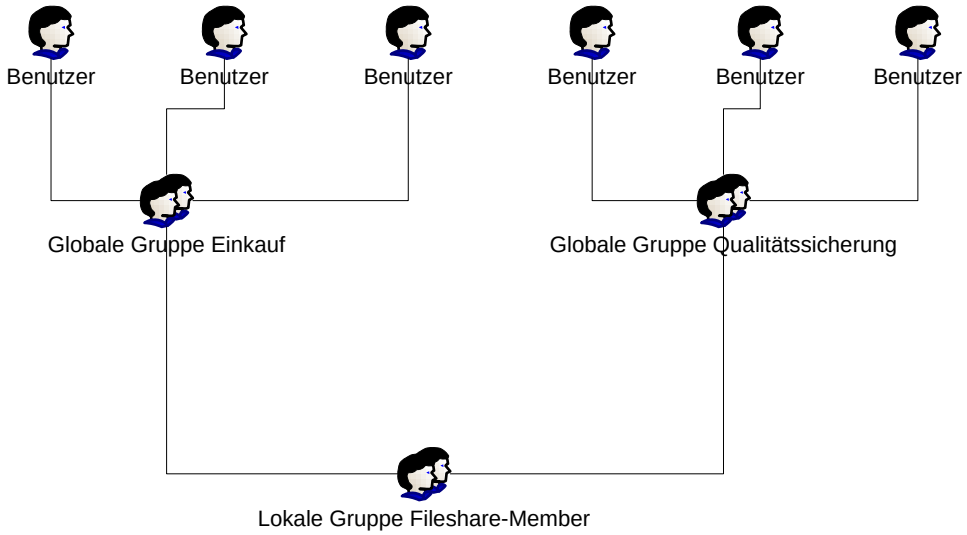


Abbildung 3.4: Gruppenmanagement

3.2.6 Schema

Das Schema ist wesentlicher Bestandteil des Active Directory. Es stellt die Datenbankstruktur, sprich die Attribute, die die jeweiligen AD-Objekte aufweisen, dar. Ein User hat beispielsweise einen Namen, eine Email Adresse, eine Telefonnummer und noch vieles mehr. Wie die Schemaausprägung letzen Endes aussieht, variiert von Unternehmen zu Unternehmen und obliegt der Konfiguration des Administrators. Wesentlich dabei ist, dass das Schema für die gesamte Struktur also auch domänenübergreifend (zum Beispiel für den gesamten Tree) einheitlich ist. Um dies sicherzustellen sind Änderungen am Schema nur auf einem DC – dem mit der Schema-Master Rolle – möglich und werden zu den anderen DCs repliziert.

3.2.7 Global Catalog

Das abschließend erwähnte aber doch sehr wesentliche Konzept im Active Directory ist der Global Catalog. Der Global Catalog stellt im Wesentlichen einen Indexdienst für die AD-Objekte zur Verfügung. Allen Objekten gleich ist, dass sie in der Domäne residieren, der sie zugehörig sind bzw. angelegt wurden. Manche von ihnen werden über die gesamte Struktur hinweg und somit zu allen DCs repliziert wie beispielsweise das Schema. Andere Objekte wie die Benutzerkonten sind lediglich innerhalb der zugehörigen Domäne auf den DCs vorhanden. Eine Replikation aller AD-Objekte zu allen Domänen wäre durch die quadratische Komplexität unwirtschaftlich. Man stelle sich nun folgendes Szenario vor: Ein User möchte die Dienste eines Netzwerkdruckers in Anspruch nehmen, der nicht in seiner Domäne liegt. Ein erster Ansatz wäre eine

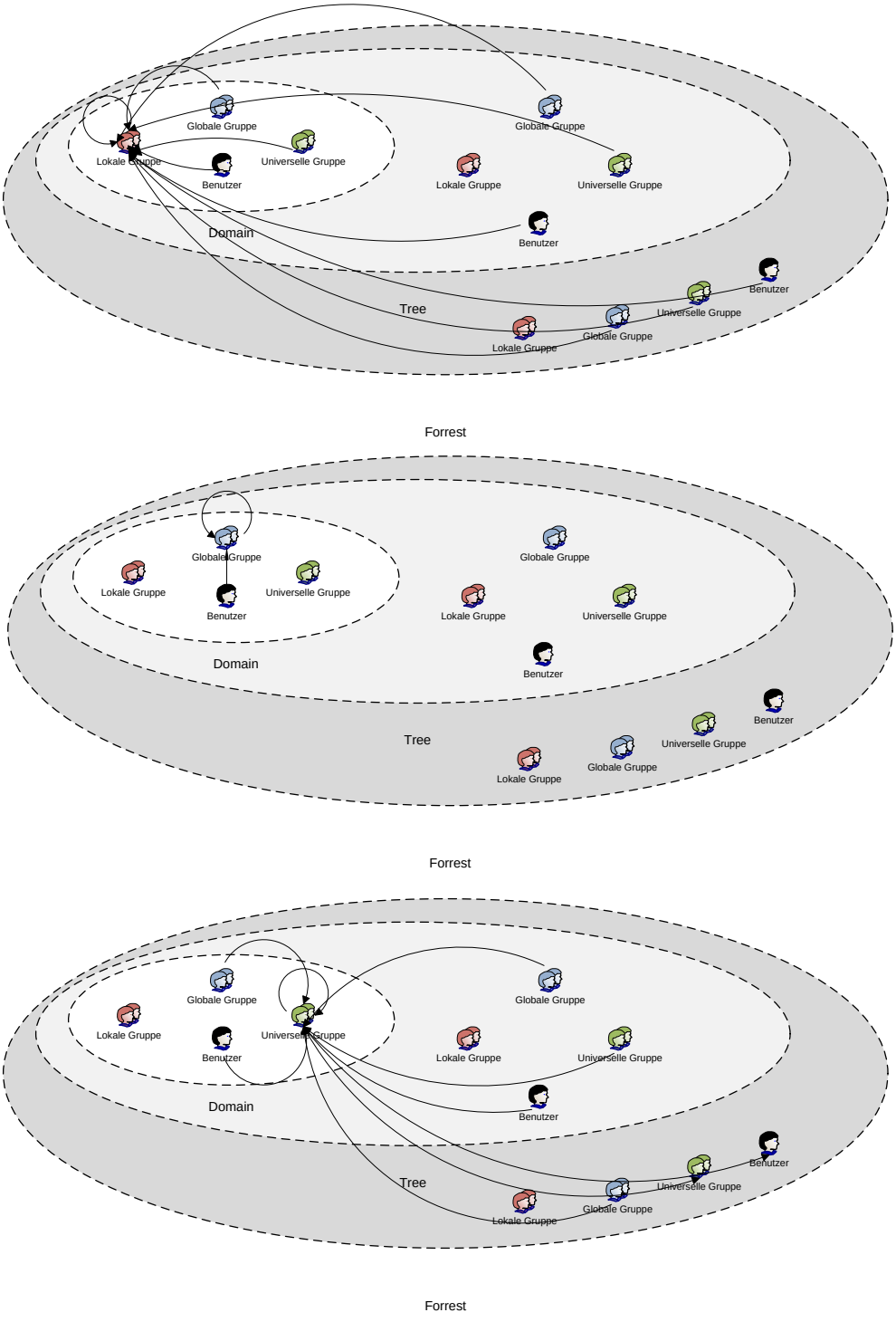


Abbildung 3.5: Gruppensichtbarkeit

Suche nach Netzwerkdruckern innerhalb der gesamten Struktur durchzuführen. Eine Entscheidung, die nicht nur lange dauern, sondern auch eine erhöhte Netzwerklast verursachen würde. Die Lösung ist ein zentraler Verzeichnisdienst, welcher Aufschluss über den Ort aller im AD enthaltenen Objekte gibt und gleichzeitig die wichtigsten Informationen über diese bereitstellt, der Global Catalog. Der GC speichert automatisch eine readonly Kopie von jedem AD-Objekt und eine Auswahl an Attributwerten zu jedem Objekt. Damit erfolgt eine ständige Replikation zwischen allen Domänen bzw. DCs und dem GC. Jeder DC ist berechtigt, die Rolle des GC zu übernehmen. Außerdem sollte aus Gründen der Ausfallsicherheit mehr als ein GC in einer Struktur, also einem Tree vorhanden sein.

Kapitel 4

RBACSystem für MS Windows 7 und Server 2008

4.1 Motivation

Microsoft Windows stellt mit dem Authorization Manager, der bereits in einer vorangegangenen Praktikumsarbeit [14] behandelt wurde, Entwickler die Möglichkeit zur Verfügung, Benutzer ihrer Applikationen alternative Zugriffskontrollsysteme abweichend von DAC anzubieten. Dabei dient der Authorization Manager als Datenbasis für das Zugriffskontrollsystem, die über die win32 Schnittstelle abgefragt und modifiziert werden oder über das Management Konsolen SnapIn „azman.msc "bearbeitet werden kann. Er bietet damit Applikationsentwickler die Möglichkeit, die in Windows bereits existierenden Gruppen und Benutzer für das gewünschte abweichende Zugriffsmodell zu verwenden, ohne dabei eine konkrete Vorgabe zu machen, wie dieses im Detail auszusehen hat. Die innerhalb des Authorization Manager definierten Permissions (dort Operation- und Task-Definitions genannt) haben ohne eine Applikation, welche den Sinn dieser Definitions interpretieren kann und auf Basis dessen konkrete OS-Operationen - wie beispielsweise einen Dateizugriff - durchführt, keinerlei Wirkung. Obwohl mit dem Authorization Manager durchaus andere Zugriffskontrollsysteme realisierbar wären, ist er durch die darin enthaltenen Konzepte für die Umsetzung von RBAC prädestiniert.

Um die Motivation dieser Masterarbeit verstehen zu können, ist es essentiell, dass die Authorisierungs-Modelle von Microsoft [5] – Impersonation Model und Trusted Subsystem Model – als auch die Grenzen von abweichenden Zugriffskontrollsystemen, die mit dem Authorization-Managers realisierbar sind, erörtert werden.

4.1.1 Impersonation Model

Wenn sich ein User am System anmeldet, so wird ein Access Token erzeugt. Jeder Prozess, der von dem User ausgeführt wird, erhält eine Kopie dieses Tokens. Greift der Prozess auf Objekte zu, die einer Zugriffskontrolle unterliegen, so wird dieser Token herangezogen, um zu verifizieren, ob der Zugriff des Users auf das Objekt auch tatsächlich erlaubt ist. Dies geschieht über den Abgleich des im Token enthaltenen SID und den eingetragenen SIDs in den ACLs der Objekte, wie in Kapitel 3 bereits beschrieben.

Das Impersonation Model ist das Standard Vorgehensmodell, um den Zugriff auf die Ressourcen des Betriebsystems zu regeln. User, die Applikationen verwenden, welche unter dem Impersonation Modell laufen, werden nach folgendem Konzept für den Ressourcenzugriff autorisiert: Die Applikation bezieht den Access Token des interagierenden Users. Sie kann damit wie der User selbst agieren und somit auf Objekte zugreifen. Die Prüfung der Legalität des Zugriffs erfolgt durch das Betriebsystem auf Basis der ACLs und dem Access Token. Diese Impersonalisierung kann durch den Aufruf von *ImpersonateLoggedOnUser* oder *ImpersonateNamedPipeClient* geschehen. *ImpersonateLoggedOnUser* benötigt als Argument den Access Token des Users, der mit der Funktion *LogonUser* bezogen werden kann. *ImpersonateNamedPipeClient* hingegen wird für Client/Server Anwendungen eingesetzt. Es erlaubt einem Server den zugreifenden Client zu impersonalisieren. Die Impersonalisierung dauert an, bis die Applikation die *RevertToSelf* Funktion ruft. Der Vorteil dieser Vorgehensweise ist eine effiziente Umsetzung von Autorisierungsmechanismen innerhalb der Applikation, da alle Zugangsprüfungen vom Referenzmonitor des Betriebsystems durchgeführt werden. Diesen Vorteil erkauft man sich allerdings mit einem erhöhten Verwaltungsaufwand für jeden einzelnen User in den NTFS Permissions. Ein großer Nachteil betrifft einen sicherheitstechnischen Aspekt. Falls die Applikation kompromittiert wurde, so kann der Umstand eintreten, dass der Angreifer durch die Impersonalisierungsfunktionen alle Rechte der interagierenden User erhält, weil er unter Verwendung ihres Kontos agieren kann. Wenn also ein Administrator mit der Applikation interagiert, so erhält der Angreifer über die Schwachstelle Administratorenrechte, die er innerhalb der Applikation nutzen kann.

4.1.2 Trusted Subsystem Model

Das Trusted Subsystem Model hingegen führt die detaillierte Zugangsprüfung innerhalb der Applikation durch. Die Zugriffskontrollpolitik ist in der Applikation umgesetzt oder liegt auf einer Datenbasis, auf welche die Applikation zugreift. Auf Basis dessen entscheidet die Applikation, ob der Zugriff des gerade interagierenden Users erlaubt ist, realisiert das Referenzmonitorkonzept also selbst, und greift – im Falle einer positiven Zugangsprüfung – schließlich auf die Ressource zu. Der Referenzmonitor des Betriebsys-

tems muss nun prüfen, ob die Applikation die jeweiligen Rechte besitzt, die Ressource zu verwenden. Die Applikation muss also mit einem Benutzerkonto in den ACLs für die Rechte eingetragen sein, die sie ihren Anwendern im Maximalfall gewährt. Der Sicherheitsvorteil hier ist, dass ein Angreifer eine mögliche Schwachstelle in der Applikation nur soweit ausnutzen kann, wie die Berechtigung der Applikation reicht. Wurde die Rechtevergabe für die Applikation nach dem Principal of Least Privilege durchgeführt und gegen eine Elevation of Privilege abgesichert, so ergibt sich im Gegensatz zum Impersonation Model eine enorme Sicherheitsverbesserung.

Für das Trusted Subsystem Model wurde der Authorization Manager von Microsoft entworfen. Er stellt hierbei die Datenbasis dar, in der die Berechtigungsstruktur verwaltet werden kann. Diese Datenbasis kann über ein GUI oder auf programmatischem Wege unter Verwendung des Win32 API angesprochen werden. Der Referenzmonitor innerhalb der Applikation stützt seine Entscheidungen auf die im Authorization Manager definierte Zugriffspolititk, welche entweder auf einem XML, MS SQL oder Active Directory Policystore liegen kann. Auf diese Weise können für Applikationen vom DAC abweichende Zugriffskontrollsysteme realisiert werden. Damit ist es möglich ein RBAC System oder auch ein MAC System für Windows Applikationen umzusetzen. Der darunterliegende Zugriff der Applikation bleibt jedoch auch beim Trusted Subsystem Model immer auf das DAC-System von Windows angewiesen, da zumindest der Applikation-Account in den ACLs der Objekte mit den notwendigen Rechten eingetragen sein muss.

Anwender, die Windows interne Programme wie den Explorer nutzen oder in einem Netzwerk auf File-Shares zugreifen, müssen dies immer unter der Verwendung des DAC Modells tun. Die Rechteverwaltung von NTFS ist strikt nach diesem Prinzip aufgebaut. Zwar existiert in Windows das Gruppenkonzept, dass eine Ähnlichkeit zu Rollen hinsichtlich der User Mitgliedschaft und Rechtezuweisung aufweist, allerdings kann man aufgrund dieser statischen und sehr minimalistischen Ausprägung noch nicht von einem rollenbasierten Konzept im Sinne des ANSI/INCITS Standards sprechen. Ziel ist es also, das Zugriffskontrollsystem von Windows so zu modifizieren, dass aus einem DAC ein RBAC System wird, bzw. diese beiden koexistieren können.

4.2 Lösungsansätze

Vor Beginn der Lösungsimplementierung wurden verschiedene Szenarien durchgespielt und Varianten durchdacht, wie ein RBAC-System mit größtmöglicher Effektivität bei gleichzeitiger Effizienz umgesetzt werden könnte. Besonderes Augenmerk lag dabei auf den in Windows bereits vorhandenen Konzepten hinsichtlich ihrer Eignung, sie in ein derartiges System integrieren zu können.

4.2.1 Gruppe vs. Rolle

Da auch schon im Paper von Sandhu et al. [13] Ähnlichkeiten zwischen Gruppen und Rollen diskutiert werden, lag die Überlegung nahe, das RBAC System auf Basis von Windows Gruppen zu realisieren. Eine Gruppe in Windows wäre also einer Rolle gleichzusetzen und würde auch als eine solche behandelt werden. Jedoch fehlt in Windows hierfür ein zentrales Konzept: Die Dynamik. Sind erst einmal alle User den Gruppen, ergo Rollen, zugewiesen, so ist diese Zuweisung statisch und kann nur durch den Administrator verändert werden. Eine Gruppe kann in Windows also nicht wie eine Rolle aktiviert werden – ein User ist ihr entweder zugeordnet oder nicht. Im Falle der Gruppen-Zuordnung erhält er auch alle damit verbundenen Rechte. Somit ist ein Benutzer in Windows immer mit seinem vollen Berechtigungs-Repertoire ausgestattet, kann also keine Abstufungen vornehmen. Die Lösung hierfür könnte ähnlich aussehen, wie sie bereits in der Praktikumsarbeit mit dem Authorization Manager umgesetzt wurde, denn auch dort existiert kein integriertes Session-Konzept. Die Lösung erfolgte dort über Kopiervorgänge innerhalb des Authorization Managers von statischen zu dynamischen Rollen-Definitionen. Letztere wurden dann für die Zugangsprüfung herangezogen. Es müsste also auch hier eine dynamische Komponente, etwa ein Windows Dienst, dafür sorgen, dass ein Client seine Rollenmitgliedschaften ändern kann. So könnte es beispielsweise statische und dynamische Gruppen geben. Die statische Gruppe ist lediglich ein Container, der keinerlei Rechte besitzt, jedoch die statische User-Gruppen-Zuweisung repräsentiert. Der Administrator weist also User immer den statischen Rollen zu. Der User kann dann den Dienst zu Laufzeit veranlassen, dass er einer dynamischen Gruppe zugewiesen wird, welche die zur Rolle zugehörigen Rechte besitzt, wenn er bereits vorab Mitglied der zugehörigen statischen Rolle ist.

Nicht nur dass der Administrator aufpassen muss, nicht in die Session einzugreifen indem er eine falsche User Zuweisung zu den dynamischen Gruppen tätigt, ein User ist somit in der Lage – wenn auch über einen Dienst – selbst zu administrieren. Dies stellt per se eine empfindliche Schachstelle dar, wenn man Benutzern ohne Administratoren-Rechte die Möglichkeit gibt, ihre Gruppenmitgliedschaften zu ändern und eröffnet unter Umständen die Möglichkeit einer Elevation of Privilege.

4.2.2 OU vs. Rolle

Auch drängt sich gewissermaßen eine Ähnlichkeit zwischen Organisational Units und Rollen auf. Hierbei wäre auf den ersten Blick zumindest eine grundlegende Dynamik vorhanden, da man mit Policies arbeiten könnte, die bei der User Anmeldung geladen werden. Da jeder User Mitglied in nur einer OU sein kann, scheidet jedoch auch die-

se Vorgehensweise aus, denn bei nur einer möglichen OU, sprich Rolle, erübrigt sich jegliches Session Konzept.

4.2.3 Konzept der Lösung

Für beide eben genannte Konzepte gilt, dass sie auf RBAC beschränkt sind. Gruppen bzw. OUs mögen zwar Rollen sehr ähnlich sein, für andere Zugriffskontrollsysteme muss das Konzept „Gruppe“ bzw. „OU“ nicht zwangsläufig eine Entsprechung finden. Damit würde man sich mit diesen Implementierungen die Flexibilität hinsichtlich der Austauschbarkeit des Zugriffkontrollsystems verwehren. Auch wäre man bezüglich der Rechte-Adressierung auf das Objekt an sich beschränkt. Gerade für RBAC ist die Flexibilität der Rechte-Adressierung von hohem Interesse. Während Permissions in NTFS immer einem Objekt explizit zugeordnet werden, könnten sie sich in RBAC auch auf Attribute eines Objektes stützen, sprich dem Objekt implizit zugeordnet werden. Dies kann beispielsweise dadurch geschehen, dass das Objekt, auf welchem ein Zugriff erlaubt wird einen gewissen Attributwert aufweist, z.B. der Ort im Dateisystem, den Zeitstempel des letzen Zugriffes, einen Wert innerhalb der File-Tags, wie den Titel oder Autor. Eine Permission würde also beispielsweise lauten „Lesezugriff auf alle Objekte, die nach dem 01.02.2010 erstellt wurden und als Autor Max Müller eingetragen haben“. Auf diese Weise könnten Wildcards für den Zugriff erstellt werden, die die Verwaltung vereinfachen.

Die Lösung stellt also ein System dar, das einen Referenzmonitor implementiert, der über dem gegenwärtigen Zugriffskontrollsystem steht und die Möglichkeit bietet, darauf aufbauend verschiedenste Zugriffskontrollsysteme zu realisieren. Um diesen Referenzmonitor zu implementieren, wurde daher nach einem Konzept - ähnlich einem „Global Hook“ für das Dateisystem – gesucht, dass im Falle eines Zugriffes die Möglichkeit bietet, diesen zu prüfen und in weiterer Folge zu unterbinden oder gewähren zu lassen. Ein Global Hook in Windows ist ein Mechanismus, der systemweite Ereignisse überwacht und einem registierten Listener die Möglichkeit gibt, auf dieses Ereignis zu reagieren. Dieses Observer Pattern wird auch oft von Malware wie beispielsweise Keylogger missbraucht. Die Lösung für den RM war schließlich nicht der Windows Global Hook an sich sondern ein Konzept namens „Filesystem Filter Driver“, welches an späterer Stelle detailliert erläutert wird. Das nachfolgend beschriebene System, *RBACSystem* versucht ein RBAC Core Set nach dem ANSI/INCITS Standard [8] bzw. RBAC0 Modell nach Sandhu et al. [13] unter Windows 7 und Server 2008 umzusetzen.

4.3 RBACSystem

RBACSystem setzt sich aus dem *RBACSystem*-Core-Package, welches aus drei Komponenten besteht, und drei weiteren Zusatzkomponenten zusammen.

- *RBACSystem*-Core-Package
  - *RBACSystemDriver*, die Windows-Kernel Komponenten, welche den Zugriff auf das File-System modifiziert.
  - *RBACSystemMonitor*, ein .NET Service, der als Referenzmonitor agiert.
  - *RBACSystemImed*, eine win32 dynamic link library, welche den Mediator zwischen dem Kernel Mode und dem User Mode darstellt.
- *RBACSystemDatabase*, eine MSSQL Datenbank, in der die Berechtigungsstruktur verzeichnet ist.
- *RBACSystemServiceCtrl+Mgmt*, einer .NET GUI, welche für die Service Steuerung und das Management der Rechtestruktur dient. Diese Applikation läuft auf dem Server, auf welchem das *RBACSystem*-Core-Package installiert ist.
- *RBACSystemClientCtrl*, einer .NET GUI, welche für die Sessionverwaltung dient. Diese Applikation läuft auf den Clientcomputern, welche mit *RBACSystem* interagieren.

Die Abbildung 4.1 zeigt die Interaktion dieser Komponenten.

4.3.1 RBACSystemDriver

Da Windows ein DAC-System zugrunde liegt, welches dem Dateisystem NTFS zugehörig ist, wurde nach Möglichkeiten gesucht, das Verhalten des Dateisystems dermaßen zu beeinflussen, dass es ein RBAC0 Verhalten nach Sandhu et al. [13] aufweist. Windows bietet die Möglichkeit mit IFS (Installable File-System Drivers), welche Bestandteil des WDK (Windows Driver Kit) sind, einen File-System Filter Driver zu implementieren, der das Verhalten des Dateisystems modifiziert. Derartige Treiber werden in der Regel auch von Antiviren Software, Verschlüsselungs- und Replikationsprogrammen wie RAID Treiber und dergleichen verwendet.

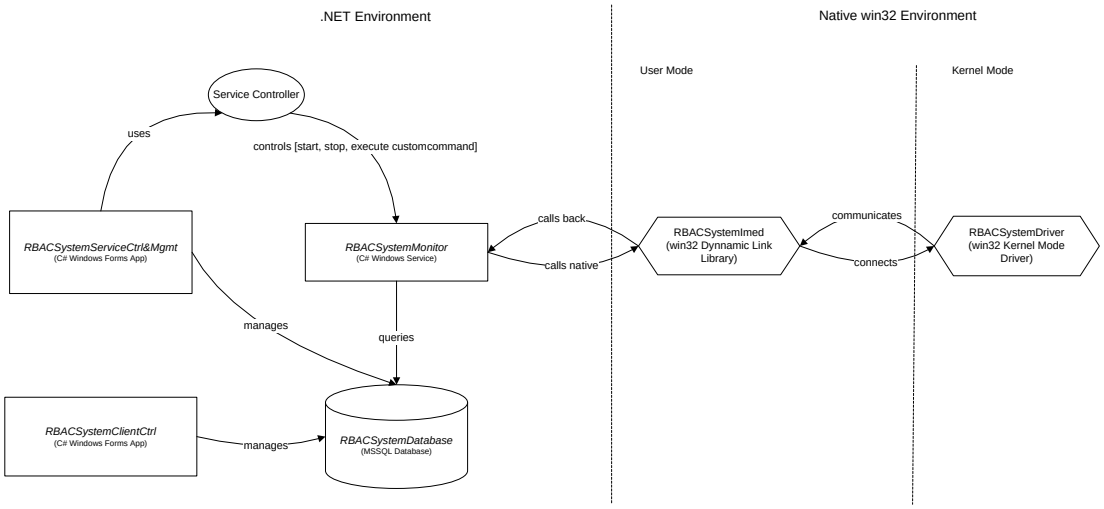


Abbildung 4.1: Architektur RBACSystem

Der File-System Filter Driver stellt dabei eine NT-Kernel Komponente dar. Diese Komponente kann Zugriffe auf ein Dateisystem loggen, beobachten, modifizieren oder verhindern. Der File-System Filter Driver platziert sich im I/O Stack zwischen dem I/O Manager und dem File-System Driver, Abbildung 4.2. Zwischen diesen beiden können beliebig viele Filter installiert werden. Der I/O Strom wird dann von der Quelle, dem I/O Manager bis zum Ziel, dem File-System Driver gefiltert und kann aufgrund der im Filter implementierten Routinen modifiziert oder beobachtet werden.

Prinzipiell werden drei Arten von Treibern für das Dateisystem unterschieden

- File-System Driver
- File-System Filter Driver
- File-System Minifilter Driver

Der File-System Driver ist der Treiber, der das originale Verhalten des Dateisystems umsetzt, im File-System I/O Stack die unterste Ebene belegt und den Request an den Storage Driver Stack weiterleitet, der den Zugriff auf die Hardware verarbeitet. Er ist für die gängigen Dateisysteme wie FAT, NTFS, etc. in Windows vorhanden und deshalb besteht in der Regel keine Notwendigkeit, diesen zu implementieren. Falls der File-System Driver nicht die gewünschte Funktionalität aufweist, so muss deshalb nicht der Treiber neu geschrieben werden sondern kann seine Funktionalität durch einen File-System Filter Driver modifiziert werden, der zwischen den File-System Driver und dem I/O Manager geschalten wird. Da diese Filter sehr schwer zu implementieren sind, erleichtert Microsoft Herstellern von Legacy-Filtern den Entwicklungsprozess und bietet

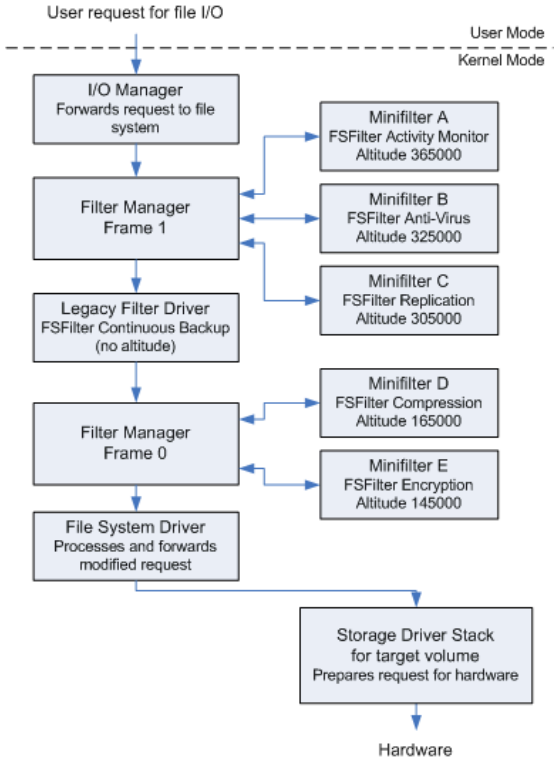


Abbildung 4.2: I/O Stack, Quelle: MSDN Library [2]

die Möglichkeit, das File-System Minifilter Driver Modell zu nutzen. Diese verwenden einen Filter-Manager, welcher Bestandteil von Windows ist, aber nur dann aktiv wird, wenn ein Minifilter Treiber geladen wird. Der Filter-Manager ist eine Kernel-Driver Komponente die dem Legacy File-System Filter Model konform ist und das Verhalten eines File-System Filter Drivers simuliert. Er ist selbst also im Wesentlichen ein File-System Filter Driver, der sich in den File-System Stack für ein bestimmtes Target-Volume integriert, jedoch Minifilter die Möglichkeit bietet, sich zu registrieren. Er leitet dann den I/O Strom auf seine registrierten Minifilter um. Diese behandeln den I/O Strom in ihren Routinen und leiten ihn dann wieder zurück an den Filter Manager. Damit wirken die Minifilter indirekt auf das Verhalten des Dateisystems, da sie nicht direkt Bestandteil des File-System Stacks sind. Sie sind jedoch wesentlich einfacher zu implementieren und robuster als File-System Filter Driver, die den I/O Strom im Stack direkt beeinflussen.

4.3.1.1 I/O Request-Typen

Es existieren zwei verschiedene Typen von I/O Requests, welche von einem File-System Filter Driver bzw. File-System Minifilter Driver gefiltert werden können. I/O Request Packets (IRP) und Fast I/O Requests. IRPs sind gewissermaßen der Standard-Mechanismus, um I/O Operationen durchzuführen. Fast I/O wurde speziell dafür ent-

worfen, um schnellen synchronen Zugriff auf Files im System-Cache zu ermöglichen. Bei dieser Zugriffsart wird das Dateisystem außer Acht gelassen und direkt aus dem System-Cache gelesen. Storage Drivers unterstützen kein Fast I/O. Falls ein File mit Fast I/O gelesen werden soll, so muss es sich vollständig im Cache befinden, ist dies nicht der Fall, so treten Seitenfehler auf und ein IRP wird generiert um die Seiten nachzuladen.

IRPs werden von den Driver Dispatch Routinen behandelt, Fast I/O Requests von den Driver Callback Routinen. Wenn ein Filter Driver initalisiert wird, registiert seine DriverEntry Routine die Dispatch Routinen und die Fast I/O Callback Routinen.

File-System Filter müssen IRPs unterstützen, jedoch nicht zwangsläufig Fast I/O. Falls Fast I/O nicht unterstützt wird, so muss jedoch zumindest die Callback-Routine den Wert FALSE liefern. Wenn der I/O Manager die Anforderung für einen synchronen File I/O erhält, so ist die erste Wahl des I/O Managers Fast I/O. Liefert diese Routine eines Filters FALSE, so kreiert der I/O Manager statt dessen IRPs und sendet sie erneut.

Minifilter können IRP und Fast I/O Requests verarbeiten bzw. muss bei der Implementierung nicht explizit zwischen den beiden Varianten unterschieden werden. Der Filtermanager übernimmt die Verarbeitung beider Requests. In den Minifiltern müssen lediglich die PreOperation- und PostOperation-Callback-Routinen für die jeweiligen Operationen definiert werden. Wie auch die File-System Filter in der Reihenfolge aufgerufen werden, wie sie in den File-System Stack geschalten wurden, so werden auch die Minifilter in der Reihenfolge aufgerufen, wie sie über das Attribut „Altitude“ – welches die Position im Stack angibt – eingereiht wurden. Ein bedachter Umgang mit der Filterreihenfolge ist entscheidend für den Erfolg. Antivirus Filter sollten sinngemäß vor Replikations-Filter geschalten werden, da sich so verseuchte Files gar nicht erst replizieren können. Das RBAC Filter kann als letzte Instanz im Stack angesiedelt werden. Bei einem I/O-Request ruft dann der Filtermanager die Callback-Routinen seiner registrierten Minifiter auf, falls diese eine entsprechende Callback-Routine für den Request-Typ registriert haben. Er tut dies in der Reihenfolge, wie sie bei ihm über das Attribut Altitude registiert wurden. Zuerst erfolgt ein sequentieller Aufruf aller Preoperation Callback Routinen, vom höchsten zum niedrigsten Minifilter, wenn der Filtermanager den „Request for Completion“ erhält, so werden in umgekehrter Reihenfolge die Postoperation Callback Routinen beginnend vom untersten bis zum obersten Minifilter aufgerufen.

4.3.1.2 Treiber-Aufbau und Code-Examples

Haupteinstiegspunkt in den Filesystem Minifilter Driver ist die Funktion *DriverEntry*

```
NTSTATUS
DriverEntry (
    __in PDRIVER_OBJECT DriverObject,
    __in PUNICODE_STRING RegistryPath
    )
{
...
}
```

In dieser Funktion müssen 3 essentielle Schritte getätigt werden. Der erste Schritt betrifft die Registrierung des Filters beim Filtermanager über die Funktion *FltRegisterFilter*.

```
FltRegisterFilter( DriverObject,
                   &FilterRegistration,
                   &globalFilterHandle );
```

Von den 3 Übergabeparametern ist der Pointer auf die *FilterRegistraton* Struktur (*FLT_REGISTRATION*) für die Registrierung am bedeutsamten. In dieser befinden sich sämtliche Attribute, die den Filter beschreiben.

```
CONST FLT_REGISTRATION FilterRegistration = {

    sizeof( FLT_REGISTRATION ),         //  Size
    FLT_REGISTRATION_VERSION,           //  Version
    0,                                  //  Flags

    NULL,                               //  Context
    Callbacks,                          //  Operation callback Array

    Unload,                            //  MiniFilter Unload Routine

    InstanceSetup,                     //  Instance Setup Routine
    InstanceQueryTeardown,             //  Instance Query Teardown
    InstanceTeardownStart,             //  Instance Teardown Start
    InstanceTeardownComplete,          //  Instance Teardown Complete

    NULL,                               //  Generate File Name
    NULL,                               //  Generate Destination File Name
```

```
    NULL                                    //  Normalize Name Component

};
```

Besonders wichtig in diesem Zusammenhang sind das Array *Callbacks* und der Funktionspointer *InstanceSetup*.

Die Callbacks stellen ein Array aus *FLT_OPERATION_REGISTRATION* Strukturen dar. Die Strukturen beinhalten den IRP Funktionscode (Auflistung aller Funktions-Codes [1]), Flags und die Pre- und PostOperation Callback Routinen. Die Initialisierung des Arrays könnte folgendermaßen aussehen:

```
CONST FLT_OPERATION_REGISTRATION Callbacks[] = {
    { IRP_MJ_CREATE,
      0,
      PreOperationCreate,
      PostOperationCreate },

    { IRP_MJ_READ,
      0,
      PreOperationRead,
      PostOperationRead },
...
}
```

InstanceSetup ist ein Pointer vom Typ *PFLT_INSTANCE_SETUP_CALLBACK* und muss auf eine Funktion zeigen, die *NTSTATUS* als Rückgabewert liefert. Diese Funktion wird immer dann aufgerufen, wenn eine neue Instanz auf einem Volume erzeugt wird. Damit bietet diese Funktion die Chance zu entscheiden ob der Filter auf ein Volume wirken soll oder nicht. Wenn diese Funktion in *FilterRegistration* nicht angegeben wird, also *NULL* übergeben wird, so wird der Filter automatisch auf alle Volumes angewandt.

Der zweite Schritt in der DriverEntry-Funktion ist der Aufruf der Funktion

```
FltStartFiltering( globalFilterHandle );
```

Damit wird der Filter gestartet.

[1] http://msdn.microsoft.com/en-us/library/ms806157.aspx

Als dritter und letzter Schritt muss ein *NTSTATUS* retouniert werden, der über den Status des Filters Auskunft gibt. Dabei kann der Rückgabewert der Funktion *FltStartFiltering* einfach weitergereicht werden. Neben den verschiedenen Routinen wie *Unload*, *InstanceQueryTeardown* usw. welche in der Regel Standardimplementierungen folgen, müssen im Folgenden die Pre- und Postoperation Callback Routinen definiert werden.

```
FLT_PREOP_CALLBACK_STATUS
PreOperationRead (
    __inout PFLT_CALLBACK_DATA Data,
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __deref_out_opt PVOID *CompletionContext
    )
{
...
}
FLT_PREOP_CALLBACK_STATUS
PostOperationRead (
    __inout PFLT_CALLBACK_DATA Data,
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __deref_out_opt PVOID *CompletionContext
    )
{
...
}
```

Diese Routinen werden vom Filtermanager aufgerufen, wenn ein IRP vorliegt für welches sie registiert wurden. Im Fall von *IRP_MJ_READ* wären dies die Funktionen *PreOperationRead* und *PostOperationRead*, welche zuvor im Callback-Array für genau diesen Request eingetragen wurden. Beim Fluss des I/O Stroms vom I/O Manager zum File-System Driver wird dieser vom zwischengeschalteten Filtermanager an die Minifilter umgeleitet. Hier kann nun der Fluss des I/O Stroms beeinflusst werden. Wie bereits vorher erwähnt, werden beim I/O Fluss vom I/O Manager zum File-System Driver die PreOperation Callback Routinen aufgerufen. Der Pointer Data vom Typ *PFLT_CALLBACK_DATA* zeigt auf eine Struktur die neben Flags und dem Thread, auf dem der I/O Request basiert, alle Informationen zum I/O Request am Pointer *Iopb* vom Typ *PFLT_IO_PARAMETER_BLOCK* enthält.

Diese Struktur kann verwendet werden, um eine Vorfilterung der Requests zu tätigen, auf die reagiert werden soll, indem die Eigenschaftswerte des I/O-Request analysiert werden. Damit kann eine genauere Unterscheidung getroffen werden, ob ein Request

behandelt werden soll oder nicht, denn eine Callback-Routine wird zwar für einen Major Function Code registriert, die Feinunterscheidung erfolgt jedoch über die Minor Function Codes sowie über zahlreiche Flags.

Dann erfolgt die Prüfung, ob die Operation, welche durch den I/O-Request beschrieben wird fortgesetzt werden kann oder unterbunden werden muss. Falls der I/O-Request fortgesetzt werden kann, so muss die PreOperationCallback Funktion den Wert *FLT_PREOP_SUCCESS_WITH_CALLBACK* retournieren. Dies ist für den RBAC-System Minifilter Driver dann der Fall, wenn die betreffende Operation aufgrund des Referenzmonitor-Ergebnisses dem zugreifenden Subjekt erlaubt werden kann.

Für den MiniFilter *RBACSystemDriver* muss jedoch auf Basis des Referenzmonitor Ergebnisses die Fortsetzung des I/O Stroms auch unterbunden werden können. Dies erfolgt über das Setzen des *IOStatus.Status* Feldes auf den finalen Status der Operation in der Callback Data-Struktur. In diesem Fall lautet der Status: *STATUS_ACCESS_DENIED*. Außerdem muss der Wert *FLT_PREOP_COMPLETE* retourniert werden. Damit reagiert dann der Filtermanager folgendermaßen: Er sendet die Operation nicht an weiter Minifilter unterhalb des unterbrechenden Filters, also auch nicht an Legacy Filter (Standard File System Filter Driver im Stack) und das Dateisystem. Er ruft weiters die PostOperation Callback Routinen der Minifilter oberhalb des unterbrechenden Filters auf, jedoch nicht die PostOperation Callback Routine des unterbrechenden Filters. Vereinfacht gesagt werden keinerlei Treiber mehr involviert, die unterhalb des unterbrechenden Minifilter Drivers liegen und lediglich die Operation mit einem negativen Bescheid komplettiert.

Eine Auflistung der möglichen Varianten wie eine Callback-Routine abgeschlossen werden kann, kann unter der MSDN Library[2] eingesehen werden.

4.3.1.3 Behandlung eines I/O Requests

Die *PFLT_CALLBACK_DATA* Datenstruktur wird bei jedem Aufruf einer Callback Routine als *IN/OUT* Parameter übergeben. Diese Struktur besitzt folgenden Aufbau:

```
typedef struct _FLT_CALLBACK_DATA {
  FLT_CALLBACK_DATA_FLAGS  Flags;
  PETHREAD CONST  Thread;
  PFLT_IO_PARAMETER_BLOCK CONST  Iopb;
  IO_STATUS_BLOCK  IoStatus;
```

[2] http://msdn.microsoft.com/en-us/library/ms793697.aspx

```
  struct _FLT_TAG_DATA_BUFFER  *TagData;
  union {
    struct {
      LIST_ENTRY  QueueLinks;
      PVOID  QueueContext[2];
    };
    PVOID  FilterContext[4];
  };
  KPROCESSOR_MODE  RequestorMode;
} FLT_CALLBACK_DATA, *PFLT_CALLBACK_DATA;
```

Von besonderem Interesse ist dabei Variable *Iopb* vom Typ *PFLT_IO_PARAMETER_BLOCK*

```
typedef struct _FLT_IO_PARAMETER_BLOCK {
  ULONG  IrpFlags;
  UCHAR  MajorFunction;
  UCHAR  MinorFunction;
  UCHAR  OperationFlags;
  UCHAR  Reserved;
  PFILE_OBJECT  TargetFileObject;
  PFLT_INSTANCE  TargetInstance;
  FLT_PARAMETERS  Parameters;
} FLT_IO_PARAMETER_BLOCK, *PFLT_IO_PARAMETER_BLOCK;
```

In dieser Struktur befinden sich die Felder *MajorFunction*, *MinorFunction* und *Parameters*. Über die *MajorFunction* und *MinorFunction* kann die Natur des I/O Requests ermittelt werden. Dies geschieht am übersichtlichsten in einer selbst definierten Funktion. Im Falle von *RBACSystemDriver* übernimmt dies die Funktion *RsdMonitorIoRequest(_in PFLT_CALLBACK_DATA Data)*, welche die Major und Minor Function des I/O Requests examiniert und entscheidet, ob die Operation von Interesse für den Filterprozess ist. Dies ist immer genau dann der Fall, wenn die MajorFunction den Wert *IRP_MJ_CREATE* besitzt. Der I/O Manager sendet *IRP_MJ_CREATE* immer dann, wenn ein neues File oder Directory erzeugt wird oder – anders als man vom Namen zunächst vermuten würde – ein bereits vorhandenes File oder Directory geöffnet wird. In der Regel wird der besagte I/O Request von einer User Mode Applikation ausgelöst, welche die Win32 Operation *CreateFile* ausführt. Er kann aber auch durch eine Kernel Komponente über die Funktionen *IoCreateFile*, *IoCreateFileSpecifyDeviceObjectHint*, *ZwCreateFile* oder *ZwOpenFile* ausgelöst werden. Egal von welcher Quelle auch immer, wichtig dabei ist, dass kein Weg an *IRP_MJ_CREAT*E vorbei führt, wenn das Dateisys-

tem in lesender oder schreibender Weise betroffen ist. Damit stellt IRP_MJ_CREATE das unmittelbare Signal zum handeln dar[3].

Wenn also ein derartiger I/O Request auftaucht, so sind drei Informationen zu ermitteln:

- Wer hat den Request verursacht? (SID des zugreifenden Subjektes)
- Auf welches Objekt zielt die Operation ab? (hier über den Ort im Dateisystem identifiziert)
- Welcher Art ist die Operation? (read, write, execute, ...)

Sind diese Informationen zusammengetragen, so können sie an die User-Mode Komponente übertragen und diese damit beauftragt werden, auf Basis der übermittelten Informationen zu entscheiden ob der Request akzeptabel ist oder nicht.

4.3.1.3.1 Requestor ermitteln

Wie im Kapitel über die Windows Sicherheit 3 bereits erläutert, besitzt jedes Subjekt und jede Gruppe im System einen Security Identifier, den sogenannten SID. Der SID muss nun aus dem I/O Request gelesen und an die User Mode Applikation geleitet werden.

Über das Feld *Parameters.Create.SecurityContext->AccessState-> SubjectSecurityContext.PrimaryToken* kann der Primäre Token des I/O Requests bezogen werden. *SECURITY_SUBJECT_CONTEXT* hat folgenden Aufbau:

```
typedef struct _SECURITY_SUBJECT_CONTEXT {
  PACCESS_TOKEN  ClientToken;
  SECURITY_IMPERSONATION_LEVEL  ImpersonationLevel;
  PACCESS_TOKEN  PrimaryToken;
  PVOID  ProcessAuditId;
} SECURITY_SUBJECT_CONTEXT, *PSECURITY_SUBJECT_CONTEXT;
```

Es beschreibt, wie auch der Name schon erahnen lässt, den Sicherheits Kontext, der für die Zugriffs-Validierung und das Auditing eines I/O Requests verwendet wird. Die für den *RBACSystemDriver* erwähnenswerten Felder sind die Access Token. Ein Access

[3] http://msdn.microsoft.com/en-us/library/ms795806.aspx

Token ist ein System Objekt, das den Sicherheits-Kontext eines Prozesses bzw. Threads beschreibt. Die Information in einem Token beinhaltet die Identität und die Privilegien eines User Accounts, welche mit dem Prozess bzw. den Threads verbunden sind. Wenn sich ein Benutzer am System anmeldet, so überprüft dieses das Passwort, authentifiziert damit den User. Dabei erzeugt es einen Access Token. Jeder Prozess der vom User nun ausgeführt wird, erhält eine Kopie dieses Access Tokens. Das System verwendet den Access Token, um den zu einem Prozess zugehörigen User zu ermitteln, falls ein darin enthaltener Thread sicherheitskritische Operationen wie beispielsweise den Zugriff auf System-Objekte durchführt[4].

Wie in der *SECURITY_SUBJECT_CONTEXT* Struktur ersichtlich, existieren dort zwei Variablen für Access Token. Der primäre Access Token (*PrimaryToken*) wird typischerweise nur vom Windows Kernel generiert. Er spiegelt, wie soeben erwähnt, den User wieder, unter dem der Prozess läuft. Der Impersonation Token (*ClientToken*) hingegen ist ein Token der in der Regel einen anderen User beschreibt. Damit ist eine Applikation in der Lage, Operationen im Auftrag eines anderen Users durchzuführen. Eine Server Applikation wird also in der Regel auf einem System mit einem bestimmten Konto ausgeführt. Wenn der Server selbst auf die Ressourcen zugreift, dann wird natürlich immer der Token der Applikation für die Zugangs-Validierung und das Auditing herangezogen. Diese Vorgehensweise nennt sich Trusted Subsystem Model. Die Applikation kann aber auch einen User impersonalisieren, also wie ein User auftreten und entsprechend agieren. Die Funktionen einen Client zu impersonalisieren sind vielfältig. Wenn ein Prozess einen User also impersonalisiert hat, so befindet sich der Access Token in der Variable *ClientToken* ansonsten ist der Wert *NULL*.

Der Treiber versucht also zuerst die Variable *ClientToken* zu verwenden ist sie nicht vorhanden, so wird die Variable *PrimaryToken* verwendet. Somit sind sowohl Applikationen, die dem Trusted-Subsystem-Model als auch solche, die dem Impersonation-Model konform sind, mit *RBACSystem* kompatibel.

```
NTSTATUS
RsdGetRequestor(__in PFLT_CALLBACK_DATA Data)
{
NTSTATUS status;
PACCESS_TOKEN  clientToken;
PTOKEN_USER tokenInfo = NULL;

if(Data->Iopb->Parameters.Create.SecurityContext->AccessState->
SubjectSecurityContext.ClientToken != NULL)
{
```

[4] http://msdn.microsoft.com/en-us/library/aa374909(VS.85).aspx

```
clientToken = Data->Iopb->Parameters.Create.SecurityContext->AccessState->
SubjectSecurityContext.ClientToken;
}else
{
clientToken = Data->Iopb->Parameters.Create.SecurityContext->
AccessState->SubjectSecurityContext.PrimaryToken;
}

status = SeQueryInformationToken(clientToken,
TokenUser,
(PVOID*)&tokenInfo);

if(NT_SUCCESS(status))
{
status = RtlConvertSidToUnicodeString(&gSidUniString,
tokenInfo->User.Sid,
TRUE);

if(NT_SUCCESS(status))
{
status = RtlStringCbCopyW(gMessage->Sid, SID_MAX_LENGTH*sizeof(wchar_t),
gSidUniString.Buffer);
}
}
return status;
}
```

Die Funktion *RsdGetRequestor* bezieht den SID des Users, der die I/O Operation ausführt und kopiert den String in die Variable *Sid* vom Typ *wchar_t[]* in die Struktur *PRBAC_SYSTEM_MESSAGE* welche das Containerformat für Nachrichten zwischen Kernel und User-Mode Software darstellt.

4.3.1.3.2 Ziel ermitteln

Die zweite Frage, auf welches Objekt der Request abzielt, kann auch mittels der Datenstruktur *FLT_CALLBACK_DATA* also der Variable *Data* gelöst werden, indem man einen Aufruf der Funktion *FltGetFileNameInformation* tätigt.

```
PFLT_FILE_NAME_INFORMATION nameInfo = NULL;
```

```
PUNICODE_STRING nameToUse;
PWSTR norm;

FltGetFileNameInformation(Data,
FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_ALWAYS_ALLOW_CACHE_LOOKUP,
&nameInfo);

norm = nameInfo->Name.Buffer;
RsdRemoveFileStream(norm);
```

Dies wird im *RBACSystemDriver* in der Callback-Funktion – für den *IRP_MJ_CREATE* Request registiert wurde – erledigt und nicht in eine separate Funktion ausgelagert. Der Aufruf der Funktion *FltGetFileNameInformation* bezieht den normalisierten Pfad-String des Objektes.

Ein Pfad gilt als normalisiert wenn alle nachfolgenden Punkte zuftreffen

- Er muss den vollen Verzeichnispfad enthalten. Im Verzeichnispfad ist auch der Volume Name enthalten. Wenn der Benutzer das File über eine File-ID öffnet, aber nicht die Erlaubnis hat, den vollen Pfad bis zur Wurzel zu traversieren, so kann in diesem Fall auch nur ein Teil des Verzeichnisses im Namen enthalten sein.

- Der Volume Name ist ein nicht persistenter Geräte Objekte Name wie beispielsweise `\Device\HarddiskVolume1`. Dieser Namen wird dann auf einen Laufwerksbuchstaben abgebildet, z.B `C:\`, wenn C der Laufwerksbuchstabe des ersten Volumes im System ist. Da C aber auch beispielsweise das zweite Volume in einem Computersystem repräsentieren könnte gilt der Volume-Name als nicht persistent.

- Alle Short Names – das sind Datei und Verzeichnisnamen mit 8 Zeichen einem Punkt und drei Erweiterungszeichen – sind auf die volle Länge ausgedehnt Beispielsweise `Docume~1` entspricht `Documents and Settings`. Die 8.3- Darstellungsform für Datei- und Verzeichnisnamen stammt aus dem FAT Dateisystem.

- Nachfolgende `:$DATA` oder `::$DATA` werden vom NTFS-Stream eliminiert.

- Alle NTFS Mount Points werden in den vollen Namen aufgelöst

Um ein Objekt eindeutig indentifizieren zu können wurde bei *RBACSystem* ein Full Quailfied Object Name (FQON) verwendet. Dieser setzt sich zusammen aus Laufwerksbuchstaben, dem vollen Verzeichnis-Pfad und dem Namen des Objektes. Dieser wird in

der Datenbank abgespeichert und dient zum Vergleich im Falle eines Zugriffes auf das Objekt.

Da der normalisierte Pfad eines Objektes nicht mit der FQON-Repräsentation von *RBACSystem* in der Datenbank übereinstimmt sind einige Übersetzungsarbeiten notwendig, damit er verglichen werden kann. Das System muss beispielsweise eine Übersetzung zwischen Laufwerksbuchstaben und Volume-Names durchführen. Weiters werden Probleme auftauchen, wenn ein Benutzer eines Shares nicht ausreichend Rechte besitzt, den vollen Pfad zu traversieren und somit das Verzeichnis nicht vollständig im normalisierten Namen aufscheint.

Nachfolgend ein Beispiel:
Lokaler Zugriff auf eine Datei Test Results.txt:

```
\Device\HarddiskVolume1\Documents and Settings\MyUser\
My Documents\Test Results.txt:stream1
```

Auf der Zielmaschine, also genau dort, wo *RBACSytem* installiert ist, muss eine Übersetzung von `\Device\HarddiskVolume1\` in den dort gültigen Laufwerksbuchstaben erfolgen, denn dieser ist in der Datenbank eingetragen. Dieser Schritt wird vom *RBACSystemMonitor*-Service (*RBACSystemService*) in der Funktion *TranslateNormalized* übernommen und ist derzeit hart codiert.

Der Stream ist ebenfalls Bestandteil des normalisierten Dateinamens, hat jedoch bei der Abfrage keinerlei Bedeutung bzw. stört in diesem Fall sogar, da er in der Datenbank nicht enthalten ist. Jegliche Zeichenfolgen nach einem Doppelpunkt werden also vom Namen eliminiert. Dieser Schritt erfolgt bereits in *RBACSystemDriver* in der Funktion *RsdRemoveFileStream*.

In der Variable *Npwos* ein Wide-Character Array (*wchar_t[] Npwos*) des Message Containers *RBAC_SYSTEM_MESSAGE* wird der normalisierte Pfad Name ohne Streams des Objektes eingetragen, auf das zugegriffen werden soll.

4.3.1.3.3 Modus ermitteln

Nun muss nur noch der Modus des Requests ermittelt werden. Wie zu Beginn erwähnt reagiert der Filter auf alle Requests, welche den Wert *IRP_MJ_CREATE* in der MajorFunction aufweisen. Andere Requests zu bearbeiten und diese an den Usermode zu senden, wäre unnötige Arbeit und Kommunikationsaufwand und daher ist der erste Moment der beste zu entscheiden, ob Interesse für einen Request besteht oder nicht.

Wenn der Filter an einem Request interessiert ist, so muss auch die genaue Art in die Message für die User-Mode Software eingetragen werden. Im Container *message* vom Typ *PRBAC_SYSTEM_MESSAGE* befindet sich ein Feld vom Typ *ULONG* mit dem Namen *AccessMask*. Die Funktion *RsdGetMode* ermittelt den Modus, in welchem auf das Objekt zugegriffen werden soll und speichert ihn als Integer-Wert in die Variable *AccessMask*.

Die Modi reichen von generischen Lese und Schreibzugriffen bis hin zu Rechten zur Modifikation der NTFS Rechtevergabe.

```
NTSTATUS
RsdGetMode(__in PFLT_CALLBACK_DATA Data)
{
NTSTATUS status;

if(Data->Iopb->Parameters.Create.SecurityContext->AccessState==NULL)
{
PT_DBG_PRINT( PTDBG_CONNECTION_ROUTINES, ("KM!RsdGetMode: missing AccessState\n"));
return STATUS_UNSUCCESSFUL;
}

gMessage->AccessMask = Data->Iopb->Parameters.Create.SecurityContext->
AccessState->OriginalDesiredAccess;

return STATUS_SUCCESS;
}
```

Damit ist die Nachricht für die User Mode Software komplett und kann übermittelt werden.

4.3.2 RBACSystemMonitor

Der *RBACSystemMonitor* ist ein in C# programmierter Windows Dienst. Er übernimmt die Funktion des Referenzmonitors. Die Hauptfunktionalität ist in der Funktion *OnMonitorAccess* implementiert. Diese auf den Delegate *MonitorAccessDelegate* gecastete und mit dem Funktionpointer *NATIVEMONITORACCESSPROC* auf Seite von *RBACSystemImed* verbundene Funktion nimmt als Parameter *sid* (Security Identifier), *npwos* (Normalized Path without Stream) und *mode* entgegen, startet eine Query über die Methode *MonitorAccess*. Über einen Service Controller ist der Dienst mit der

Windows Forms Application *RBACSystemServiceCtrl+Mgmt* verbunden. Die Methode *OnCustomCommand* führt Anweisungen aus, die von *RBACSystemServiceCtrl+Mgmt* gesendet werden. Zusätzlich existiert ein Logging Mechanismus, der Zugriffe in der Windows-Ereignisanzeige verzeichnet. Ausserdem löst er noch ein weiteres Problem.

Wie bereits in der Beschreibung des Treibers erwähnt, erhält der Service einen normalisierten Pfad ohne Streams um das Objekt des Zugriffes zu identifizieren. Bei den ersten Testläufen wurde deutlich, dass Windows an den normalisierten Pfad noch weitere Zeichen anhängt. Einige davon sind Unicode-Surrogate Characters im Bereich U+D800 bis U+DFFF. Aber auch im Bereich von 0 bis 127 der Unicode Characters waren immer wieder Anhängsel an den Datei und Verzeichnisnamen zu erkennen, die eine Identifikation mit den in der Datenbank gespeicherten Strings unmöglich machten. Auch waren immer wieder verschiedene Systemprogramme, wie beispielsweise *desktop.ini*, welche für die Appearance der Ordner im Explorer lesende Tätigkeit durchführen muss, im Pfad verzeichnet, obwohl sie gar nicht in dem betreffenden Ordner lagen. Nach verschiedenen Versuchen, diese Zeichenketten mit Filtern zu eliminieren, blieb als letzte Möglichkeit die Implementierung der Methode *pathChecking*. Dabei greift der Dienst selbst auf das Dateisystem zu und versucht zu verifizieren, welche Zeichen zu dem gerade übergebenen Pfad gehören und welche nur Windows-Intern an den Pfad angehängt wurden. Da der Service nun Leserechte auf dem Dateisystem benötigt, muss er nun quasi über seinen eigenen Request entscheiden.

Der Monitor erhält also von *RBACSystemDriver* eine Nachricht, dass jemand versucht, auf einen gewissen Pfad im Dateisystem unter einem bestimmten Modus zuzugreifen. Er muss daraufhin den in der Nachricht enthaltenen Pfad auf seine korrekte Form prüfen und ihn gegebenenfalls korrigieren, dies realisiert er in der Methode *pathChecking*. Damit greift er selbst auf das Dateisystem zu und tritt dabei einen weiteren I/O Request los, sollte also die daraus resutlierende Nachricht vom Treiber ebenfalls beantworten. Er selbst wartet aber für seine Antwort auf ersten Request auf das Resultat seiner Prüfungsmethode *pathChecking* und ein Deadlock entsteht. Eine mögliche Lösung wäre über Threading zu realisieren, jedoch wurde eine wesentlich simplere und effizienter Lösung gewählt. In der Funktion *RsdVerfiyAccess* wird einfach jeglicher Request mit TRUE beantwortet, der den SID S-1-5-18 trägt. Dies ist auf jedem Windows-Betriebsystem der *Local System* Account, unter dem absofort der Service immer laufen muss, damit *RBACSystem* funktioniert. Unter *Local System* kann man prinzipiell jeden Windows-Dienst starten. Damit kann auch jeder installierte Dienst, der das *Local System*-Konto nutzt, ohne weitere Prüfung auf *RBACroot* zugreifen. Dies muss beachtet werden, falls File-Server – zum Beispiel FTP, WebDAV usw. – unter dem Trusted Subsystem Model auf dem System installiert werden!

4.3.3 RBACSystemImed

RBACSystemDriver kontrolliert den I/O Strom vom I/O Manager zum Filesystem, leitet ihn weiter oder unterbricht ihn. Damit unterscheidet er einen binären Zustand – die Operation erlauben oder verweigern. Die für diese Entscheidung notwendigen Grundlagen bezieht er von einem Referenzmonitor, der wiederum eine Datenbasis befragt, in der die Zugriffsrechte verzeichnet sind. Der Referenzmonitor wird durch die Komponente *RBACSystemMonitor*, einem .NET Windows Dienst, realisiert.

Hier existiert die Herausforderung eine Kommunikation zwischen dem Treiber und dem Referenzmonitor zu ermöglichen. Microsoft Windows unterscheidet bei der Ausführung zwischen dem Kernel-Mode und dem User-Mode. Im User-Mode laufen alle vom Benutzer oder Betriebsystem gestarteten Anwendungen. Im Kernel-Mode alle Betriebsystem spezifischen Routinen, welche die Verwaltung der Ressourcen (Geräte, Rechenzeit, Speicher) steuern. Treiber, so auch der File-System Filter Driver, fallen daher unter die Gruppe der Kernel-Mode Komponenten. Eine Kernel-Mode Komponente kann nicht direkt auf eine User Mode Komponente zugreifen und vice versa, da sie im virtuellen Speicher jeweils in unterschiedlichen, streng voneinander getrennten Bereichen ausgeführt werden. Diese Architektur-Entscheidung wurde von Microsoft bewusst gewählt, da man davon ausgeht, dass Kernel-Mode Komponenten untereinander implizit immer vertrauen und somit nicht vertrauenswürdige Komponenten außen vor gehalten werden müssen. Deutlich wird dies, wenn man einen Treiber installiert. So muss dieser entweder digital signiert sein, oder der Benutzer wird explizit danach gefragt, ob er die Installation fortsetzen möchte, wenn der Treiber den Validierungs-Prozess der digitalen Signatur nicht bestanden hat oder schlichtweg keine Signatur enthält. Einer einmal installierten Kernel Komponente vertraut das Betriebsystem also blind. Kernel-Mode Komponenten dürfen User-Mode Komponenten nicht blind vertrauen. Im Gegenteil – Sie müssen Daten und Adressen jeglicher Art, die sie von User-Mode Komponenten erhalten, immer validieren, um die Systemsicherheit aufrecht zu erhalten.

Microsoft definiert zwei Grundsätze, die beim Entwurf von Filtern beachtet werden sollten [4]:

- Kernel Mode Komponenten dürfen keine User Mode Komponenten rufen. Damit ist sichergestellt, dass eine Kernelkomponente keine vertraulichen Daten unmotiviert in den User-Mode leitet.

- Kernel Mode Komponenten müssen alle Adressen und Daten die sie von User Mode Komponenten erhalten validieren.

Policy Entscheidungen und Interaktionen mit dem User sollten immer von User-Mode Komponenten behandelt werden. Policies sollten niemals den Treiber erreichen. [4] Kernel Mode Komponenten sollten nur Aufgaben übernehmen, die nicht von User-Mode Komponenten bewerkstelligt werden können. Treiber können keine Win32 Routinen ausführen sondern sind auf die Driver Support Routines beschränkt[5]. Damit wäre ein Datenbankzugriff aus einem Treiber heraus nicht nur eine schwere Verletzung der Design-Guidelines sondern schlicht und einfach nicht möglich.

Damit muss der Datenbankzugriff von einer Win32 User-Mode Applikation durchgeführt werden. In Folge besteht genau hier das Kommunikationsproblem zwischen dem Kernel-Mode Driver und der User-Mode Applikation, da beide miteinander interagieren müssen. Microsoft bietet hierfür folgende Varianten an:

4.3.3.1 Plug and Play Notification

Plug & Play ist ein Event, für den sich beliebige User und Kernel Mode Komponenten registrieren können und somit benachrichtig werden, wenn ein *Plug & Play* Event für ein spezielles Gerät erfolgt. Dies erfolgt beispielsweise beim Anstecken eines USB Sticks. *Plug & Play* ist eine Einweg Kommunikation vom Treiber zu den Applikationen. Durch die vom Treiber initiierte Kommunikation darf keine Antwort vom User-Mode erfolgen, da dies eine Verletzung der Richtlinien zum Entwurf von Treibern darstellen würde, bzw. wäre es gar nicht feststellbar, von welcher auf den *Plug & Play* Request reagierenden User Mode Applikationen die Antwort kommt. Wenn eine Antwort von der User Mode Applikation erforderlich ist, so sind Driver Defined IOCTLs das Mittel der Wahl.

4.3.3.2 Driver Defined IOCTLs

Die User-Mode Komponente erzeugt einen seperaten Thread und sendet *DeviceIoControl* – einen I/O Request – an den Treiber. Der Treiber retourniert den Status *STATUS_PENDING*. Um die User Mode Komponente zu rufen komplettiert er den Request. Diese Technik ist mit einer gewissen Ineffizienz verbunden, da ein Treiber prinzipiell den Status *STATUS_PENDING* für unbestimmte Zeit für den besagten I/O Request halten kann.

Diese beiden Techniken sind prinzipiell für die Notification in uni- bzw. bidirektionale Richtung geeignet, jedoch fehlt die Möglichkeit, Informationen zu übertragen. Hierfür

[5] http://msdn.microsoft.com/en-us/library/ms795146.aspx

existieren Mechanismen wie Shared Handles, bzw. Shared Memory auf die hier nicht weiter eingegangen wird.

Grund dafür ist, dass bei der Implementierung eines Minifilters keine der oben genannten Varianten von Nöten ist. Ein weiterer Vorteil, wenn man sich für die Implementierung eines Minifilters entscheidet. Dieser erlaubt nämlich eine bidirektionale Kommunikation zwischen User-Mode und Kernel-Mode über Ports. Die Sicherheit bei den Minifiltern ist dadurch gegeben, dass das Port-Objekt mit einem Security Descriptor ausgestattet wird. Somit kann nicht jede beliebige User-Mode Applikation auf den Treiber zugreifen. Der Treiber erhält implizit eine Bestätigung der Autorität des Rufers über den Access Token des selbigen.

Der Treiber erzeugt und registriert mit dem Aufruf von *FltCreateCommunicationPort* ein Port Objekt. Auf diesem lauscht er nun für einkommende Verbindungsversuche. Falls sich nun eine User-Mode Komponente auf diesen Port verbindet, so wird vom Treiber die *ConnectNotifyCallback* Routine ausgeführt. Auf Clientseite erfolgt dies durch den Aufruf von *FilterConnectCommunicationPort* – einer Minifilter User- Mode Application Function. Der Filtermanager übergibt der User-Mode Komponente den Endpunkt der Kommunikation, ein spezielles File-Handle. Somit steht eine Verbindung zwischen User-Mode und Kernel-Mode Komponente, die einseitig aufgelöst werden kann. Beendet die User-Mode Komponente die Kommunikation, so wird im Treiber die Routine *DisconnectNotifyCallback* aufgerufen und der Treiber erhält die Chance, sein Handle zu der beendeten Verbindung zu schließen. Wichtig dabei ist, dass je nachdem wie der Port zu Beginn konfiguriert wurde, mehrere Verbindungen von verschiedenen User-Mode Komponenten möglich sind. Dabei erhält jede Verbindung ihre separate Message-Queue und private Kommunikations-Endpunkte. Das Versenden von Nachrichten erfolgt in einem Message-Buffer – ein simpler Zeiger des Datentyps *VOID*. Die beiden Endpunkte müssen sich also im Klaren sein, wie die Struktur der übertragenen Daten im Detail aussieht. Die Kernel Mode Komponente, sprich der Treiber reagiert mit der *MessageNotifyCallback* Routine auf einkommende Nachrichten. Auf Seite der User Komponete lautet die Funktion *FilterGetMessage* zum empfangen der Nachricht, zum senden *FilterSendMessage* Auf Seiten des Kernel Mode Drivers lautet die Funktion zum senden *FltSendMessage*.

Ursprünglich war vorgesehen, dass die Kernel-Mode Komponente *RBACSystemDriver* direkt mit der .NET Komponente kommuniziert. Nach zahlreichen Versuchen, den Aufruf über Native Calls aus dem managed C# Code zu realisieren und einer Anfrage im Forum von OSR Online[6] fiel die Entscheidung, als Kommunikationsendpunkt auf der User-Mode Seite eine native Win32 DLL zu verwenden, da die Kommunikation auf nativer Ebene anstandslos funktionierte. *RBACSystemImed* dient also dazu, die Kommu-

[6] http://www.osronline.com/

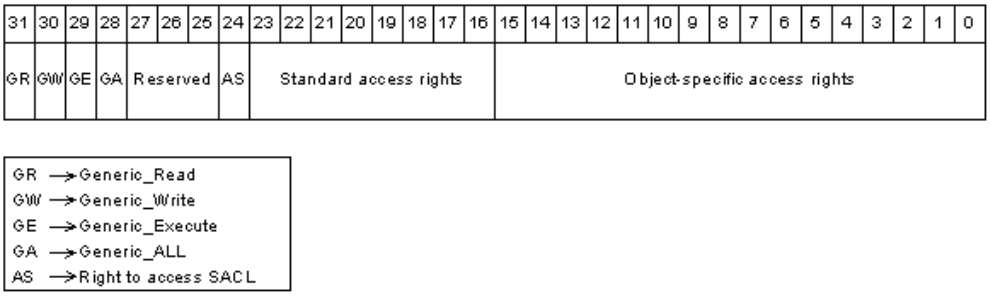


Abbildung 4.3: Windows Access Mask, Quelle: [2]

nikation mit der Kernel-Mode Komponente *RBACSystemDriver* abzuhalten und über .NET native Calls und Delegates mit der User Mode Komponente *RBACSystemMonitor* zu kommunizieren. Sie stellt daher wie im Namen auch ersichtlich einen Mediator zwischen dem Treiber und dem Referenzmonitor dar.

4.3.3.3 Übersetzung der Zugriffsrechte

In *RBACSstemImed* ist auch die Übersetzung von Microsoft auf Unix-Access Modes realisiert, damit leitet die Dynamic Link Library die Nachricht vom Treiber zum Service nicht nur weiter sondern modifiziert sie auch. Zugriffsrechte sind in Windows in einer 32 bit AccessMask organisiert wie in Abbildung 4.3 dargestellt.

Die 4 most significant bits repräsentieren die generischen Rechte. Dahinter kommen die standard Berechtigungen und die objektspezifischen Berechtigungen. Wichtig für *RBACSystem* sind die objektspezifischen Berechtigungen für Dateien, da File-Server mit *RBACSystem* abgesichert werden sollen. Die Umsetzung der Microsoft Rechte erfolgt in der Funktion *MicrosoftToUnixAccessMode* unter Verwendung der Übersetzungstabelle 4.1.

4.3.4 RBACSystemDatabase

Der Referenzmonitor, welcher über den Windows Dienst *RBACSystemMonitor* realisiert wird, trifft seine Entscheidungen aufgrund der Berrechtigungseinträge in einer Datenbasis. Windows bietet hierfür den Authorization Manager an, der bereits in einer vorangegangenen Praktikumsarbeit [14] behandelt wurde. Dabei stellt er eine Datenbasis für Applikationen über den Policy-Store bereit. Die Daten-Repräsentations-Varianten des Policy Store umfassen XML, MS SQL und Active Directory. Er beinhaltet die User, Permissions (dort Tasks und Operations genannt) als auch die Rollen und die Verknüpfungen dieser Elemente. Die nützende Applikation greift dann auf den Policy-Store auf

Name	MS	Unix
FILE_ADD_FILE	0x0002	2
FILE_ADD_SUBDIRECTORY	0x0004	2
FILE_ALL_ACCESS		7
FILE_APPEND_DATA	0x0004	2
FILE_CREATE_PIPE_INSTANCE	0x0004	2
FILE_DELETE_CHILD	0x0040	2
FILE_EXECUTE	0x0020	1
FILE_LIST_DIRECTORY	0x0001	4
FILE_READ_ATTRIBUTES	0x0080	4
FILE_READ_DATA	0x0001	4
FILE_READ_EA	0x0008	4
FILE_TRAVERSE	0x0020	1
FILE_WRITE_ATTRIBUTES	0x0100	2
FILE_WRITE_DATA	0x0002	2
FILE_WRITE_EA	0x0010	2

Tabelle 4.1: MS Zugriffsrechte für Datei Objekte, Quelle: [2]

programmatischem Weg zu, trifft die Zugriffsentscheidung und greift dann für den ausführenden User auf die Ressource zu. Dieses Konzept wird Trusted Subsystem Model genannt, bei dem die Applikation für den User auf die Ressource zugreift, während beim Impersonation Model die Applikation den Access Token des Users bezieht, ihn damit impersonalisiert und dann unter Verwendung seines Accounts agiert.

Auch für RBACSystem wurde angedacht, die Entscheidungsgrundlage für den Referenzmonitor in Microsofts Authorization Manager zu realisieren. Der Authorization Manager hat jedoch zwei Nachteile: Er hat kein integriertes Session Konzept. Das bedeutet, wenn User und Permissions erst mal den Rollen zugewiesen wurden, so ist diese Zuweisung statisch, und nicht mehr ohne Interaktion des Administrators veränderbar. Wenn nun der Policy-Store mit einer Query aus der Applikation heraus abgefragt wird, so betrifft diese Abfrage immer die statische Zuweisung. Damit kann ein Sessionkonzept nur über programmatischem Wege realisiert werden. Im RBAC0 Model, als auch im ANSI/INCITS RBAC Core Model ist jedoch das Sessionkonzept als zentrales Merkmal vorhanden. Die Lösung erfolgte in der Praktikumsarbeit [14] über dynamische Kopiervorgänge in Subfolder innerhalb des Authorization Managers, welche dann als Session Container angesehen wurden und für eine Query in einem aufwändig programmierten Mechanismus abgefragt wurden. Außerdem war die Session Menge wieder beschränkt, da in der Applikation a priori festgelegt werden musste, welche Subfolder für die Zugangsprüfung abgefragt werden müssen bzw. wohin die Rollen zu kopieren sind.

Der zweite Grund, der gegen die Verwendung des Authorization Managers sprach, war ebenfalls ein programmatischer. Wenn ein User eine Rolle in einer Session aktivieren möchte, so muss sichergestellt sein, dass ihm diese auch vorab zugewiesen wurde. In

der Praktikumsarbeit wurde dies ebenfalls über die Programmierung der Applikation sichergestellt, indem in der GUI nur die Rollen für die Aktivierung angeboten wurden, die bereits statisch zugewiesen waren. Im Authorization Manager, in welchem die Elemente durch den Administrator beliebig verwaltet werden können wäre ein Kopiervorgang einer nicht vorab zugewiesenen Rolle in einen der Session-Folder in jedem Fall ohne Weitere Sicherheitsprüfung möglich. Im Falle der Praktikumsarbeit war es nur ein Prototyp, der lediglich die Funktionsweise konzeptuell zeigen sollte. Daher war sowohl die Client- als auch die Serverfunktionalität in einer Applikation konzipiert. Wenn allerdings nun die Client-Applikation auf einen Client-Host verlagert wird, was zwingend notwendig ist, damit User ihre Sessions aktivieren und verwalten können, so ergibt sich ein weiteres Sicherheitsproblem. Böswillige Benutzer könnten die Software so manipulieren, dass sie in der Lage wären, Rollen zu aktivieren, die sie gar nicht besitzen, da die Sicherheitsprüfung in der Applikation durchgeführt wird, die lokal auf ihrem Rechner liegt. Damit wird deutlich, dass diese Prüfung vom Anwender entfernt und zentralisiert durchgeführt werden muss. Der Authorization Manager kann zwar zentral platziert werden, bietet jedoch keinerlei Werkzeuge für die Sicherheitsprüfung.

Die Entscheidung fiel daher auf eine MS SQL Server 2008 Datenbank. Zum einen, weil dort das Sessionkonzept ohne viel programmatischen Aufwand effizient umgesetzt werden kann, zum anderen, weil eine signifikante Sicherheitsverbesserung damit erzielt werden konnte. Außerdem erspart man sich damit viele unnötige proprietäre Socket Connections und andere Verbindungen, wie aus dem Konzept später ersichtlich werden wird. Ein weiterer Vorteil liegt in der effizienten Replikation im Falle des Einsatzes von Windows Domänen und Active Directory.

Die Datenbank enthält 9 Tabellen, drei Views und eine SQL-Funktion.

```
host {VARCHAR(50) name, VARCHAR(50) descript}
```

Die Tabelle Host speichert alle Server Hostnamen ab, auf denen ein RBAC System laufen soll.

```
perm {VARCHAR(50) name (NOT NULL), VARCHAR(50) descript,
VARCHAR(128) fqon, int mode, VARCHAR(100) host.name}
primary key: {fqon, mode, host.name}
foreign key: host.name
```

Diese Tabelle steht für die Permissions. Eine Permission setzt sich in *RBACSystem* aus dem Host-Namen auf dem sie gültig ist, dem „Full Qualified Object Name“ , also dem Objekt auf das sie zutrifft und dem Modus also der Zugriffsart zusammen.

```
usr {VARCHAR(80) sid, VARCHAR(50) name, VARCHAR(50) pwd}
primary key: sid
```

Die Tabelle usr beinhaltet alle User, die am RBACSys partizipieren. Die User werden über ihren Security Identifier eindeutig indentifiziert.

```
roles {VARCHAR(50) name, VARCHAR(50) descript}
primary key: name
```

Die Tabelle roles beinhaltet die Rollendefinitionen die in der Regel eine Job-Funktion im Unternehmen ausdrücken.

Dies sind die Basiselemente für das RBACSystem. Sie werden über die folgenden Relationen miteinander verbunden.

```
u2r {VARCHAR(80) usr.sid, VARCHAR(50) roles.name}
primary key: {usr.sid, roles.name}
foreign key: {usr.sid, roles.name}
```

```
p2r {VARCHAR(50) perm.host.name, VARCHAR(100) perm.fqon,
int perm.mode, VARCHAR(50) roles.name}
primary key: {perm.host.name, perm.fqon, perm.mode, roles.name}
foreign key: {perm.host.name, perm.fqon, perm.mode, roles.name}
```

u2r und *p2r* stellen das $n : m$ Mapping von User und Permissions über das Intermediationskonzept Rolle dar, wie es der RBAC Standard vorsieht.

```
session {int id (Identity), VARCHAR(80) usr.sid (NOT NULL)}
primary key: id
foreign key: usr.sid
```

Verbindet einen User aus usr mit einer Session-ID.

```
r2s {VARCHAR(50) roles.name, int session.id}
primary key: {roles.name, session.id}
foreign key: {roles.name, session.id}
```

Verbindet die Session-ID mit Rollen aus roles.

```
msid {VARCHAR(80) usr.sid, VARCHAR(80) msid, VARCHAR(50) reshost}
primary key: {usr.sid, msid}
foreign key: {usr.sid}
```

Verbindet die SID eines Server-Computer-Kontos mit der eines Host-Computer-Kontos.

Auf der Table *r2s* wurde ein Constraint definiert. Dieses verhindert, dass für den User, der die Session – identifiziert über die Session-ID – hält, Rollen eingetragen werden können, wenn sie nicht bereits in der Table *u2r* für ihn vorhanden sind. Damit wird

nun das Eingangs erwähnte Sicherheitsproblem gelöst. Die Prüfung erfolgt nicht mehr in der Applikation sondern in der Datenbasis selbst. Ein Angreifer kann keinen SQL String erzeugen und eine Prüfung der Applikation umgehen. Auf *u2r* erhält er keine Rechte, da dort nur der Administrator über die *RBACSystemServiceCtrl+Mgmt* Applikation Modifikationen übernehmen muss. Somit kann ein Benutzer auch die notwendige vorab Zuweisung einer Rolle nicht bewerkstelligen.

Selbst über die direkte Modifikation der Datenbank ohne Zuhilifenahme der *RBACSystemServiceCtrl+Mgmt* Applikation ist ein direktes Einfügen in die Tabelle *r2s* ohne ein Konto mit ausreichender Berechtigung nicht möglich – ganz im Gegensatz zum Konsolen-Snap-In des Authorization Managers.

4.3.5 RBACSystemServiceCtrl+Mgmt

Um die Rechtevergabe durchzuführen und den Referenzmonitor Service zu steuern, wurde die in C# programmierte Windows Forms Anwendung *RBACSystemServiceCtrl+Mgmt* Applikation entworfen. Diese Applikation ist eine optionale Komponente und dient zur Verwaltung der Datenbasis und zur Konfiguration des *RBACSystemMonitor* Dienstes. Die Konfiguration der Datenbank könnte beispielsweise auch lokal über die MSSQL Server Management Studio oder remote über eine Command-Shell bewerkstelligt werden. Für die Konfiguration des *RBACSystemMonitor* Dienstes ist sie allerdings notwendig, da nur so Verbindungseinstellungen für den Dienst zur Datenbank verwaltet werden können. Die Abbildungen 4.4, 4.5, 4.6, 4.7, 4.8, 4.9 und 4.10 beschreiben *RBACSystemServiceCtrl+Mgmt*.

4.3.6 RBACSystemClientCtrl

Um dem Anwender die Möglichkeit zu geben, Sessions abzuhalten und darin Rollen zu aktivieren, wurde die *RBACSystemClientCtrl* Applikation entwickelt. Diese in C# programmierte Windows Forms Anwendung verfügt ebenfalls über eine Verbindung zur Datenbank mit eingeschränkten Rechten. Diese umfassen Leseoperationen auf der Tabelle *u2r* und Lese/Schreiboperationen auf *session* und *r2s*. Die Abbildungen 4.11 und 4.12 zeigen die Applikation.

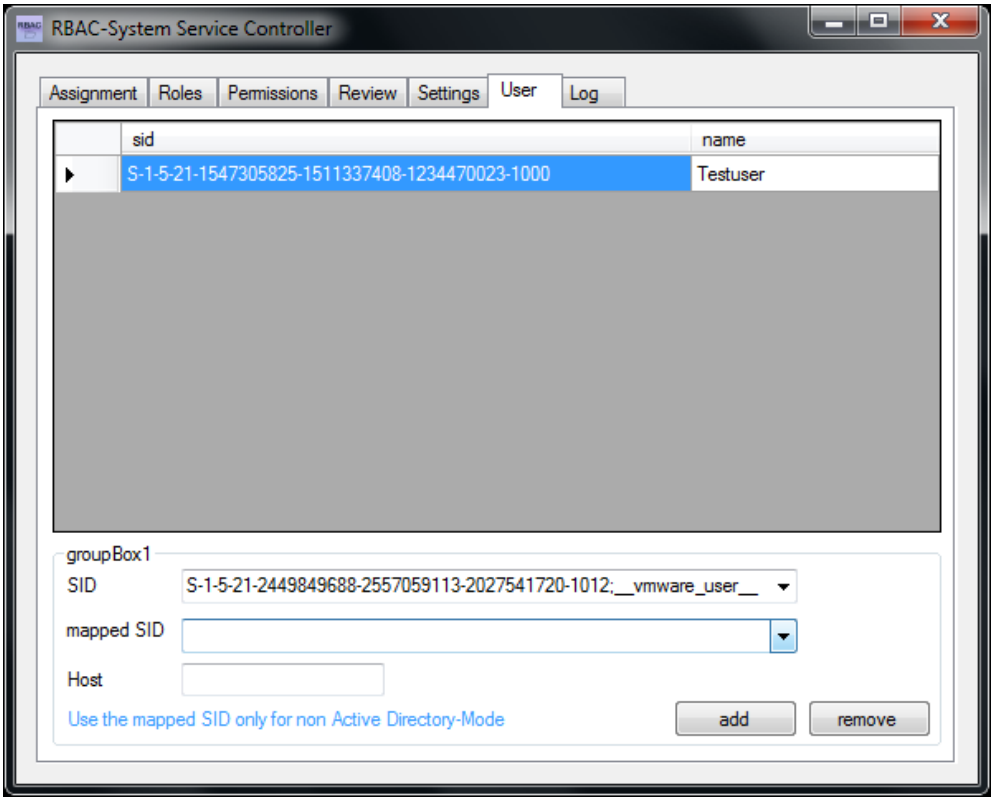


Abbildung 4.4: RBACSystemServiceCtrl+Mgmt User

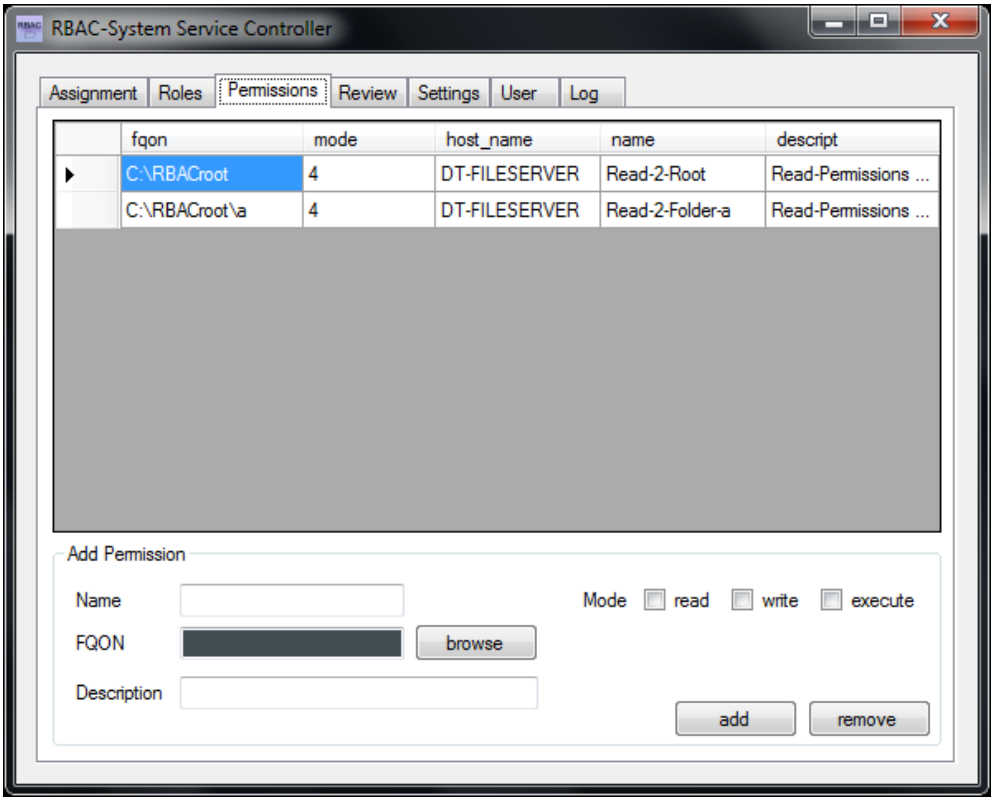


Abbildung 4.5: RBACSystemServiceCtrl+Mgmt Permissions

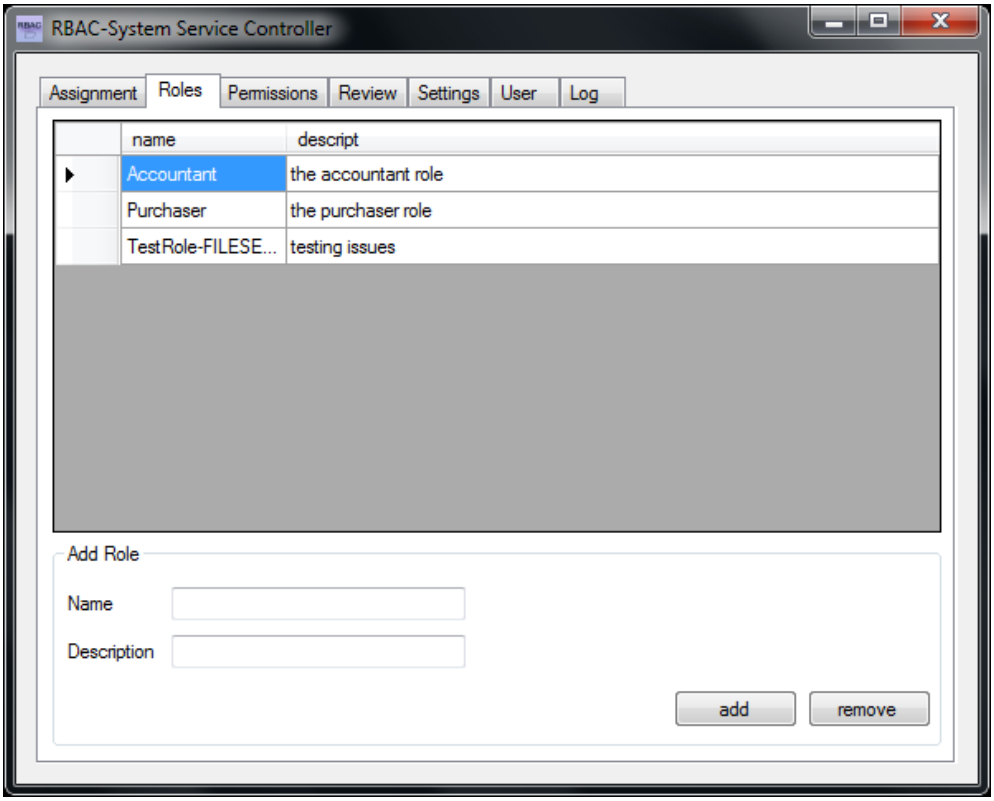


Abbildung 4.6: RBACSystemServiceCtrl+Mgmt Roles

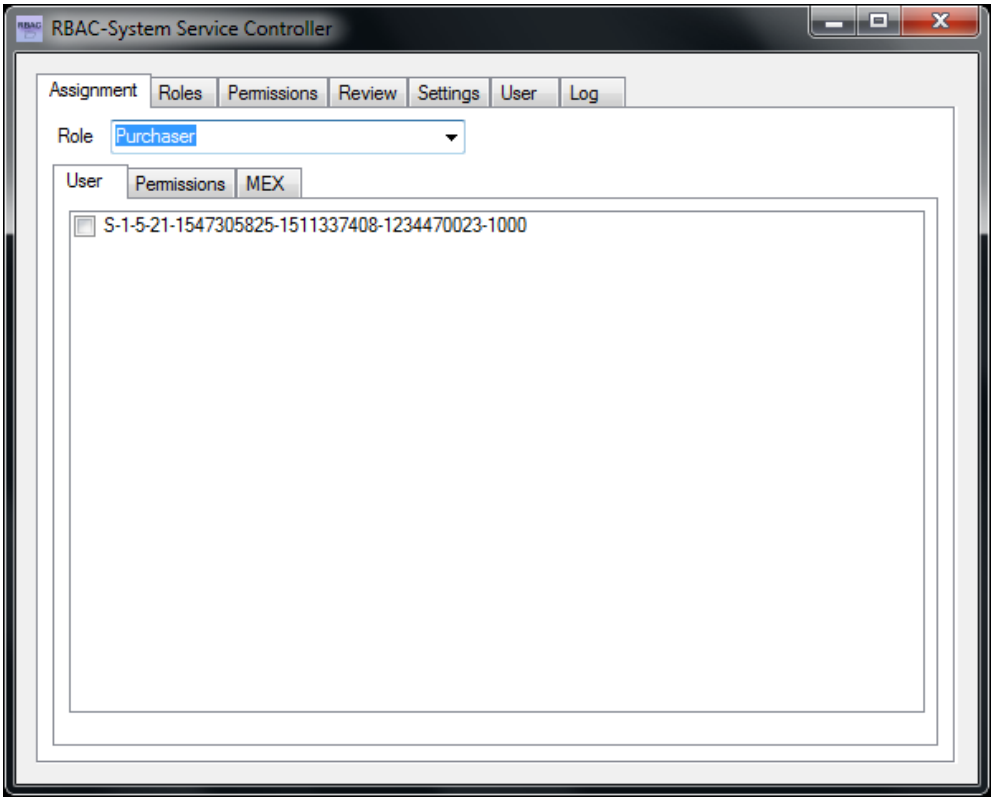


Abbildung 4.7: RBACSystemServiceCtrl+Mgmt Assignment

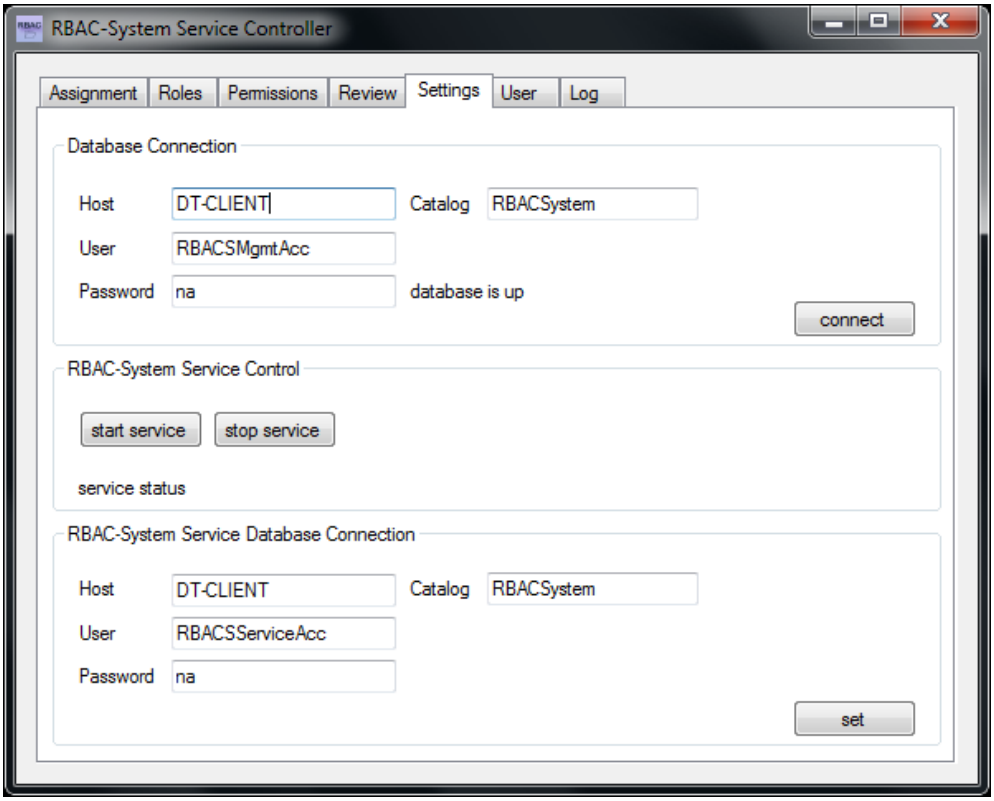


Abbildung 4.8: RBACSystemServiceCtrl+Mgmt Settings

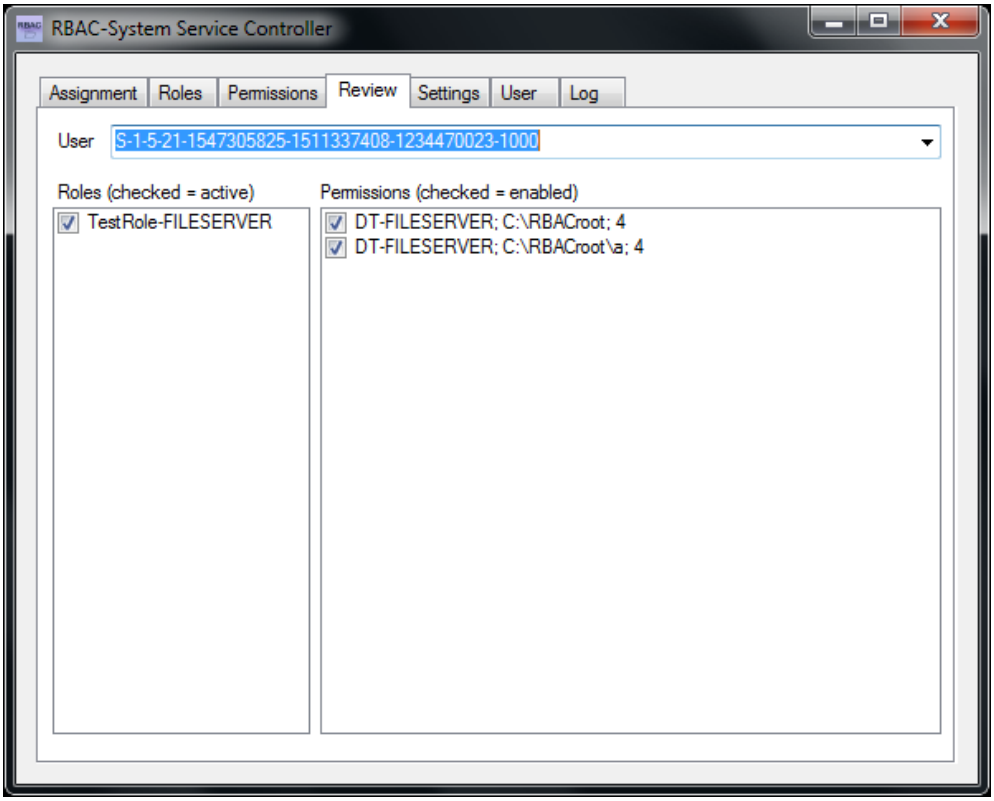


Abbildung 4.9: RBACSystemServiceCtrl+Mgmt Review

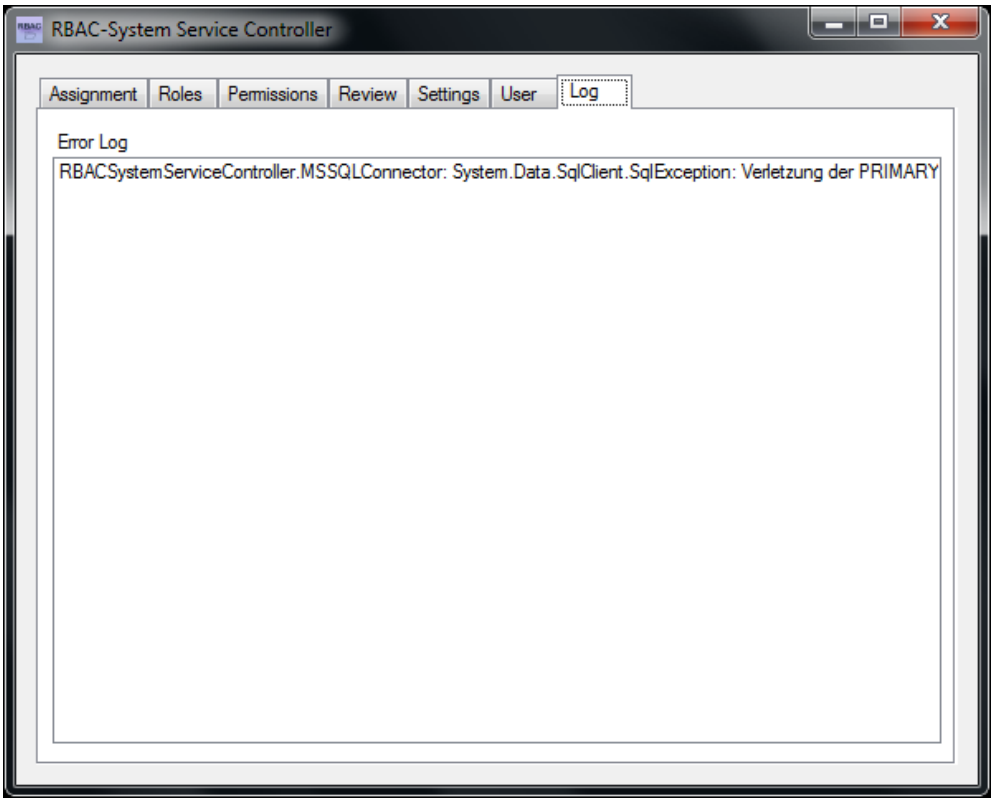


Abbildung 4.10: RBACSystemServiceCtrl+Mgmt Logging

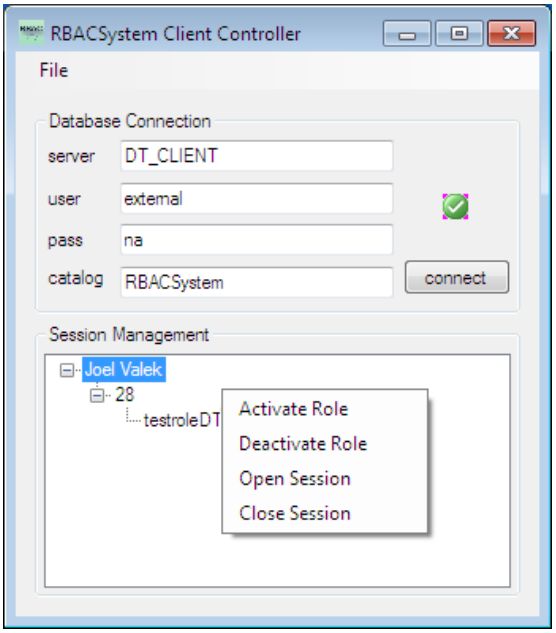


Abbildung 4.11: RBACSystemClientCtrl MainForm

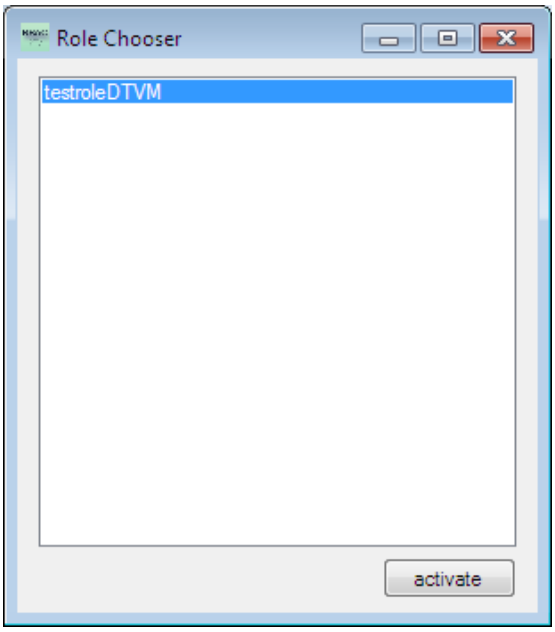


Abbildung 4.12: RBACSystemClientCtrl RoleChooseForm

4.4 Struktur der Testumgebung

Für die Testumgebung wurde als Target OS Windows Server 2008 auf einem Windows 7 Host unter VMWARE Server 2.0[7] virtualisiert. Für das Debugging des File-System Minifilter Drivers *RBACSystemDriver* können zwei unterschiedliche Werkzeuge verwendet werden – *DebugView* und *WinDbg*. Das Tool *DebugView* muss auf dem Host gestartet werden, auf dem auch der Treiber läuft. Damit kann das Tool bei einem Absturz des Treibers keine Aufschlüsse mehr darüber liefern, was zu dem Absturz geführt hat, denn ein Fehler in einer Kernelmode-Komponenten verursacht in der Regel einen Blue-Screen. Daher wurde für die Entwicklung des Treibers auf Hostseite *WinDbg*[8] verwendet, das über einen virtuellen seriellen Port mit dem Guest verbunden wurde und erst später, als diese Komponente fehlerfrei lief, wurde *DebugView* verwendet.

Nachfolgende Eintstellungen sind, wie in Abbildung 4.13 dargestellt, für VMWARE Server notwendig, um den virtuellen seriellen Port bereit zu stellen. Der Name der definierten Pipe lautete `\\.\pipe\dbgpipe` und wurde in den Eigenschaften des virtuellen seriellen Ports eingetragen. *This end is the server* wurde gewählt und *The other end is an application*. Als Standard sollte die Pipe mit der Option *Connect at power on* immer eingeschalten sein und in den erweiterten Einstellungen muss *Yield CPU on poll* angehakt werden, um den Kernel auf der Target Maschine dazu anzuhalten, nicht den Interruptmode für den Zugriff auf den seriellen Port zu verwenden[9].

[7] http://www.vmware.com/de/products/server/
[8] http://www.microsoft.com/whdc/devtools/debugging/installx86.Mspx
[9] http://www.vmware.com/support/gsx3/doc/devices_serial_debug_gsx.html

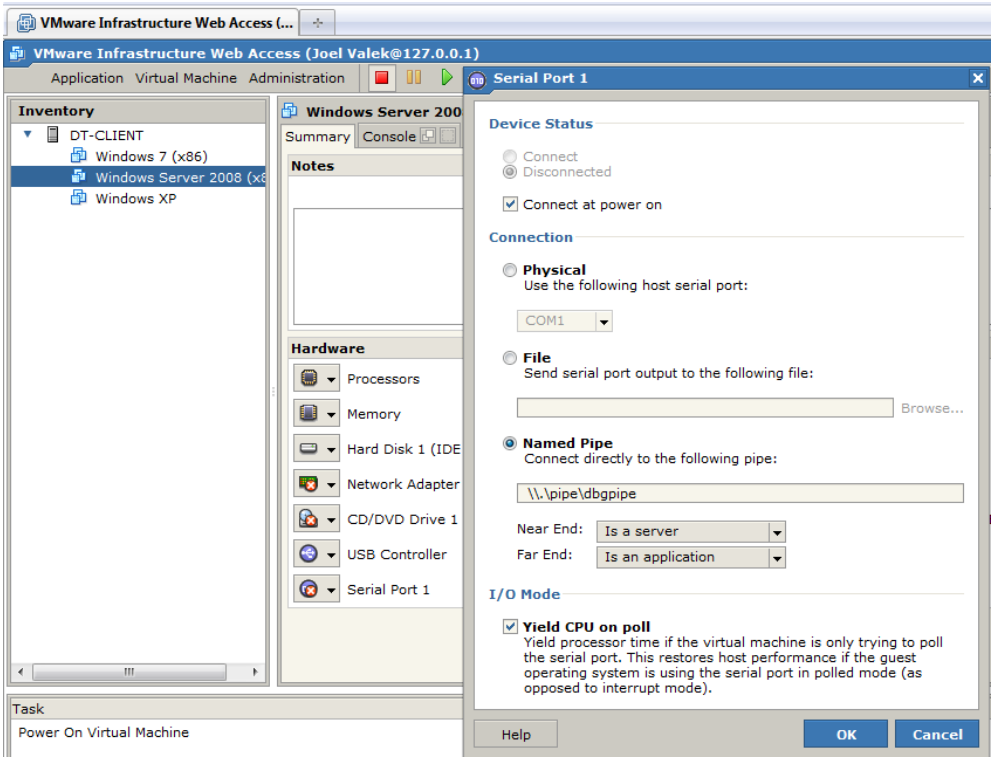


Abbildung 4.13: VMWARE Server Settings

Der Verbindungs-Aufruf vom Host-OS erfolgte über eine Elevated Shell (Ctrl+Shift+Enter bei Eingabe von *Run* → *cmd*) über die Eingabe des Befehls: `windbg -k com:port=\\.\pipe\dbgpipe,pipe`

Auf dem Host-OS lief auch die MS SQL Server 2008 Datenbank, auf die der Referenzmonitor auf Guest-Seite unter Verwendung eines eingetragenen Kontos zugreifen konnte. Auch stellte das Host-OS die Clientseite dar, welche den Zugriff auf den Server durchführte.

4.5 Stuktur der Produktivumgebung

Eben beschriebene Testumgebung ist lediglich für Präsenationszwecke geeignet. Nachfolgend werden alle erforderlichen Hard- und Software-Komponenten beschrieben, die für den realen Einsatz von RBACSystem erforderlich sind, Abbildung 4.14.

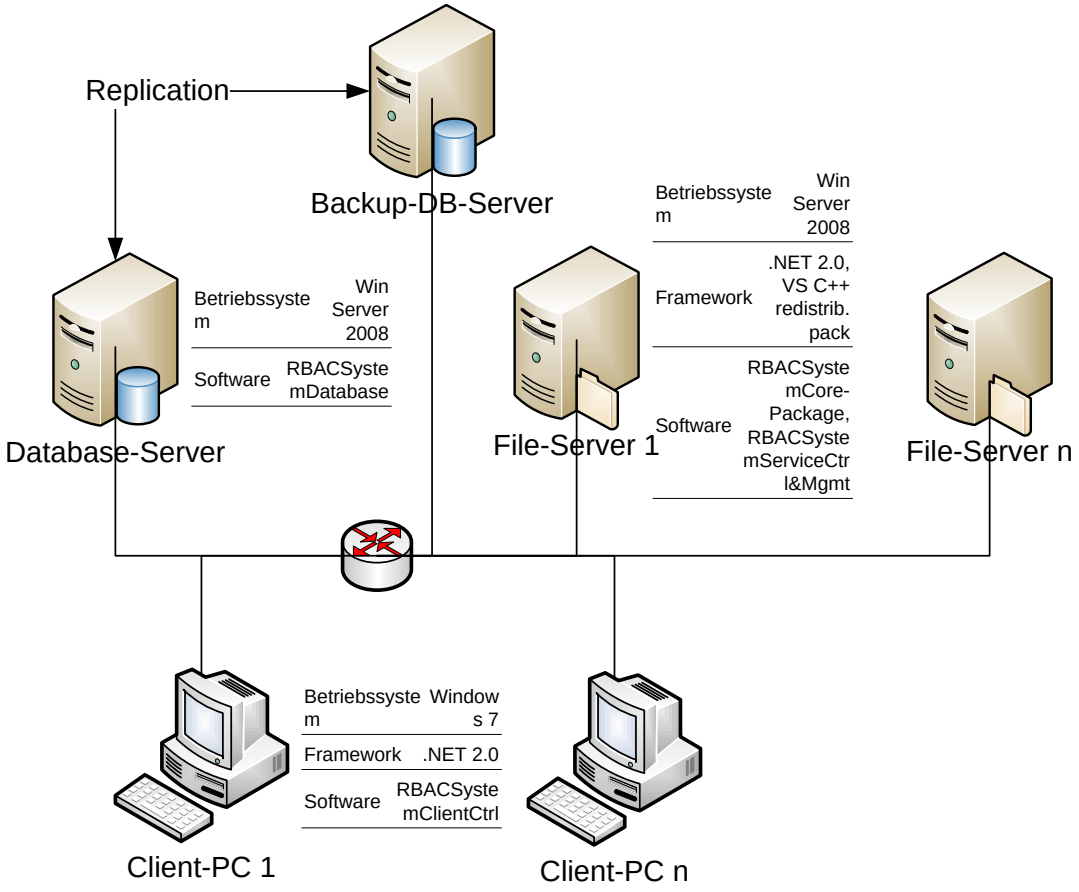


Abbildung 4.14: Produktiv-Plattform-Setup RBACSystem

4.5.1 File-Server

In einem Netzwerk kommen in der Regel mehrere unterschiedliche Server zum Einsatz, die Objekte hosten, welche unter einem rollenbasierten Zugriff stehen können. Dies sind im Falle von *RBACSystem* immer File-Server. Es kommen dafür alle Betriebsysteme der Windows NT Linie ab Windows XP in Frage. Auf dem Testsystem wurde Windows Server 2008 eingesetzt, da bei einer Domänenverwaltung mit Active Directory auch diese Wahl getroffen werden würde. Für die Testzwecke spielt es letztlich keine Rolle, da ein File-Share auch auf anderen Betriebsystemen wie beispielsweise Windows 7 realisiert werden kann und nicht unbedingt Windows Server 2008 dafür notwendig ist. Auf den Servern wird das *RBACSystem*-Core-Package installiert. Weiters läuft auch die *RBACSystemServiceCtrl+Mgmt* Applikation auf jedem Server.

Es ist daher das .NET Framework 2.0 als auch eventuell das Visual C++ Redistributable Package von Nöten. Letzteres, damit *RBACSystemImed.dll* aus dem RBACSystemCore-Package lauffähig ist.

4.5.2 Database-Server

Weiters ist ein Datenbank-Server zu installieren, im AD eventuell auch ein Backup-DB-Server, der sich mit dem Master-DB-Server repliziert. Für diese Maschine kommen alle Betriebsysteme in Frage, auf denen SQL Server 2008 lauffähig ist. Läuft der Datenbank-Server auf einem anderen physikalischen Host als der File-Server, so muss die Netzwerk-Kommunikation, welche zwischen dem *RBACSystem*-Core-Package des File-Server und dem Datenbank-Server statt findet, verschlüsselt werden, damit die Entscheidungsgrundlage für Referenzmonitor nicht getampert werden kann.

4.5.3 Client-PCs

In einer Netzwerk-Struktur existieren mehrere Client-Computer, welche auf Objekte, die unter unter rollenbasiertem Zugriff stehen, zugreifen wollen. Es ist für die *RBACSystemClientCtrl* Applikation das .NET Framework 2.0 erforderlich. Da die Applikation in .NET programmiert wurde, spielt hier das OS keine Rolle solange die .NET 2.0 CLR vorhanden ist.

4.6 Installation des Systems

Das *RBACSystem*-Core-Package setzt sich aus den drei Komponenten *RBACSystem-Monitor*-Service, *RBACSystemImed* und *RBACSystemDriver* zusammen. Zusätzlich sind noch die *RBACSystemDatabase* und die Applikationen *RBACSystemServiceCtrl+Mgmt* und *RBACSystemClientCtrl* zu konfigurieren.

4.6.1 *RBACSystemMonitor* – Installation des Referenzmonitor Dienstes

Aufrufen einer Elevated Shell. Wechsel in das Verzeichnis von *RBACSystemService.exe* und Eingabe von `installutil RBACSystemService.exe`, damit wird der Dienst installiert. Hinweis: Das Programm *installutil.exe* ist Teil des .NET Frameworks, eventuell ist der Eintrag des Pfades zu *installutil.exe* in der Umgebungsvariable *Path* notwendig, damit das Programm gefunden werden kann. Nach erfolgreicher Installation kann er unter Eingabe von `net start RBACSystemMonitor` gestartet werden. Der Dienst kann mit `net stop RBACSystemMonitor` angehalten und mit `installutil /u RBACSystemService.exe` deinstalliert werden. Unter dem Konsolen-Snap-In *services.msc* kann der Dienst auf „Autostart“ gestellt werden. Alternativ kann auch der Registrierungsschlüssel `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RBACSystemMonitor\Start` von 3 (manual) auf den Wert 2 (automatic) gesetzt werden. Damit wird er bei jedem Start des Betriebsystems automatisch gestartet.

4.6.2 *RBACSystemImed* – Installation der Win32 Dynamic Link Library

RBACSystemImed.dll muss lediglich in den Ordner `C:\Windows\system32` kopiert werden.

4.6.3 *RBACSystemDriver* – Installation der Kernel Treiber Komponente

Im Explorer zum Verzeichnis navigieren, wo *RBACSystemDriver.sys* und *RBACSystemDriver.inf* liegen. Rechtsklick auf die Datei *RBACSystemDriver.inf* und im Kontextmenü „installieren“ wählen. Aufrufen einer Elevated Shell, Eingabe von `sc start RBACSystemDriver`. Der Treiber kann mit `sc stopp RBACSystemDriver` angehalten und mit `sc delete RBACSystemDriver` deinstalliert werden. Da ein Treiber-Dienst nicht im Konsolen-Snap-In services.msc aufscheint, muss man ihn über den Registrierungsschlüssel

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RBACSys\` bearbeiten. Dort kann durch Setzen der Variable `Start` auf den Wert 2 ein Autostart für den Treiber definiert werden.

4.6.4 *RBACSystemDatabase* – Installation der Datenbank des RM

Zuerst muss MS SQL Server 2008 installiert werden. Dabei ist einfach strikt dem Installer von Microsoft zu folgen, als Name sollte *MSSQLSERVER* belassen werden. Ist die Installation abgeschlossen so muss noch eine Ausnahme für `c:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER \MSSQL\Binn\sqlservr.exe` in der Firewall des Betriebssystems definiert werden. (der Pfad der .exe-Datei des MSSQL-Service kann auch unter *Run* → *services.msc* eingesehen werden) Die SQL-Scripte müssen alle am Database-Server ausgeführt werden. Da Abhängigkeiten bestehen ist folgende Reihenfolge strikt einzuhalten:

1. RBACSystem
2. host
3. usr
4. perm
5. roles
6. u2r
7. p2r
8. session
9. checkAssigned
10. r2s
11. msid
12. v_effectPerm
13. v_rolesActive

14. v_u2p

An der Datenbank sind Konten für den Zugriff zu definieren. Dabei hat man die Wahl zwischen Windows Authentication und SQL-Server Authentication Accounts. Windows Authentication verwendetet Betriebsystem-Konten, während SQL-Server Authentication SQL Server spezifische Konten verwendet. Das Feature der Windows-Authentication wurde für den Protoypen nicht umgesetzt. Es sind daher SQL Server Konten zu spezifizieren, die den Zugriff auf die Datenbank regeln. Folgende Konten sind für *RBACSystem* empfehlenswert:

- *RBACSServiceAcc* Das Konto, das der Service verwendet, wenn er als Referenzmonitor agiert. Daher SELECT auf der View: *v_effectPerm*

- *RBACSMgmtAcc* Das Konto das die *RBACSystemServiceCtrl+Mgmt* Applikation zu administrativen Zwecken verwendet. SELECT, INSERT, DELETE auf den Tabellen *Host*, *Usr*, *Perm*, *Roles*, *u2r*, *p2r*, *msid* und SELECT auf *session*, *r2s*, *v_effectperm v_rolesActive v_permActive* und *v_u2p*.

- *RBACSClientCtrlAcc* Das Konto, das für die Session Verwaltung auf den Client-Computern eingesetzt wird, daher SELECT, INSERT, DELETE auf *session* und *r2s* und SELECT auf *u2r* und *v_rolesActive*.

4.6.5 *RBACSystemServiceCtrl+Mgmt* und *RBACSystemClientCtrl*

Diese Applikationen können ohne Installation direkt gestartet werden, wenn das .NET Framework 2.0 installiert ist. Die an SQL Server 2008 definierten Konten und Passwörter sind für den Dienst *RBACSystemMonitor* via *RBACSystemServiceCtrl+Mgmt* für die *RBACSystemServiceCtrl+Mgmt*-Applikation selbst und für die *RBACSystemClientCtrl*-Applikation einzutragen.

4.7 Testen des Systems

Das Test-Szenario beschreibt einen SMB-Zugriff eines Clientcomputers namens *DT-CLIENT* auf einen File-Share, der auf einem Server-Host namens *DT-FILESERVER* definiert ist. Das Netzwerk verfügt über keinerlei Domänenverwaltung. Wie in Abbildung 4.15 ersichtlich, verwaltet der Client seine Session, indem er die Datenbank modifiziert und greift via SMB auf den File-Share zu.

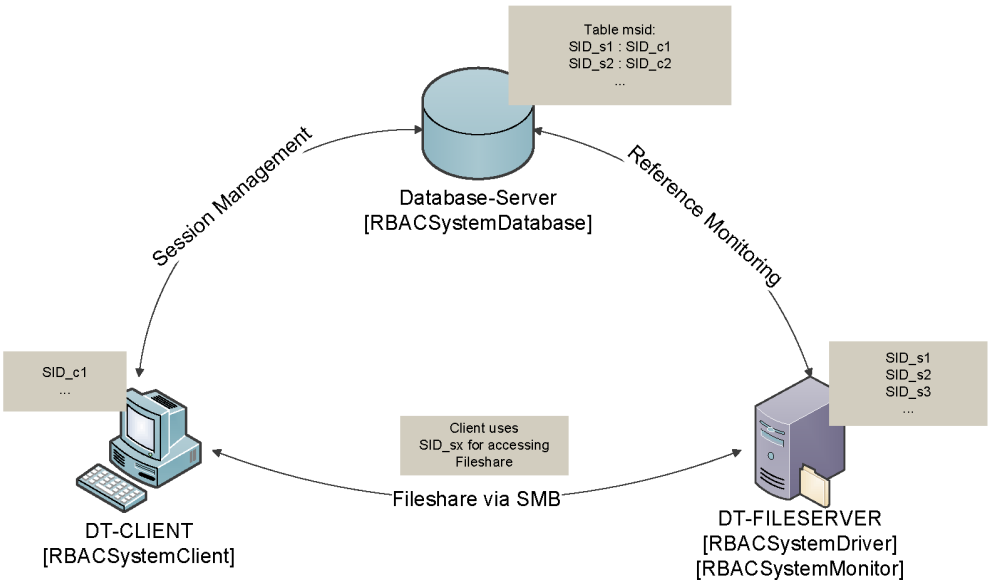


Abbildung 4.15: Test-Szenario

Auf *DT-FILESERVER* ist eine Windows Server 2008 Standard Edition (x86) installiert und auf *DT-CLIENT* läuft eine Version von Windows 7 (x86).

4.7.1 Installation RBAC-Core Package

Auf dem File-Server liegen bereits der *RBACSystemDriver*, die *RBACSystemImed.dll* und die GUI-Applikationen für die Installation bereit, wobei davon auf dem Server nur *RBACSystemServiceCtrl+Mgmt* benötigt wird, Abbildung 4.16.

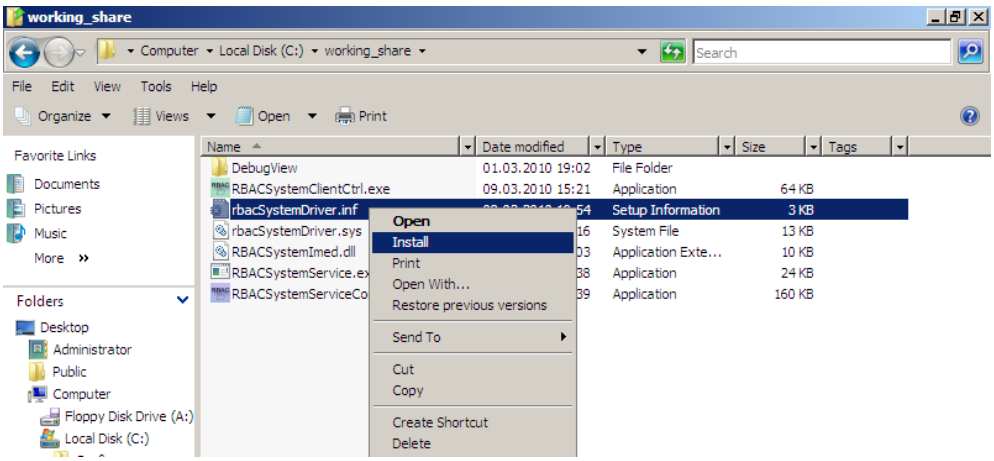


Abbildung 4.16: Installation des Treibers

Im ersten Schritt wird der Treiber installiert. Dies geschieht einfach mit einem Rechtsklick auf das .inf-File und im Kontextmenü unter der Auswahl von „Install“.

Im Registrierungseditor (*regedit*) kann der Eintrag für den Treiber eingesehen werden, Abbildung 4.17.

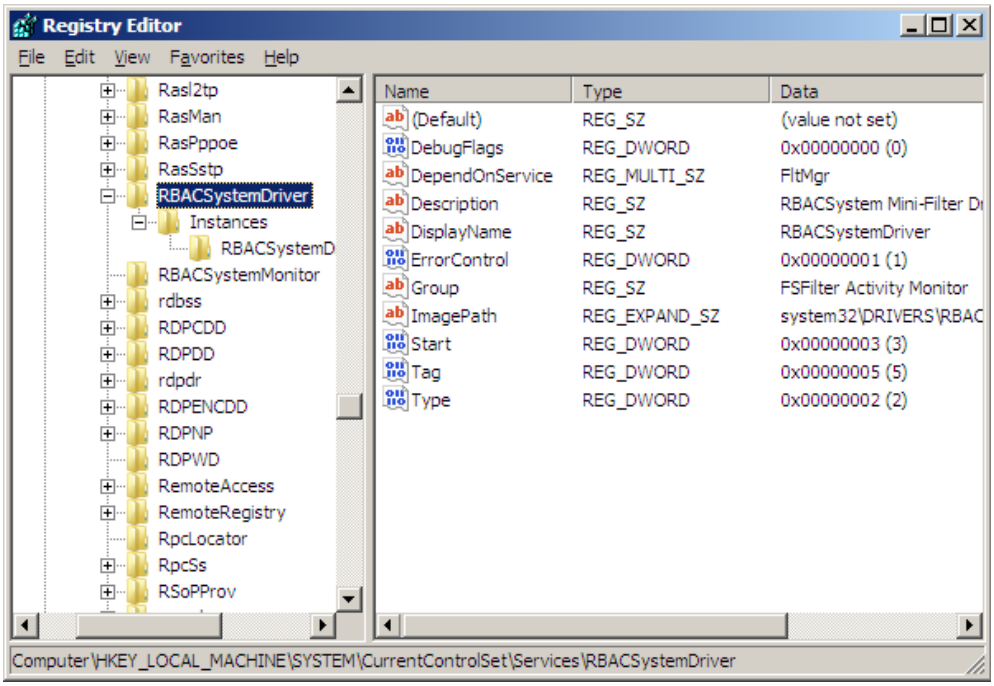


Abbildung 4.17: Registry-Einträge des Treibers

Für Testzwecke wurde der Starttyp auf 3, also manuellen Start festgelegt, kann jedoch für den Produktivbetrieb auf 2 – automatischen Start – festgelegt werden.

Die Installation für den *RBACSystemMonitor*-Service läuft folgendermaßen ab. Zuerst muss eine Shell geöffnet werden. In das Verzeichnis von *RBACSystemService.exe* navigieren und mit dem Befehl `installutil RBACSystemService` installieren, Abbildung 4.18.

Im Registrierungseditor können wieder die Details des Service eingesehen werden, Abbildung 4.19.

Die Serviceinstallation erzeugt Install- und Uninstall-Logs im Verzeichnis von *RBACSystemService.exe*, Abbildung 4.20.

Nun folgt noch die Installation der *RBACSystemImed.dll*, welche den Service und den Treiber verbindet. Einfach die .dll in das Verzeichnis *System32* von Windows ziehen, Abbildung 4.21.

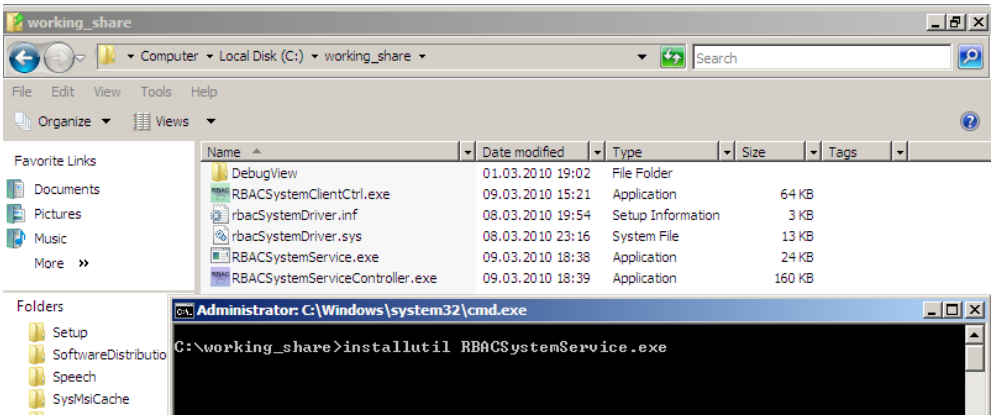


Abbildung 4.18: Installation des Service

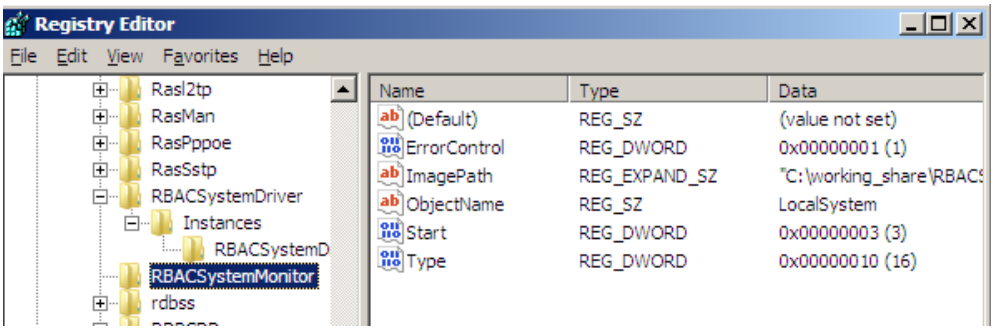


Abbildung 4.19: Registy-Einträge des Services

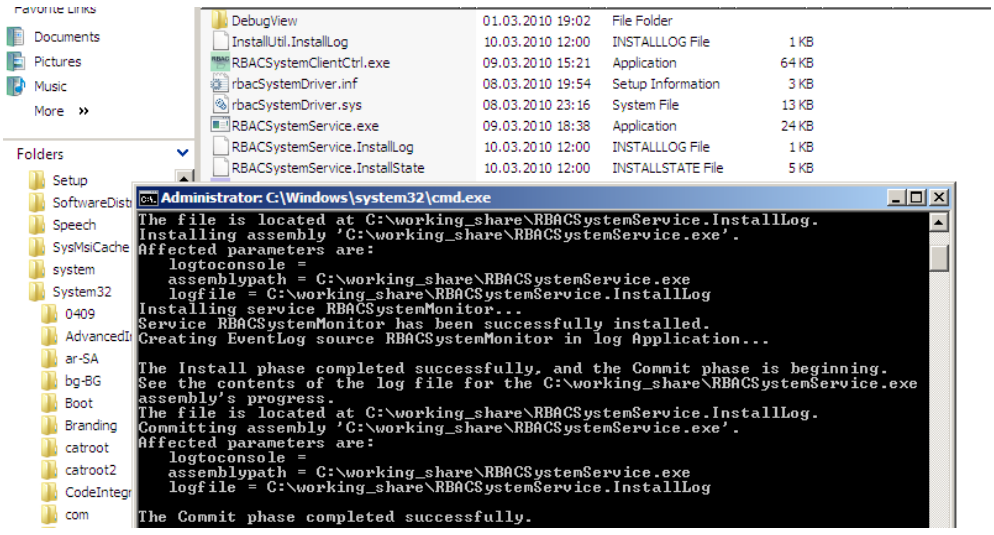


Abbildung 4.20: Service Installation Abschluss

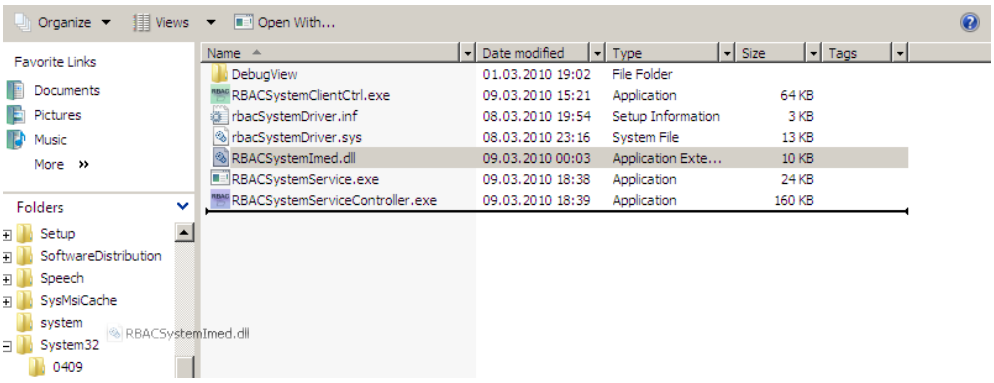


Abbildung 4.21: Installation RBACSystemImed.dll

Nun ist alles bereit für den ersten Start des Core-Packages Unter `sc start RBACSystemDriver` wird der Treiber gestartet, Abbildung 4.22. Es ist sicherzustellen, dass der Treiber zeitlich vor dem Service gestartet wird.

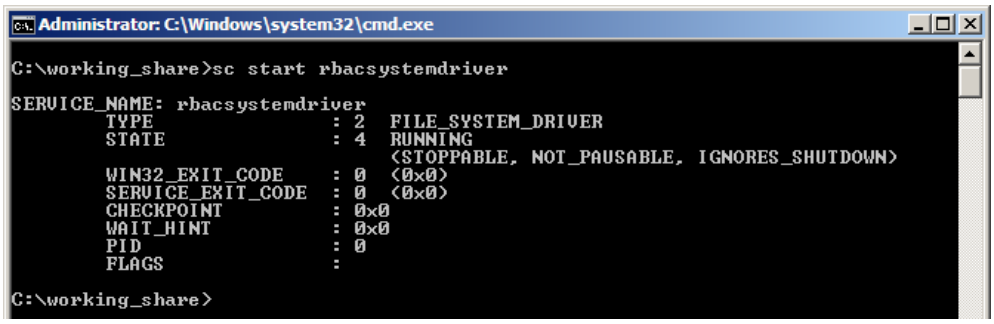


Abbildung 4.22: Start des Treibers

Danach kann der Service unter `net start RBACSystemMonitor` gestartet werden, Abbildung 4.23.

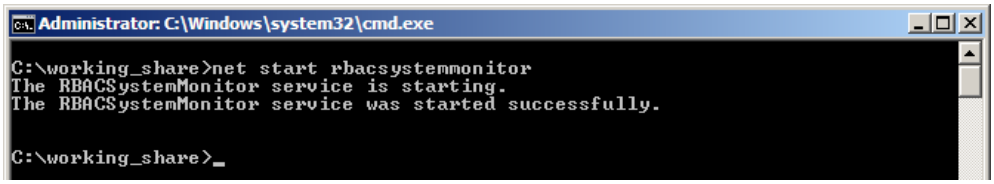


Abbildung 4.23: Start des Monitor-Services

Im Output von *DebugView* ist die initiale Kommunikation des Treibers mit dem Service deutlich ersichtlich, Abbildung 4.24.

#3 in Abbildung 4.24 zeigt an, dass ein Port-Objekt erfolgreich erzeugt wurde. #4 zeigt den Start des *RBACSystemMonitor*-Services an, der als erste Handlung die Callback-Funktion für die Kommunikation zwischen nativem und gemanagtem Code platziert.

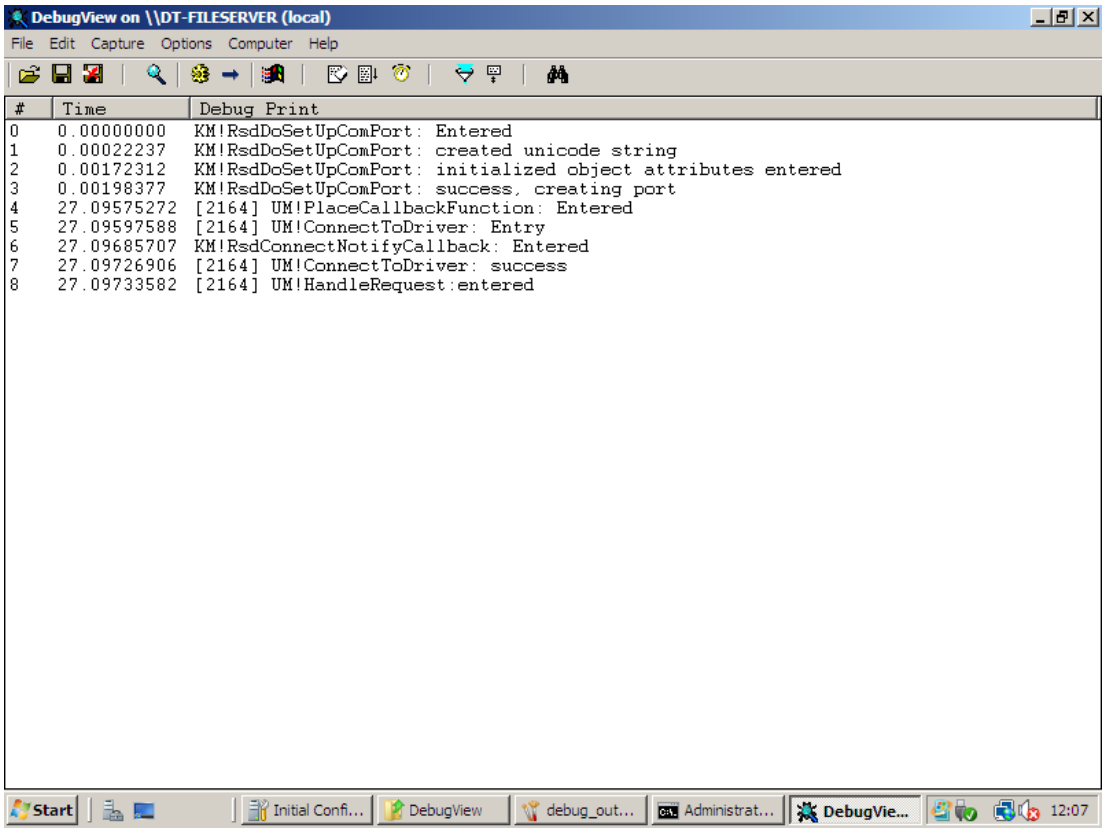


Abbildung 4.24: Initiale User/Kernel Mode Kommunikation

In #5 verbindet sich nun der *RBACSystemMonitor*-Service mittels *ConnectToDriver* über die *RBACSystemImed.dll* zum Treiber und verursacht in #6 somit den Aufruf der *RsdConnectNotifyCallback* Funktion auf Kernelseite. *ConnectToDriver* auf User-Mode Seite liefert somit den Status „Success“. Damit sind die Port-Objekte, über die die Kommunikation zwischen User-Mode und Kernel-Mode abläuft, ausgetauscht und es kann wie in #8 von *RBACSystemImed.dll* auf eingehende Nachrichten gelauscht werden, die sie dann an den *RBACSystemMonitor*-Service weiterleitet.

DT-FILESERVER ist somit unter dem Folder `c:\RBACroot` mit einem rollenbasiertem Zugriffsmodell ausgestattet. Dies wird deutlich, wenn man nun versucht, unter dem Windows Explorer diesen Ordner zu öffnen. Diese Operation schlägt nämlich fehl. Grund dafür ist, dass noch keine Rollen, Permissions und User definiert wurden.

4.7.2 User/Rollen/Permission Management

Im ersten Schritt wird, wie in Abbildung 4.25 dargestellt, ein User angelegt. Über die SID-Combo-Box kann ein Security-Identifier des Server Host, also in diesem Fall von *DT-FILESERVER* gewählt werden.

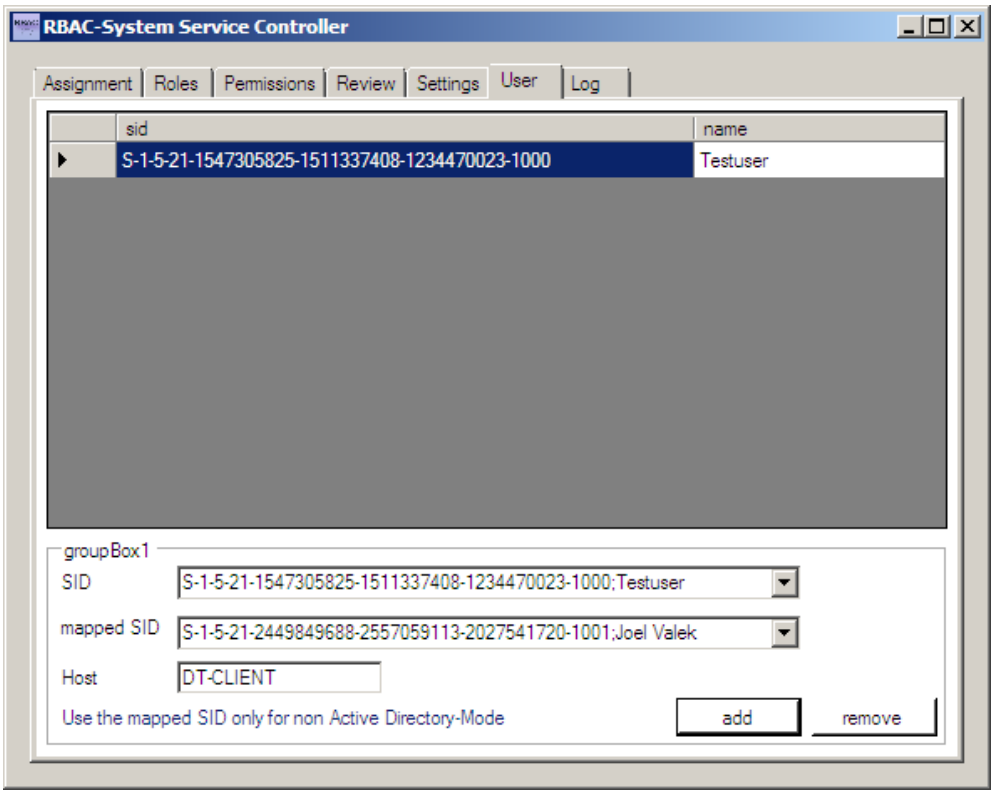


Abbildung 4.25: User anlegen

RBACSystem kann im Active Directory- oder Workgroup-Modus betrieben werden. Der wesentliche Unterschied zwischen den beiden Modi ist der Speicherort der Konten. Im AD sind alle Konten innerhalb eines Trees der Domänenstruktur über den GC erreichbar. Im Falle von Arbeitsgruppen residieren die Konten immer auf den Rechnern, auf denen sie eingerichtet wurden und es existiert keine zentrale Datenbank, über der die Konten erreichbar wären.

Das bedeutet dem Server-Host *DT-FILESERVER* und dem Client-Host *DT-CLIENT* sind die Konten auf der jeweilig anderen Maschine wechselseitig unbekannt. Zwar kann unter der Angabe des Host-Namens im Windows Netzwerk nach Konten auf anderen Maschinen gesucht werden, die NTFS-Berechtigung auf Objekte kann aber immer nur für die Konten vergeben werden, die auf demselben Host wie die betreffenden Objekte liegen und ein derartiges Konto wird auch für den Zugriff auf das Objekt über das Netzwerk verwendet.

Im Active Directory können über den zentralen Verzeichnisdienst auch Rechte für Konten auf Objekte vergeben werden, die nicht zwangsläufig auf derselben Maschine wie die Objekte residieren.

RBACSystem verwendet für den Workgroup-Modus eine Relation namens *msid*. Die Mapped SID Combobox bietet also beim Eintrag eines Users in die Datenbank die Möglichkeit, die SID eines Server-Kontos mit der SID eines Client-Kontos zu verbinden. Wichtig dabei ist, dass dies ein *1:n* Mapping von Client-Konten zu Server-Hosts und ein *1:1* Mapping von Client-Konten auf Server-Konten darstellen muss.

In diesem Falle wird der *Testuser*, der Maschine `DT-FILESERVER` auf *Joel Valek* der Maschine *DT-CLIENT* abgebildet, da der Workgroup-Modus betrieben wird.

RBACSystemclientCtrl liest beim Start die SID des gegenwärtig angemeldeten User aus. Sie sieht anschließend in der Tabelle *msid* nach, ob eine Abbildung für dieses Konto existiert. Tut sie das nicht, so befindet sich der Client im AD Modus. Das bedeutet der SID, den die Applikation ausgelesen hat, kann für die Session-Verwaltung verwendet werden und ist auch der gleiche, der für den Zugriff auf Server in der Domäne verwendet wird.

Sind in der Tabelle *msid* Abbildungen eingetragen, so wird dem Benutzer eine Auswahl der Server-Host Computer angeboten, denn sein SID kann auf n Server-Hosts für dort jeweils ein Konto gemappt sein. Der Benutzer trifft die Auswahl und die SID des Servers-Kontos wird anstatt der des Client-Kontos für die Sessionverwaltung verwendet. Unter dieser SID erfolgt auch der Zugriff auf die Server-Ressource wenn sich der Client anschließend über den Fileshare verbindet. Diese Thematik wird auch in Abbildung 4.15 ersichtlich.

Im nächsten Schritt müssen Permissions definiert werden. In diesem Fall wird eine Leseberechtigung auf das Wurzeverzeichnis `c:\RBACroot` und auf den darin enthaltenen Folder *a* definiert, Abbildung 4.26.

Damit dies geschehen kann, muss entweder das Core-Package angehalten werden (Service und Treiber stoppen) oder der *RBACSystemServiceCtrl+Mgmt* ausführende User mit Berechtigungen versehen werden, falls der Browse-Dialog verwendet wird, denn auch dies stellt einen lesenden Zugriff dar und würde ohne Berechtigungen geblockt werden. Zweite Möglichkeit, also einen User zu definieren, der rekursiven administrativen Zugriff auf `c:\RBACroot` besitzt, existiert derzeit in *RBACSystem* noch nicht. Das bedeutet, wenn neue Ordner oder Dateien innerhalb angelegt werden, so können Berechtigungen erst dann vergeben werden, wenn das Core-Package für Wartungsarbeiten angehalten wird. Eine weitere Möglichkeit wäre, *RBACSystemServiceCtrl+Mgmt* unter *Local System* zu starten, denn für dieses Konto wurde, wie bereits erwähnt, eine Ausnahme im Treiber definiert, damit *RBACSystemMonitor* seinen Dienst erfüllen kann. Damit dies funktioniert müsste allerdings die Logik von *RBACSystemServiceC-*

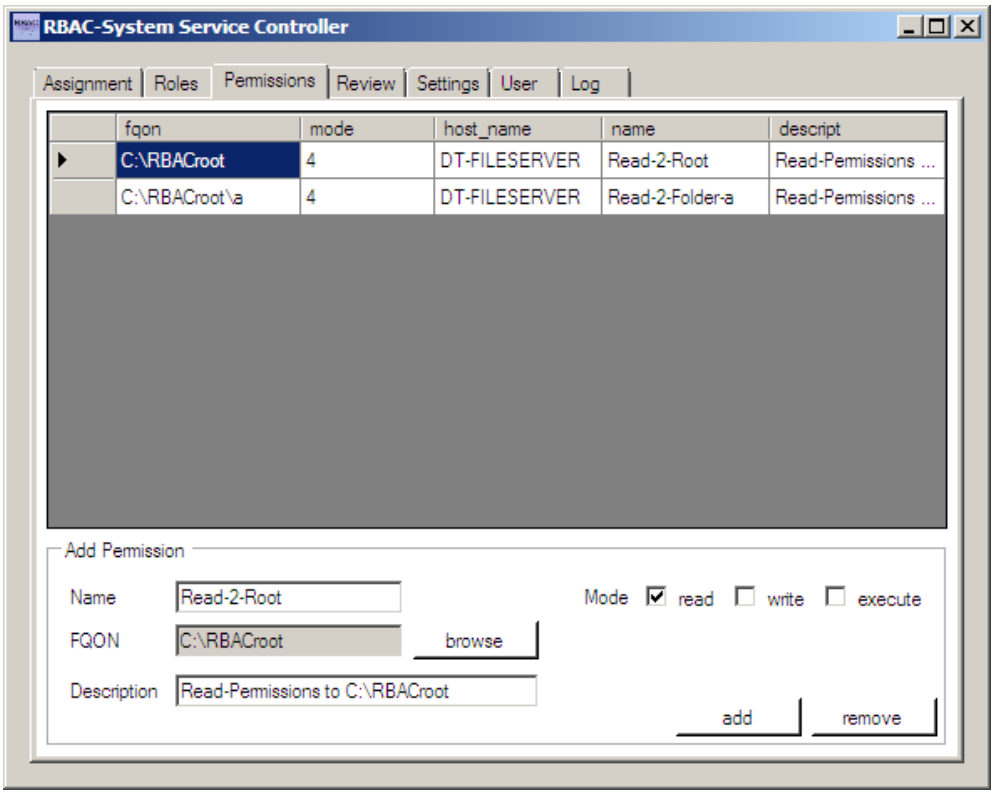


Abbildung 4.26: Permission anlegen

trl+Mgmt in einen Dienst ausgelagert werden, denn eine Windows-Forms Anwendung kann nicht unter *Local System* ausgeführt werden.

Der dritte Schritt, bevor die Zuweisung erfolgen kann, ist die Definition einer Rolle. In diesem Beispiel wird die Rolle „Test-Role-FILESERVER“ angelegt, Abbildung 4.27.

Die User/Rolle/Permission-Zuweisung erfolgt im Assignment Tab. Hier sind einfach die Benutzer und Permissions auszuwählen, die einer Rolle zugeordnet werden sollen, Abbildungen 4.28 und 4.29.

Nun erfolgt zur Kontrolle noch ein Blick in das Review-Tab, ob der Benutzer auch alle Rollen und die damit verbundenen Permissions erhalten hat, Abbildung 4.30.

Dadurch, dass die Rollen und Permissions nicht angehakt sind, erkennt man, dass der Testuser nicht in einer Session mit diesen aktiv tätig ist.

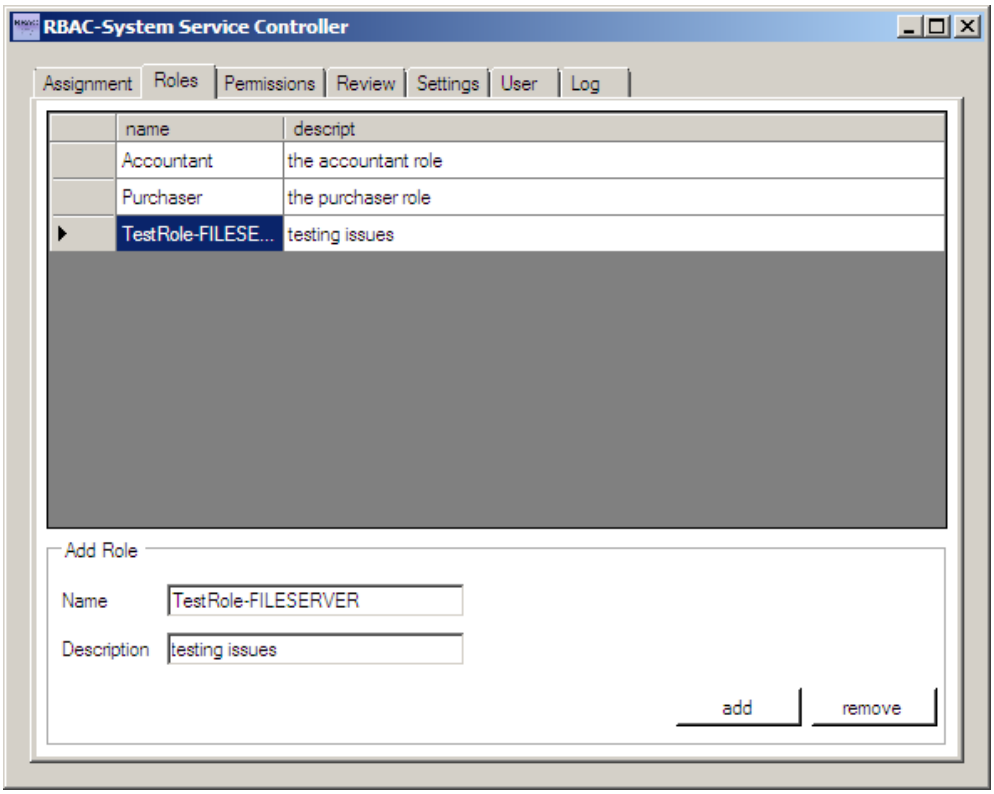


Abbildung 4.27: Rolle anlegen

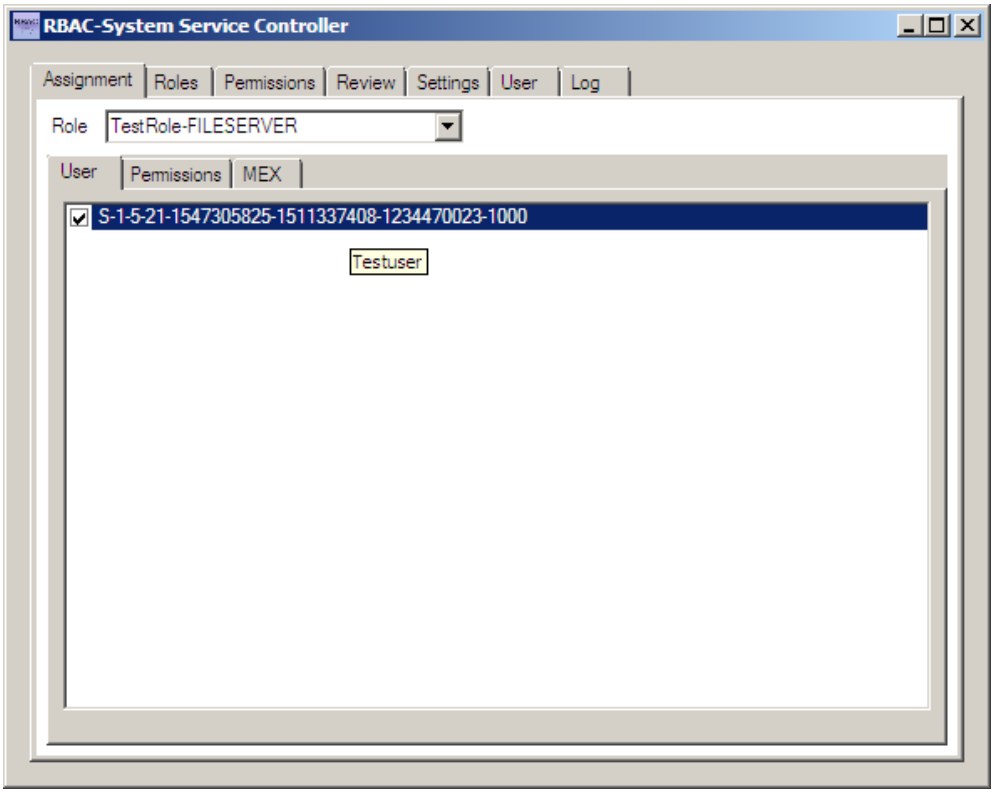


Abbildung 4.28: User zuweisen

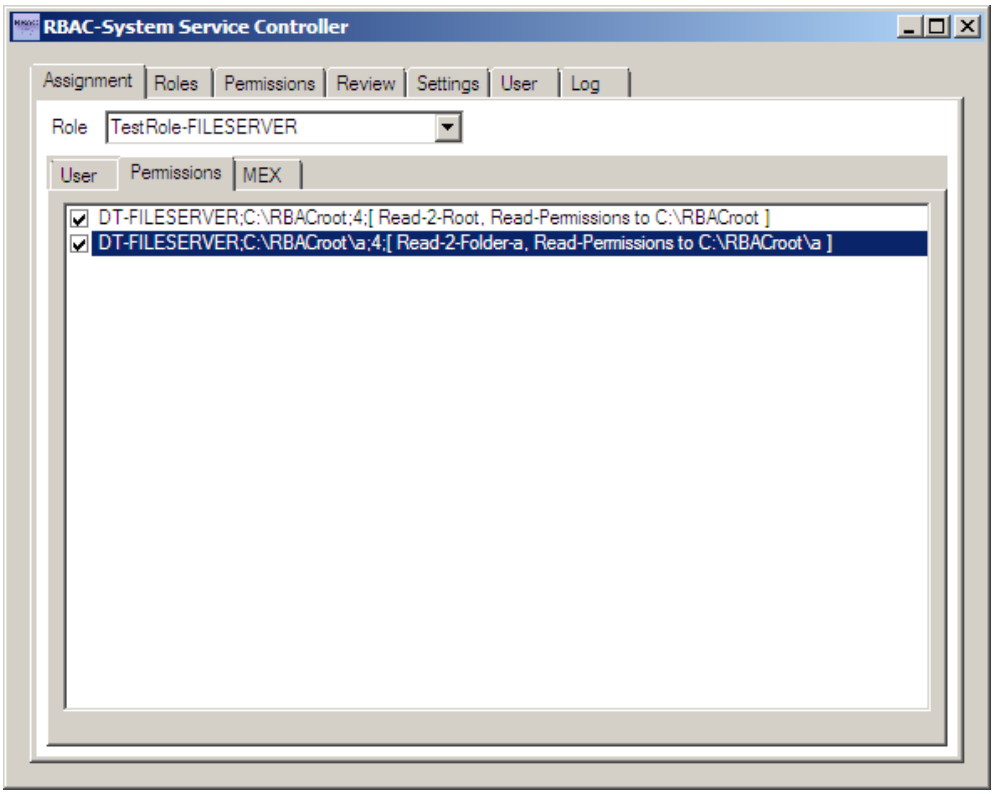


Abbildung 4.29: Permission zuweisen

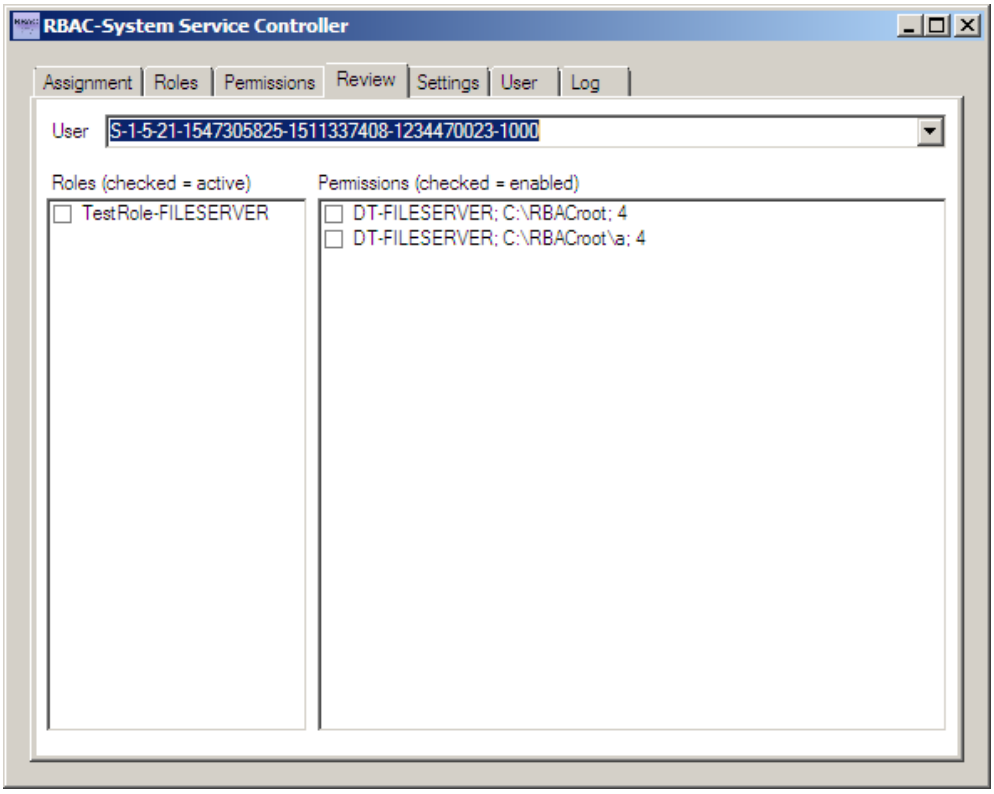


Abbildung 4.30: Review der User-Zuweisung

4.7.3 Zugriff auf RBACroot

Nun ist es Zeit, einen Zugriff auf `c:\RBACroot` durchzuführen. Bewerkstelligt wird dies über einen File-Share, welcher auf *DT-FILESERVER* definiert wird, Abbildung 4.31.

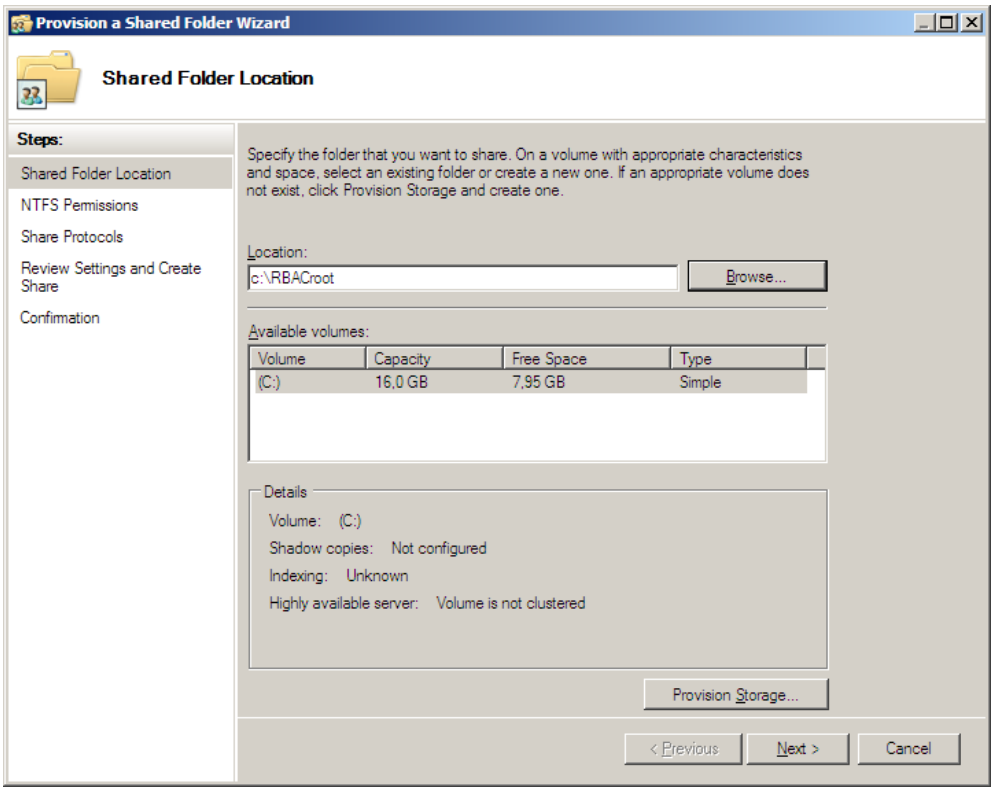


Abbildung 4.31: Einrichten des Fileshares

Wichtig in diesem Zusammenhang ist, dass *Testuser* sowohl Full Control über den Share als auch in den NTFS Permissions eingeräumt wird. Der Share stellt im Normalbetrieb ohne *RBACSystem* so etwas wie einen Vorfilter dar, den die User passieren müssen. Ist beispielsweise lediglich *Read* am Share für einen User definiert, so sind die dahinter definierten, über eine Leseberechtigung hinaus gehenden NTFS-Permissions obsolet. Im Klartext bedeutet dies, dass die User den Share überwinden müssen, damit überhaupt NTFS Permissions wirken können. Analog verhält es sich nun zwischen *RBACSystem* und Share+NTFS. Wenn in den Share- oder NTFS-Permissions geringere Rechte definiert sind, als die Rolle zuweist, so wirken die Rechte der Rolle unvollständig bis gar nicht, Abbildung 4.32.

Die Filterreihenfolge lautet also $Share - Permissions \rightarrow NTFS - Permissions \rightarrow RBAC - Permissions$.

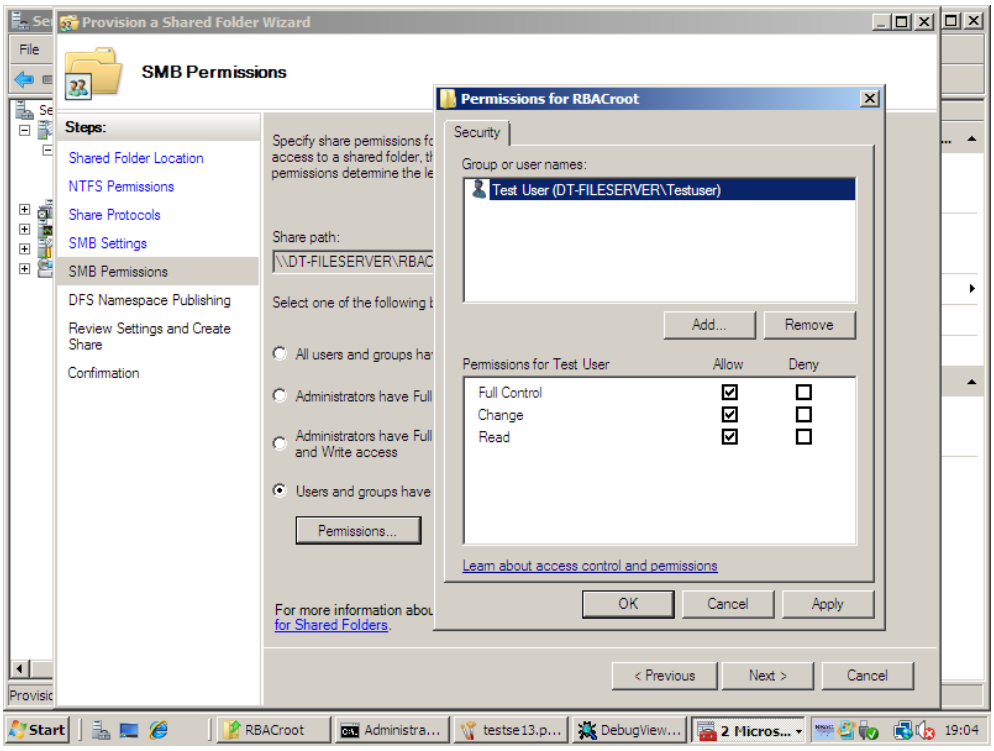


Abbildung 4.32: Share-Permissions

Um die volle Rechtedynamik von *RBACSystem* auszunützen, müssen also die NTFS Permissions und der Share auf ihre Maximalberechtigungen gesetzt werden, Abbildung 4.33. Die tatsächliche Rechtevergabe erfolgt dann ausschließlich über die User/Rollen/Permission-Zuweisung in *RBACSystemServiceCtrl+Mgmt*.

Auf der Maschine *DT-CLIENT* wird nun die *RBACSystemClientCtrl* Applikation gestartet, Abbildung 4.34. Wie zuvor erwähnt, erscheint das Auswahlfenster für den Host Computer - In diesem Fall *DT-FILESERVER*, denn mit dem dort liegenden Konto *Testuser* ist der User *Joel Valek* von *DT-CLIENT* verlinkt.

Ersichtlich wird dies nur über die Datenbank in der Tabelle *msid*, Abbildung 4.35.

Mit einem Rechtsklick auf die Session Management Baumansicht können neue Sessions eröffnet und geschlossen, Rollen aktiviert und deaktiviert werden, Abbildung 4.36.

In diesem Fall wird einfach eine neue Session mit der Rolle „TestRole-FILESERVER“ eröffnet, Abbildung 4.37.

Abbildung 4.38 zeigt den Client mit der soeben erstellten Session-Nummer 29 und der darin aktiven Rolle „TestRole-FILESERVER“.

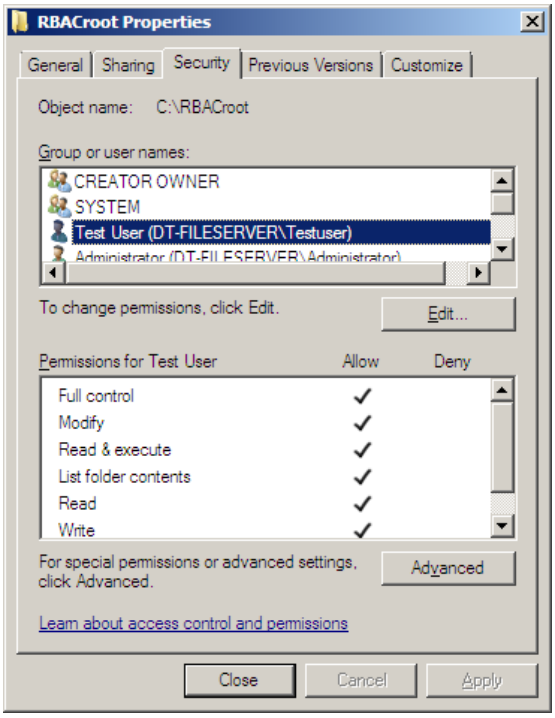


Abbildung 4.33: NTFS-Permissions

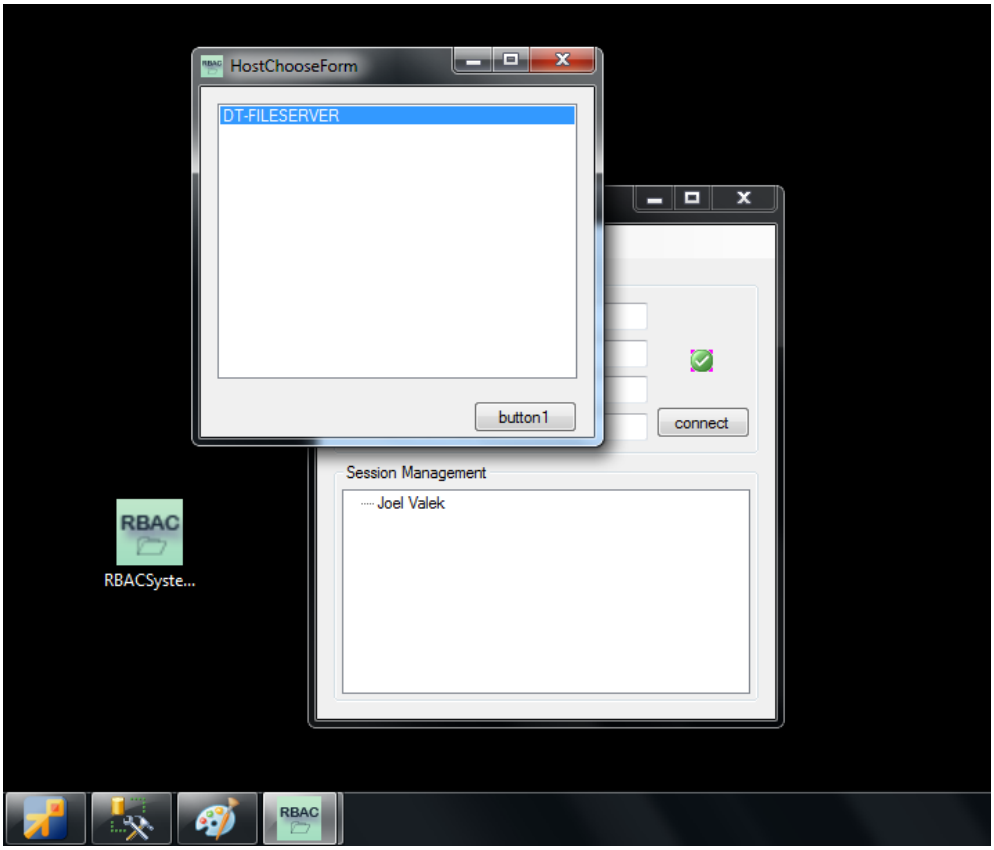


Abbildung 4.34: Start RBACSystemClientCtrl

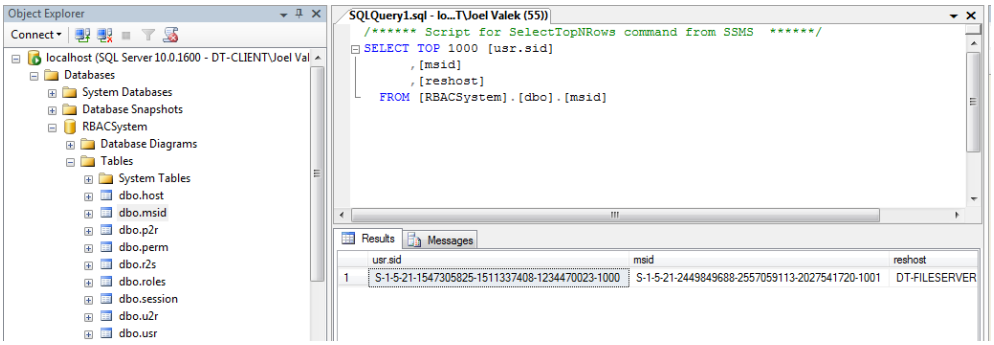


Abbildung 4.35: Datenbankeinträge des Usermappings

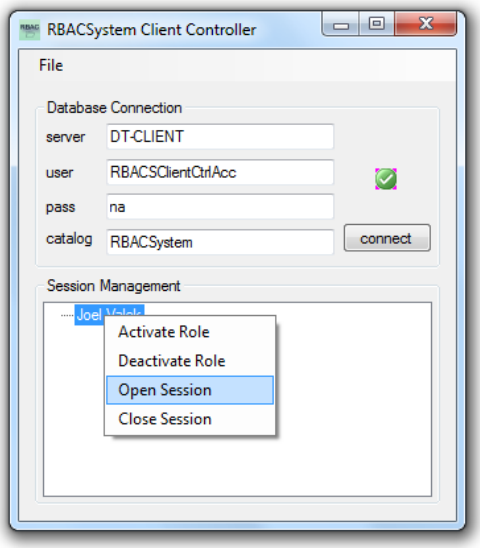


Abbildung 4.36: Session eröffnen

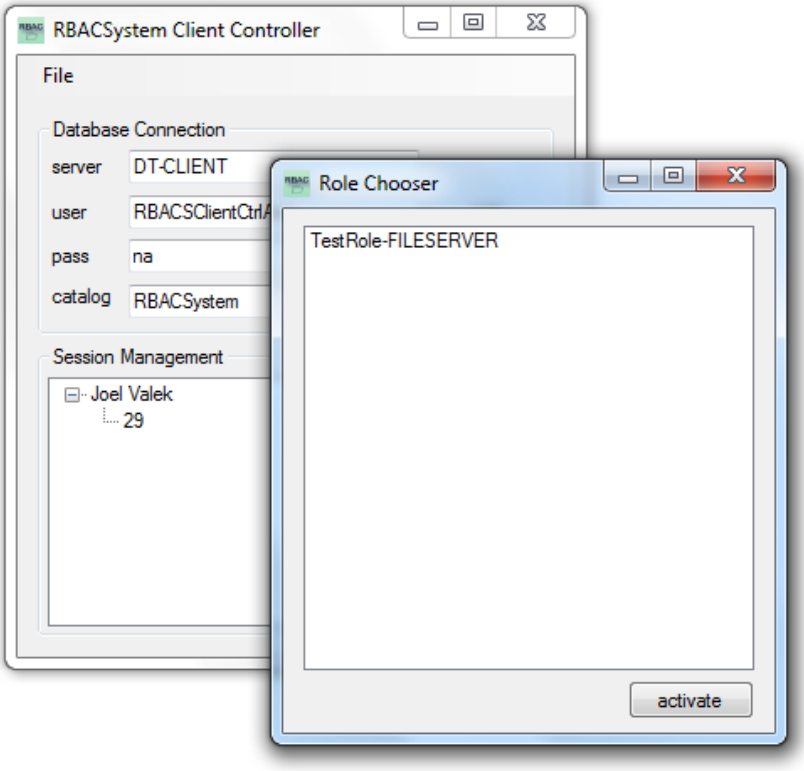


Abbildung 4.37: Rolle auswählen

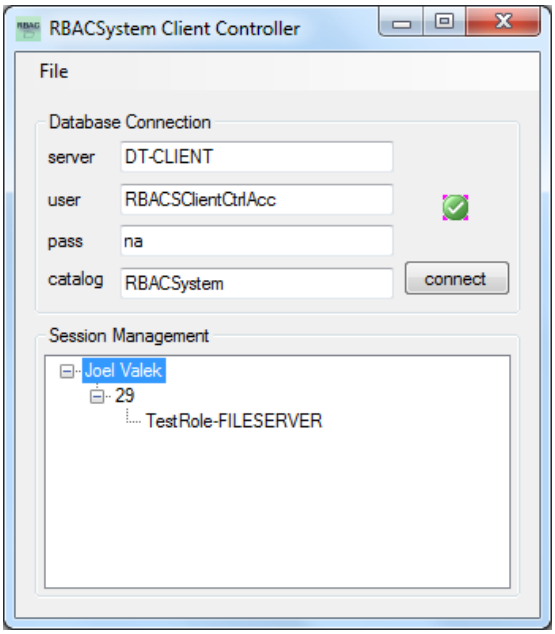


Abbildung 4.38: Client mit aktivierter Session und Rolle

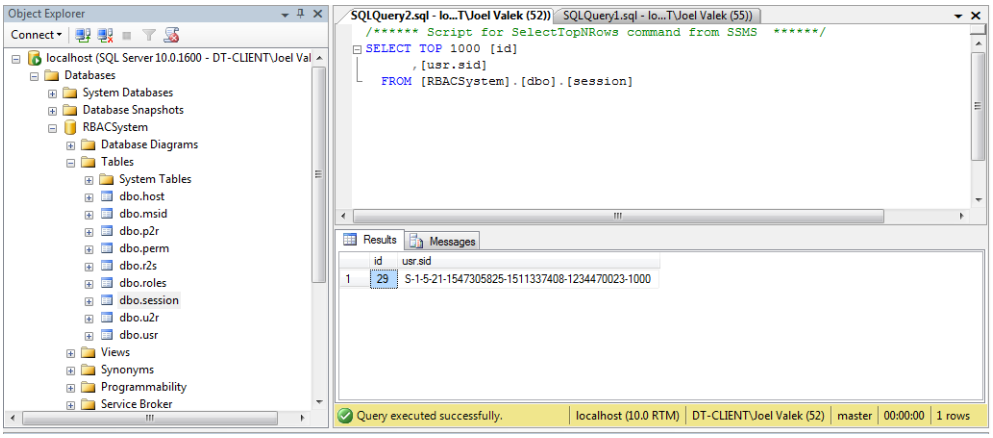


Abbildung 4.39: Session-Eintrag in der Datenbank

Ein Einblick in die Datenbank der Tabelle *session* zeigt die gerade erzeugte User-Session, Abbildung 4.39.

Nun steht dem Zugriff über den Share auf `c:\RBACroot` nichts mehr im Wege.

Da bereits für Datei-Kopier-Zwecke ein Share namens *working_share* benutzt wurde und hierfür ein anderes Konto als *Testuser* verwendet wurde, so muss zuerst die SMB-Session geschlossen und der Benutzer geändert werden. Die Schließung der SMB-Session kann entweder über den Timeout abgewartet werden, oder man greift manuell ein. Serverseitig schließt man die Session am besten mit einem Aufruf von `fsmgmt.msc` und terminiert dort die Session. Clientseitig tätigt man in einer Shell den Aufruf `net use * /delete` und anschließend definiert man das zu verwendende Konto über net `use \\<servername> <passwort> /USER:<username>`, Abbildung 4.40.

Abbildung 4.40: Ändern des Share-Zugriffs-Kontos

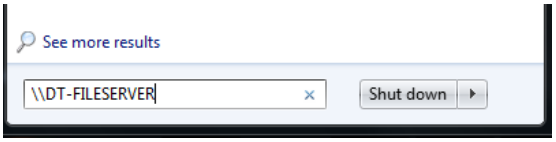


Abbildung 4.41: Share-Zugriff

Unter *Start* → *Run* wird der Share über den UNC String aufgerufen, Abbildungen 4.41 und 4.42.

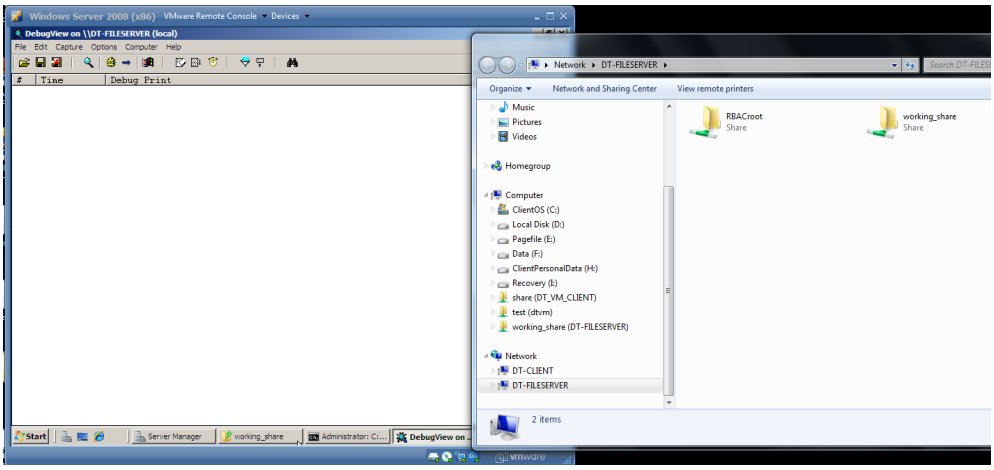


Abbildung 4.42: Auswahl der Shared Folder

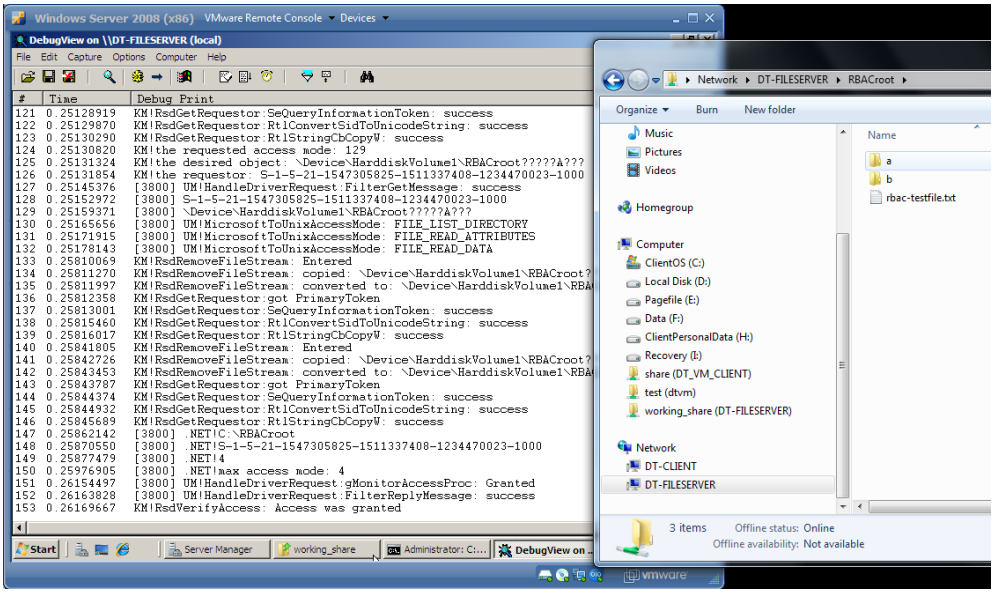


Abbildung 4.43: Zugriff auf RBACroot

Der Zugriff auf `c:\RBACroot` funktioniert einwandfrei. Der Requestor-SID entspricht *Testuser*. Der Zugriffsmode ist 4, entspricht also einem Read und auch der Full Qualified Object Name stimmt mit dem Target-Object überein, Abbildung 4.43.

Analog verhält es sich mit dem Folder *a* unter `c:\RBACroot`. An den Fragezeichen, die am Kernel-Mode-Nomalized-Path angehängt sind – z.B Eintrag #471, erkennt man nun die Problematik der eingangs erwähnten Surrogate Characters und an den Zeichen

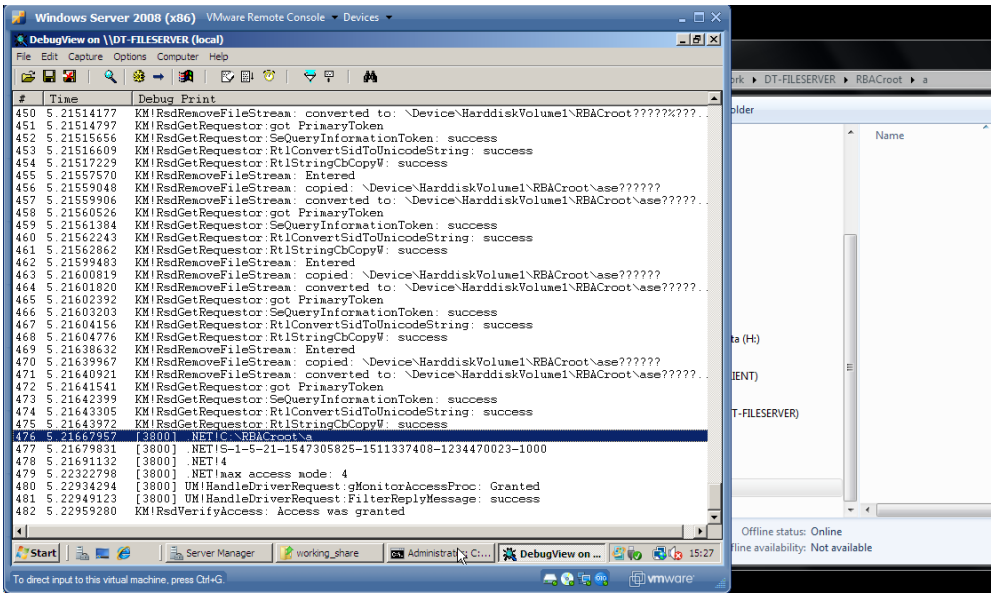


Abbildung 4.44: Zugriff auf Folder a

„s“ und „e“, die an dem Folder *a* angehängt sind, die Notwendigkeit der erwähnten *pathChecking* Funktion, Abbildung 4.44.

Auf Folder *b* wird wie gewünscht kein Zugriff gewährt und der Explorer quittiert den Zugriffsversuch mit einer Message Box, Abbildung 4.45.

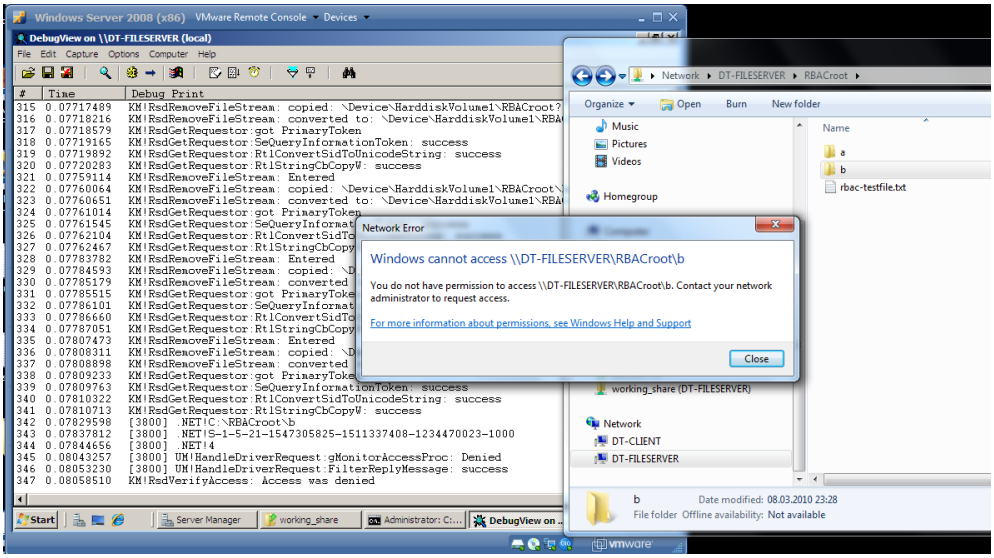


Abbildung 4.45: Zugriff auf Folder b

4.8 Sicherheitsüberlegungen

Da ein Zugriffskontrollsystem bzw. der RM Teil der Trusted Computing Base ist, ist es von besonderer Bedeutung, die Schwachstellen und Angriffspunkte zu mitigieren oder sie zumindest aufzudecken und sich darüber bewusst zu werden.

Ein mögliches Angriffszenario auf das System könnte von einem der Client-Computer aus statt finden, wenn Benutzer versuchen, auf Ressourcen zu zugreifen, für die sie keine Berechtigung haben. Ein besonderes Augenmerk liegt also auf *RBACSystemClientCtrl*. Das Konto, das die Applikation für die Verbindung zur Datenbank aufnimmt ist für alle Clients gleich. Eine Verschlüsselung des zu übertragenden Passwortes macht daher keinen Sinn, da es ohnehin öffentlich bekannt ist. Auch die Kommunikation, die zwischen dem Client und dem Datenbankserver abläuft, stellt kein Geheimnis dar – es werden ja lediglich Rollen aktiviert, bzw. desaktiviert. Die Gefahr, dass ein Benutzer mit einem manipulierten SQL-String Rollen aktiviert, die er gar nicht besitzt, wurde mit dem Contraint auf der Tabelle s2r aus der Welt geschafft, vorausgesetzt natürlich, dass das Konto *RBACSMgmtAcc* nur Administratoren vorbehalten bleibt.

Ein Benutzer könnte der Datenbank einen gespooften SID vorspiegeln und dann damit agieren. Dies wäre zwar theoretisch möglich, hat aber für den Angreifer keinerlei konstruktive Bedeutung. Den Token des Prozesses, ergo den SID, unter dem der Windows-Fileshare-Zugriff statt findet, zu tampern, ist nämlich wesentlich schwieriger. Doch dies wäre notwendig, wenn die Attacke sinnhaft sein soll. Der einzige Nachteil, der sich also daraus ergibt, ist destruktiver Natur. So könnte er bei Kenntnis des SID eines Users die Session des selbigen beeinflussen, Rollen darin aktivieren oder desaktivieren, während der Benutzer gerade arbeitet.

Ein zweiter nicht zu unterschätzender Angriffspunkt betrifft das Core-Package. Da auf `c:\RBACroot` die Share- als auch die NTFS-Permissions vollen Zugang gewähren, wäre dieser Bereich des Dateisystems im Falle eines Ausfalls des gesamten Core-Packages für jeden Benutzer unlimitiert zugängig. Hier muss die Ausfallswahrscheinlickkeit der Komponenten im Detail betrachtet werden.

Gelingt es einem Angreifer den *RBACSystemService* – beispielsweise mit einer DoS Attacke – außer Betrieb zu setzen, so ist immer noch *RBACSystemDriver* aktiv. In diesem Fall funktioniert der RM nicht mehr und der Zugriff auf alle Objekte in `c:\RBACroot` wird ausnahmslos verweigert.

Kann ein Absturz von *RBACSystemDriver* provoziert werden, so ist `c:\RBACroot` schutzlos. Daher ist es von höchster Wichtigkeit, dass diese Komponente zuverlässig funktioniert. Für ein kommerzielles System wäre es angebracht, die Korrektheit des

Treibers verifizieren bzw. ihn von Microsoft signieren zu lassen. Der Treiber bietet verschiedene Ansatzpunkte für Angriffe. Die Kommunikation erfolgt immer aus der Initiative des Treibers heraus. Auf eine seitens des User-Mode initiierte Kommunikation reagiert *RBACSystemDriver* nicht. Somit kann über die Port Kommunikation nicht unter Stress gesetzt und eine DoS Attacke provoziert werden. Allerdings sind mutiple Dateizugriffe in kürzester Zeit und somit eine Provokation des Überlaufes der Port-Message-Queue durchaus denkbar, denn für jeden Request sendet der Treiber eine Message an den User-Mode, die erst dann als erledigt gilt, wenn er eine Antwort erhalten hat, bzw. das Timeout abgelaufen ist.

Gegen einen Bufferoverflow muss *RBACSystemDriver* nicht explizit geschützt werden. Die einzige Schnittstelle die Daten entgegen nimmt, sind wieder die Port-Objekte und die zugehörigen Funktionen entstammen der Library von Microsoft – es kann bzw. muss ihnen also vertraut werden. Vorraussetzung für einen zuverlässigen Betrieb des Core-Packages ist, dass die Sicherheitskonfiguration des Servers keine Lücken aufweist.

Der dritte Angriffspunkt ist die Datenbank selbst. Der Server, auf dem sie läuft, als auch sie selbst muss daher ebenfalls eine sichere Konfiguration aufweisen. Die Konten für Management-Aufgaben als auch für den Service sind mit starken Passwörtern zu schützen. Sie sollten nach den Berechtigungs-Vorlagen, wie sie bei der Installations-Beschreibung von *RBACSystemDatabase* definiert wurden, erstellt und verwendet werden. SQL-Injections sind über *RBACSystemClientCtrl* nicht möglich, weil die Applikation vom Benutzer keine String-Eingaben entgegen nimmt. Weiters weist das Konto *RBACSClientCtrlAcc* – falls gemäß der erwähnten Vorlage definiert – derart minimale Rechte auf, sodass es Kommandos wie *CREATE*, *DELETE* oder *DROP* ohnehin nicht ausführen kann. Sollte die Datenbank auf einen anderen physikalischen Host als der File-Server untergebracht sein, so muss auch die Kommunikation zwischen *RBACSystemMonitor* und *RBACSystemDatabase* verschlüsselt werden, damit die Integrität des Resultates sichergestellt ist.

Vorsicht ist geboten, falls Services am System installiert sind, die unter dem *Local System Account* laufen, nach dem Trusted Subsystem Model aufgebaut sind und Benutzern einen Dateizugriff anbieten, denn das Konto *Local System* ist in *RBACSystemDriver* von einer Prüfung ausgeschlossen und kann somit ungehindert auf `c:\RBACroot` zugreifen.

4.9 Fazit

Wie in der Aufgabestellung angedacht, ist es mir gelungen, ein RBAC-System auf Windows 7 und Server 2008 umzusetzen. Das realisierte System repräsentiert die Funktionalität eines RBAC0 Modells nach Sandhu et al. [13].

Die Frage, ob das System wie vorhergesehen auf Betriebssystem-Ebene agiert oder doch nur eine Applilkation ist, die darauf läuft, ist schwer zu klären bzw. eine philosophische. Da der Ansatzpunkt von *RBACSystem* im OS-Kernel liegt, modifiziert es die NTFS- Zugriffskontrolle, also das Betriebsystem. Allerdings kann der DAC Mechanismus nicht eliminiert werden – RBAC baut gewissermaßen darauf auf. Eine Eliminierung wäre nur dann möglich, wenn der Quellcode von MS offen liegen würde und das DAC-System durch ein RBAC-System vollständig ausgetauscht werden könnte. Trotzdem das DAC im OS verbleibt, verhält sich das Zugriffskontrollsystem für alle Applikationen und Programme, die darauf laufen sowie User, die mit dem OS interagieren und auf Objekte, die in `c:\RBACroot` liegen wie ein rollenbasiertes. Das bedeutet jegliche Gruppenmitgliedschaften eines Users, die in DAC von Bedeutung sind, sind an diesem Ort obsolet. Die NTFS Permissions werden für diesen Ordner und alle darin enthaltenen Unterordner im Dateisystem außer Kraft gesetzt. Da der Sourcecode nicht offen liegt und somit auch nicht verändert und neu kompiliert werden kann, bleibt hier nur der Umweg, über ein „Full Control“ für die Gruppe „Everyone“ in den NTFS Permissions. Was zunächst wie eine ernsthafte Sicherheitslücke aussieht, wird durch *RBACSystem* abgesichert. Die bereits erwähnte Kette $Share - Permissions \rightarrow NTFS - Permissions \rightarrow RBAC - Permissions$ macht die Analogie deutlich. Auch wenn die Share-Permissions *Full Control* erlauben, so sind in einem System ohne *RBACSystem* noch die NTFS-Permissions dahinter vorhanden. Ist *RBACSystem* auf dem OS installiert, so können Share als auch NTFS Permissions *Full Control* erlauben, *RBACSystem* sitzt als letzte Instanz in der Kette dahinter und sichert ab.

Sieht man sich nun die Umsetzung auf Basis der Motivation an, so erkennt man, dass das tatsächliche Zugriffskontrollsystem wie gewünscht austauschbar ist. Lediglich die Datenbasis, Server- und Client-Management Applikationen müssen angepasst werden (*RBACSystemDatabase*, *RBACSystemServiceCtrl+Mgmt*, *RBACSystemClientCtrl*). So wäre also auch ein MAC System ohne Probleme realisierbar. Auch wird dem User keinerlei Kontrolle über die Rechteverwaltung erteilt, was bei einem Gruppen bzw. OU basiertem Konzept notwendig gewesen wäre.

Im Zuge der Durchsicht der Diplomarbeit und der Diskussion mit Herr DI Praher ergab sich folgende Fragestellung, auf die ich noch näher eingehen möchte.

Da das Beispiel, mit welchem das *RBACSystem* getestet wurde, einen Fileshare via SMB darstellt, kann der Eindruck erweckt werden, dass dies das einzige Anwendungs-Szenario von RBACSystem sei und ein solches Verhalten auch über einen WebDAV- oder FTP-Server mit entsprechendem rollenbasiertem Zugriff realisiert werden könnte. Für einen einfachen Fileshare unter Windows stimmt dies im Wesentlichen, jedoch existieren zwei zentrale Unterschiede.

Der größte Unterschied liegt darin begründet, dass ein WebDAV-Service auf dem OS aufsetzt, also kein Standard-Konzept (wie beispielsweise der Explorer) von Microsoft Windows ist, sondern nachgerüstet werden muss. So kann beispielsweise über den IIS, der wiederum ebenfalls vorab installiert werden muss, ein WebDAV-Service installiert und gestartet werden. Ein WebDAV Server mit RBAC wäre also ein klassisches Beispiel für die Weiterentwicklung eines der zahlreichen Open Source-Projekte, das einen rollenbasierten Zugriff umsetzt, dabei den Authorization Manager von MS verwendet und unter dem Trusted Subsystem Model agiert. Der WebDAV-Server könnte also Windows Benutzer für den Authentifizierungs- und Autorisierungsvorgang verwenden und die zugehörige Zugriffskontrollstruktur im Authorization Manger verwalten. Wesentlicher Unterschied wäre hier also wieder, dass der WebDAV-Service über DAC in NTFS mindestens die Rechte besitzen muss, die er seinen Usern über RBAC zur Verfügung stellt, man das Konto, über das der WebDAV-Service agiert – und sei dies lediglich Local System – wieder mit entsprechenden Permissions in den ACLs der Objekte eintragen müsste.

Somit wäre zwar über ein solches System der Zugriff für den User auf den mittels WebDAV abgesicherten und zur Verfügung gestellten Folder `c:\RBACroot` rollenbasiert. Andere Applikationen bzw. Services, die auf dem selben Server-Host wie der WebDAV-Service laufen und auf Objekte, die in `c:\RBACroot` liegen, zugreifen, können dies aber nur unter rollenbasiertem Verhalten tun, indem sie ebenfalls den Umweg über die WebDAV- Schnittstelle nehmen. Hier wird der zweite Unterschied deutlich. Applikationen verwenden die Standard-API Funktionen, die ihnen das OS für den Dateizugriff zur Verfügung stellt. Ein Umweg über WebDAV wäre nicht nur ineffizient, sondern für viele Applikationen schlichtweg unmöglich – vor allem dann nicht, wenn der Quellcode nicht offen liegt und somit die Dateizugriffsfunktionen nicht verändert werden können. Ausserdem könnte die rollebasierte Absicherung auf diese Weise leicht unterlaufen werden, indem ein Service bzw. Applikation auf `c:\RBACroot` eben nicht unter Verwendung der WebDAV Schnittstelle zugreift und einem Benutzer den Zugriff darauf ohne RBAC ermöglicht.

Mit *RBACSystem* ist also der rollenbasierte Zugriff eines Users auf Objekte nicht nur über SMB möglich sondern auch über WebDAV, FTP, HTTP und alle anderen erdenklichen Services und Applikationen, die auf dem Server-Host laufen, weil diese Standard-

OS-API Zugriffsfunktionen verwenden, welche immer einen *IRP_MJ_CREATE*-Request verursachen, der vom RM abgefangen wird. Vorraussetzung dafür ist dann allerdings, dass diese Services bzw. Applikationen nicht unter dem Trusted Subsystem Model sondern unter dem Impersonation Model laufen, den gegenwärtig zugreifenden User daher über eine Named-Pipe impersonalisieren. Denn ein User kann nur seine eigene Session modifizieren, nicht jedoch die der Services, falls diese überhaupt unter rollenbasierter Verwaltung stehen, was beim Impersonation Model nicht zwingend notwendig ist. Sollte die Applikation unter dem Trusted Subsystem Model aufgebaut sein, so ist ihre Zuweisung zu einer oder mehreren Rollen in RBACSystem notwendig, damit sie auf `c:\RBACroot` Zugriff erhält. Auch muss sich die Applikation um ihre eigene Session-Verwaltung kümmern, die selbe Schnittstelle wie *RBACSystemClientCtrl* zur Datenbank also nutzen und ihre Rollen aktivieren.

RBACSystem ist längst noch nicht perfekt und bietet ein weites Betätigungsfeld für den Ausbau. So kann die Permissions-Adressierungsflexibilität erhöht werden. Gegenwärtig beziehen sich Permissions auf den Pfad eines Objektes im Dateisystem. Wie bereits erwähnt, bieten auf Attribute basierende Permissions wesentlich mehr Flexibilität und erleichtern die Administrationsarbeit.

Auch existiert noch kein Adminstratoren-Konzept. Das bedeutet es müssen die Objekte angelegt, die Berechtigungsstruktur aufgestellt und anschließend das System scharf geschalten werden, da dann eine Berechtigungsvergabe einen Zugriff bedeuten würde, der einer Berechtigung bedarf – ein Henne Ei Problem, dem man nur mit einem designierten Administratoren-Konto Herr werden kann.

Das Erzeugen von Objekten stellt somit ebenfalls ein Problem dar. Ein User kann zwar ein Objekt in einem Ordner erzeugen, wenn er die notwendigen Berechtigungen dafür besitzt. Allerdings kann anschließend niemand auf diesem Objekt Berechtigungen vergeben, wenn das RBACSystem läuft. Somit wären für den realen Einsatz Wartungszeiten einzuplanen, in denen *RBACSystemDriver* außer Kraft gesetzt werden muss.

Andere Erweiterungen betreffen den RBAC-Standard. So könnten Vererbung und diverse Constraints umgesetzt werden. Die Erweiterungsliste endet nicht bei RBAC sondern könnte verschiedene Zugriffkontrollsysteme in einer Suite zusammenfassen, die nach Bedarf für unterschiedliche Ordner im Dateisystem gewählt werden können, denn schließlich basiert ein Zugriffskontrollsystem immer auf einem Referenzmonitor.

Literaturverzeichnis

[1] Ulrich B. Boddenberg. *Windows Server 2008 R2*. Galileo Computing, ISBN 978-3-8362-1528-2, 3., aktualisierte und erweiterte auflage edition, 2010.

[2] Microsoft Corporation. Msdn library, http://msdn.microsoft.com/en-us/library.

[3] Microsoft Corporation. Windows server 2003 product help, understanding groups, http://technet.microsoft.com/en-us/library/cc776995(ws.10).aspx.

[4] Microsoft Corporation. User-mode interactions: Guidelines for kernel-mode drivers. Technical report, Microsoft Corporation, 2006.

[5] David Crawford and Dave McPherson. Developing applications using windows authorization manager. Technical report, Microsoft Corporation, August 2006.

[6] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[7] Linda Ibrahim, Joe Jarzombek, Matt Ashford, Roger Bate, Paul Croll, Mary Horn, Larry Labruyere, and Curt Wells. Common criteria for information technology security evaluation, 2000.

[8] Information Technology Industry Council (ITI). *RBAC Standard, ANSI INCITS 359-2004*. American National Standards Institute, Inc. 25 West 43rd Street, New York, NY 10036, 2004.

[9] Guido Grillenmeier Jan de Clercq. *Microsoft Windows Security Fundamentals. For Windows 2003 SP1 and R2*. Butterworth Heinemann, ISBN 978-1555583408, 2006.

[10] Matunda Nyanchama and Sylvia Osborn. Access rights administration in role-based security systems. In *Database Security VIII: Status and Prospects*, pages 37–56. North-Holland, 1994.

[11] National Institute of Standards a. Technology (NIST). Role based access control (rbac) and role based security.

[12] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The nist model for role-based access control: Towards a unified standard. In *In Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, 2000.

[13] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[14] Joel Valek. Rollenbasierte zugriffskontrollsysteme anhand microsofts authorization manager. Institut für Informationsverarbeitung und Mikroprozessortechnik, Juni 2009.